Windows Process Status Helpers API

What do we have in this session?

Introduction
Process Information
Module Information
Device Driver Information
Process Memory Usage Information
Working Set Information
Memory-Mapped File Information
Using PSAPI: Program Examples
Enumerating All Processes Program Example
Enumerating All Modules for a Process Program Example
Enumerating All Device Drivers in the System Program Example
Collecting Memory Usage Information for a Process Program Example
Taking a Snapshot and Viewing Processes
PSAPI Reference
PSAPI Functions
PSAPI Structures

Introduction

The **process status application programming interface (PSAPI)** is a helper library that makes it easier for you to obtain information about processes and device drivers.
These functions are available in **Psapi.dll** (you must include the **Psapi.lib** in your project). The same information is generally available through the performance data in the registry.
The process status API (PSAPI) provides sets of functions for retrieving the following information:

1. Process Information
2. Module Information
3. Device Driver Information
4. Process Memory Usage Information
5. Working Set Information
6. Memory-Mapped File Information

The following sections explain the detail of the process information that can be extracted by PSAPI.

Process Information

The system maintains a list of running processes. You can retrieve the identifiers for these processes by calling the EnumProcesses() function. This function fills an array of DWORD values with the identifiers of all processes in the system.
Many functions in PSAPI require a process handle. To obtain a process handle for a running process, pass its process identifier (obtained from EnumProcesses()) to the OpenProcess() function. Remember to call the CloseHandle() function when you are finished with the process handle.

Module Information

A module is an **executable file** or **DLL**. Each process consists of one or more modules. You can retrieve the **list of module handles** for a process by calling the EnumProcessModules() function. This function fills an array of HMODULE values with the module handles for the specified process. The first module is the executable file. Remember that these module handles are most likely from some other process, so you cannot use them with functions such as GetModuleFileName(). However, you can use PSAPI functions to obtain information about a module from another process. The following procedure describes how to obtain module information from another process. To obtain module information from another process

1. Call the GetModuleBaseName() function. This function takes a process handle and a module handle as input and fills in a buffer with the base name of a module (for example, Kernel32.dll). A related function, GetModuleFileNameEx(), takes the same parameters as input but returns the full path to the module (for example, C:\Windows\System32\Kernel32.dll).
2. Call the GetModuleInformation() function. This function takes a process handle and a module handle and fills a MODULEINFO structure with the load address of the module, the size of the linear address space it occupies, and a pointer to its entry point.

If an application requires module information for the current process, it should use the GetModuleFileName() function instead of the PSAPI module functions. This helps application performance in two ways:

1. The GetModuleFileName() function is more efficient than the PSAPI module functions, and
2. An application can avoid loading psapi.dll if it does not use any PSAPI functions.

The GetModuleBaseName() and GetModuleFileNameEx() functions are primarily designed for use by debuggers and similar applications that must extract module information from another process. If the module list in the target process is corrupted or is not yet initialized, or if the module list changes during the function call as a result of DLLs being loaded or unloaded, these functions may fail or return incorrect information.

Device Driver Information

Device drivers and modules are similar in that they are both based on PE (Portable Executable format) files. However, while each process has its own private list of loaded modules, device drivers have modules that are global to the system. Therefore, PSAPI has specific functions for obtaining the list of device drivers and their names.
You can retrieve the **load address for each device driver** by calling the EnumDeviceDrivers() function. This function fills an array of LPVOID values with the load addresses of all device drivers in the system.
The GetDeviceDriverBaseName() function takes a driver load address as input and fills in a buffer with the base name of the driver (for example, Win32k.sys). A related function, GetDeviceDriverFileName(), takes the same parameters and returns the path to the device driver (for example, C:\Windows\System32\Win32k.sys).

Process Memory Usage Information

The GetProcessMemoryInfo() function takes a process handle as input and fills a PROCESS_MEMORY_COUNTERS structure with information about the **memory statistics for the process**. The cb member receives the size of the structure. The PageFaultCount member receives the number of page faults. The remaining members receive the current and peak memory usage in the following categories:

1. working set
2. paged pool
3. nonpaged pool
4. pagefile

The working set is the amount of memory physically mapped to the process context at a given time. Memory in the paged pool is system memory that can be transferred to the paging file on disk (paged) when it is not being used. Memory in the nonpaged pool is system memory that cannot be paged to disk as long as the corresponding objects are allocated. The pagefile usage represents how much memory is set aside for the process in the system paging file. When

memory usage is too high, the virtual memory manager pages selected memory to disk. When a thread needs a page that is not in memory, the memory manager reloads it from the paging file.

Working Set Information

The working set of a process is the amount of memory physically mapped to its process context. PSAPI enables you to take snapshots of the working set or to monitor the working set.
The QueryWorkingSet() or QueryWorkingSetEx() function fills a buffer with a snapshot of the information for every page in the current working set of the specified process. The function reports only those pages that are physically present at the exact moment it is called.
You can use working set monitoring to find out how much additional RAM a particular operation takes (for example, saving a file). To begin monitoring the working set, call the InitializeProcessForWsWatch() function. Not all processes let you read their working set information, so be sure that the function returns a nonzero value before you continue. Next, call the GetWsChanges() function. This function reports only the pages that have been loaded in memory since you began monitoring the working set. The function returns data in an array of PSAPI_WS_WATCH_INFORMATION structures, one structure for each new page added to the working set of the process. The structure tells you which pages are in memory, and what caused the system to page them in.
The EmptyWorkingSet() function takes a process handle. It removes as many pages as possible from the process working set. This operation is useful primarily for testing and tuning. Note that the SetProcessWorkingSetSize() function does the same thing if you pass it -1 for the minimum and maximum sizes.

Memory-Mapped File Information

A memory-mapped file (or file mapping) is the result of **associating a file's contents with a portion of the virtual address space of a process**. It can be used **to share a file or memory between two or more processes**.
The GetMappedFileName() function receives a process handle and a pointer to an address as input. If the address is within a memory-mapped file in the virtual address space of the process, the function returns the name of the memory-mapped file. The file names returned by GetMappedFileName() use device form, rather than drive letters. For example, the file name c:\winnt\system32\ctype.nls would look like this in device form:

```
\Device\Harddisk0\Partition1\WINNT\System32\ctype.nls
```

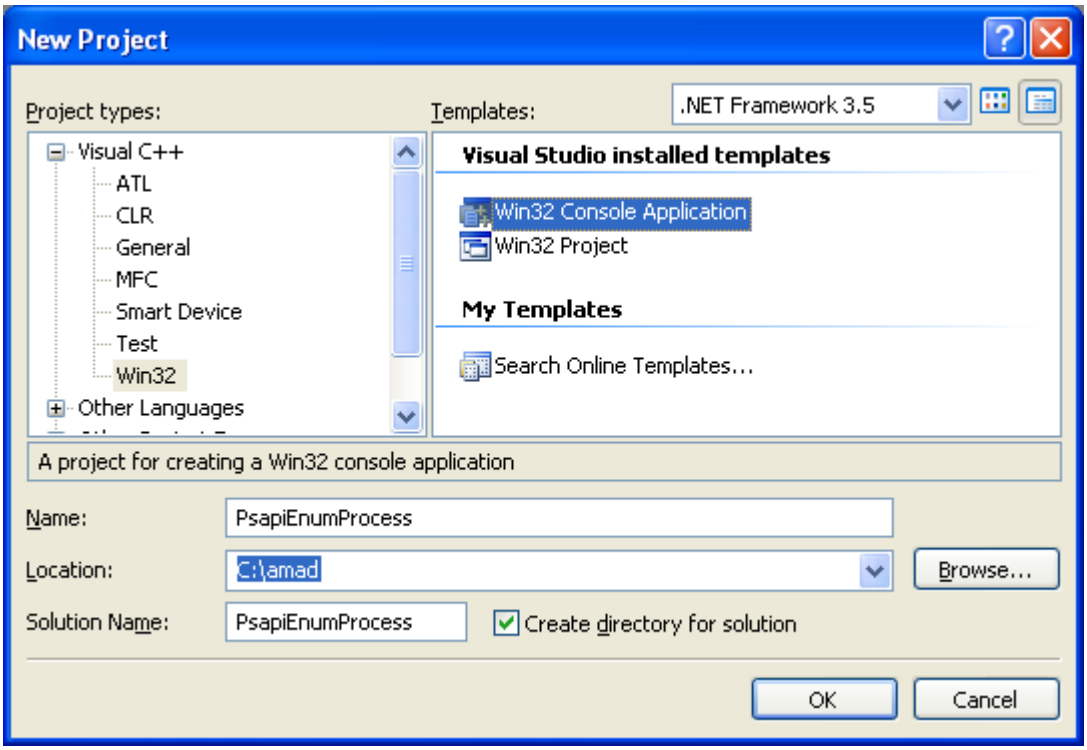
Using PSAPI: Program Examples

The following examples demonstrate how to use the PSAPI functions:

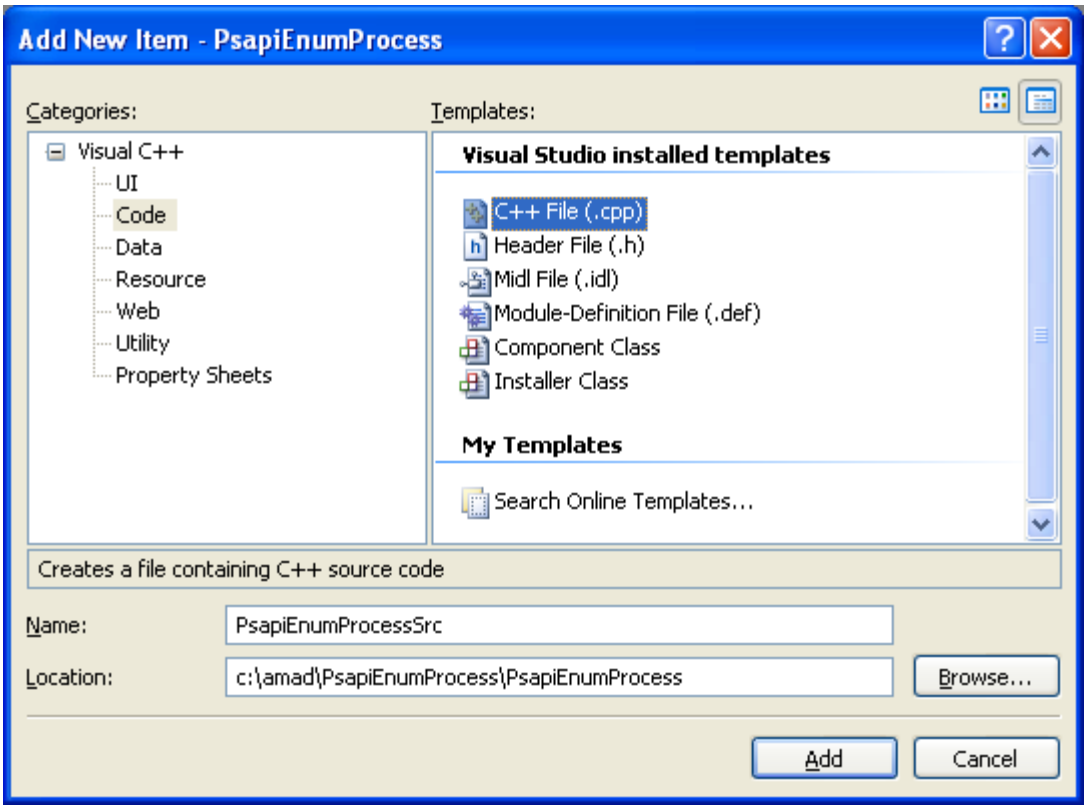
1. Enumerating all processes
2. Enumerating all modules for a process
3. Enumerating all device drivers in the system
4. Collecting memory usage information for a process
5. Taking a Snapshot and Viewing Processes

Enumerating All Processes Program Example

The following sample code uses the EnumProcesses() function to enumerate the current processes in the system.
Create new empty Win32 console application project. Give a suitable project name and change the project location if needed.



Then, add the source file and give it a suitable name.

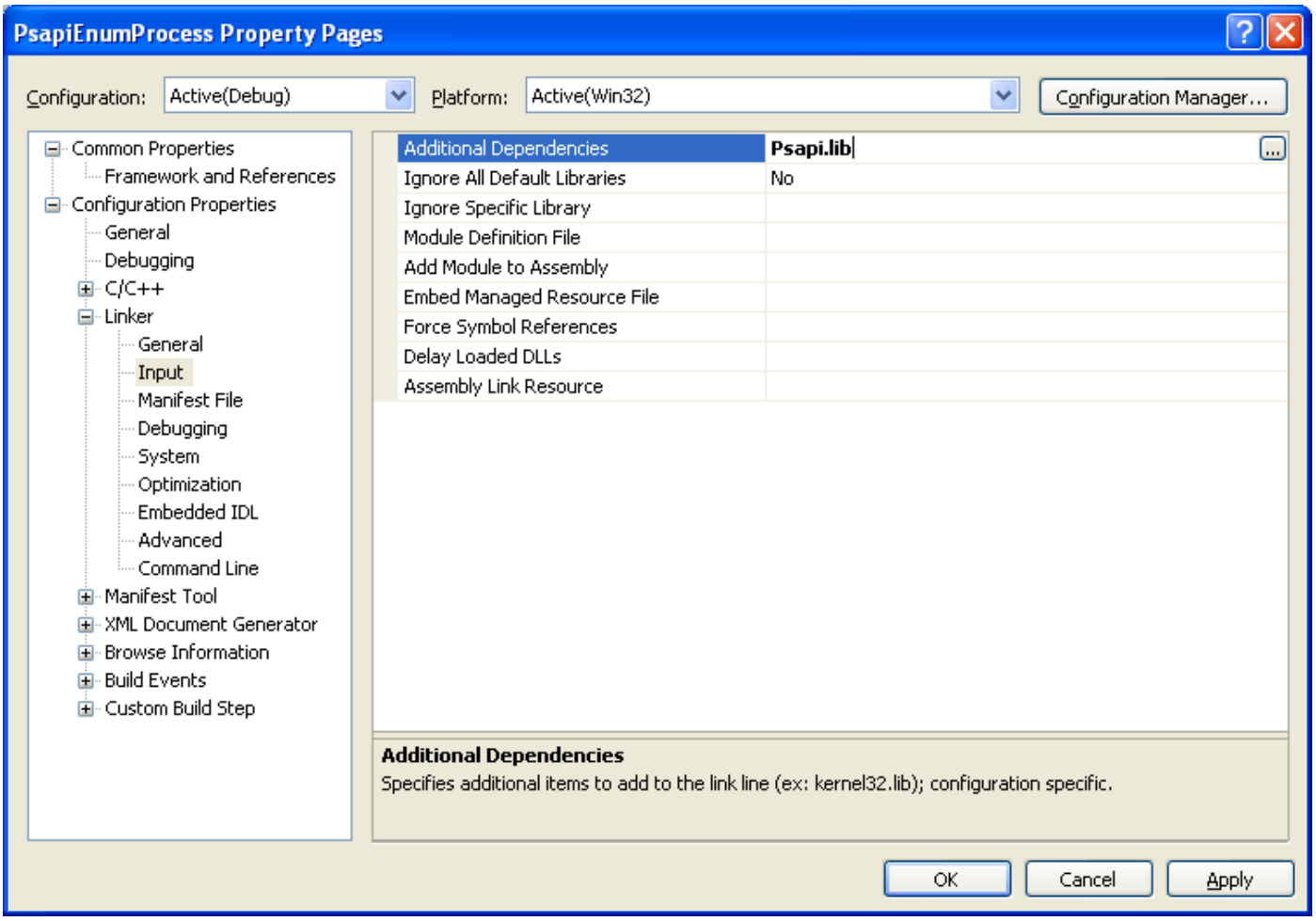


Next, add the following source code.

```
#include <windows.h>
#include <stdio.h>
// Kernel32.lib on Windows 7 and Windows Server 2008 R2,
// Psapi.lib on Windows Server 2008, Windows Vista,
// Windows Server 2003, and Windows XP/2000
#include <psapi.h>

// Another way to link to the desired library, we can use
// the following pragma directive
// #pragma comment(lib, "Psapi.lib")

void PrintProcessNameAndID(DWORD processID)
{
    WCHAR szProcessName[MAX_PATH] = L"<Unknown>";
      static int i;
      HMODULE hMod;
      DWORD cbNeeded;

    // Get a handle to the process.
    HANDLE hProcess = OpenProcess( PROCESS_QUERY_INFORMATION |
                                   PROCESS_VM_READ,
                                   FALSE, processID );

    // Get the process name.
```

```
    if (hProcess != NULL)
    {
            if (EnumProcessModules(hProcess, &hMod, sizeof(hMod), &cbNeeded))
        {
            GetModuleBaseName(hProcess, hMod, szProcessName,
sizeof(szProcessName)/sizeof(WCHAR));
        }
    }

    // Print the process name and identifier.
      wprintf(L"Process #%i: %s\t(PID: %u)\n", i, szProcessName, processID);
      i++;

      CloseHandle(hProcess);
      /*
    if(CloseHandle(hProcess) != 0)
            wprintf(L"hProcess handle was closed successfully!\n");
      else
            wprintf(L"Failed to close hProcess handle! Error %d\n",
GetLastError());
            */
}

int wmain(int argc, WCHAR **argv)
{
    // Get the list of process identifiers.
    DWORD aProcesses[1024], cbNeeded, cProcesses;
    unsigned int i;

    if (!EnumProcesses(aProcesses, sizeof(aProcesses), &cbNeeded))
        return 1;
      else
            wprintf(L"EnumProcesses() is OK!\n");

    // Calculate how many process identifiers were returned.
    cProcesses = cbNeeded / sizeof(DWORD);

    // Print the name and process identifier for each process.
    for (i = 0; i < cProcesses; i++)
        if(aProcesses[i] != 0)
            PrintProcessNameAndID(aProcesses[i]);

      return 0;
}
```

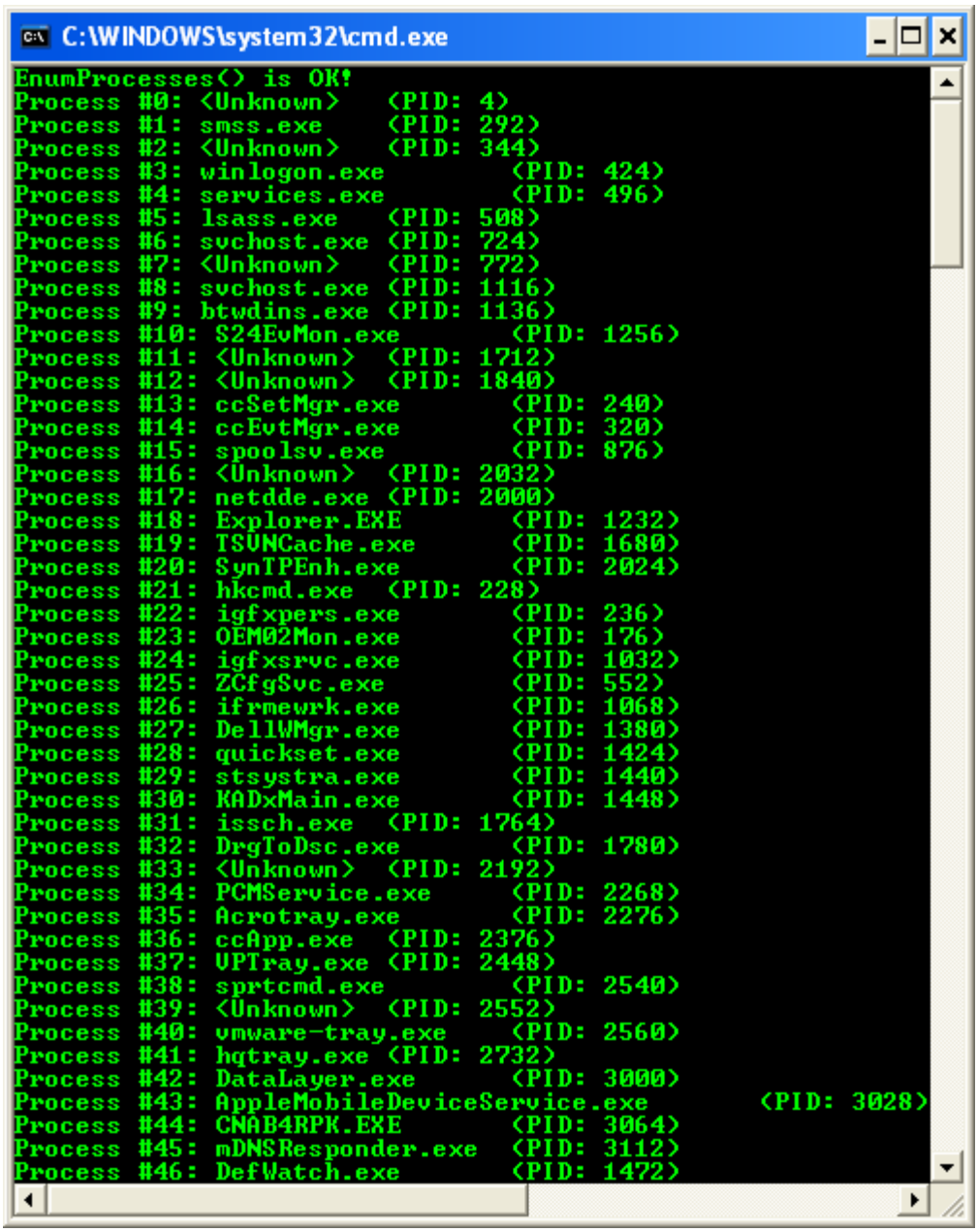
Add the Additional Dependencies.



Another way to link to the library, we can use the #pragma directive and the following is for this example.

```
#pragma comment(lib, "Psapi.lib")
```

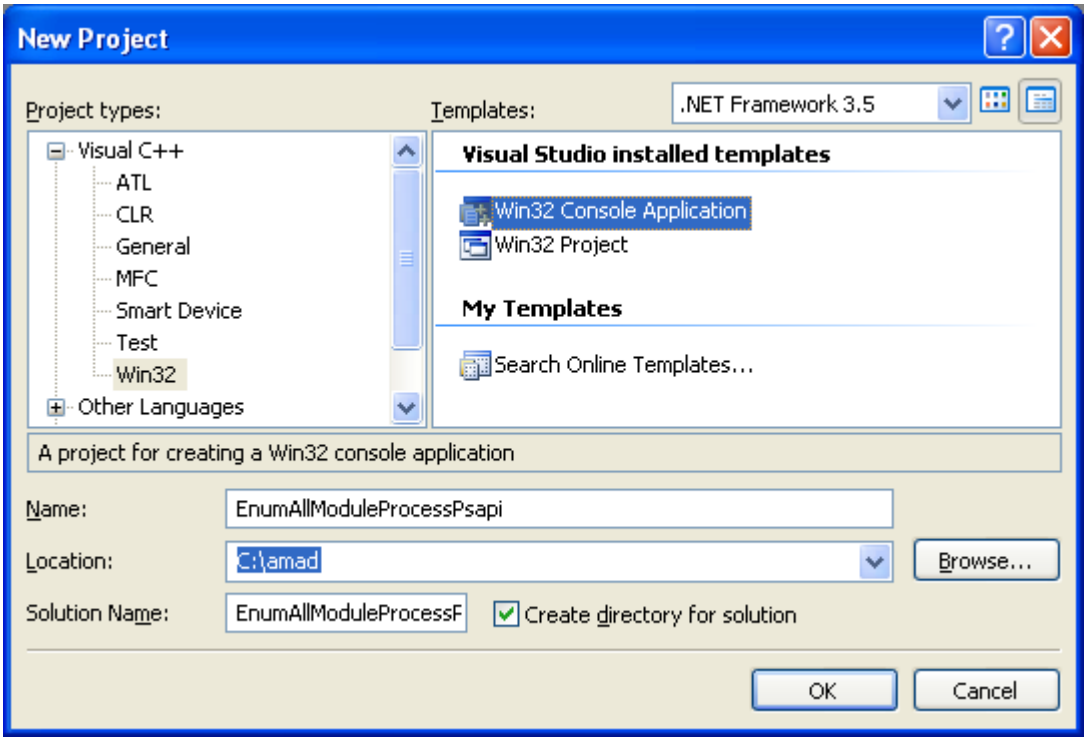
Build and run the project. The following screenshot is a sample output.



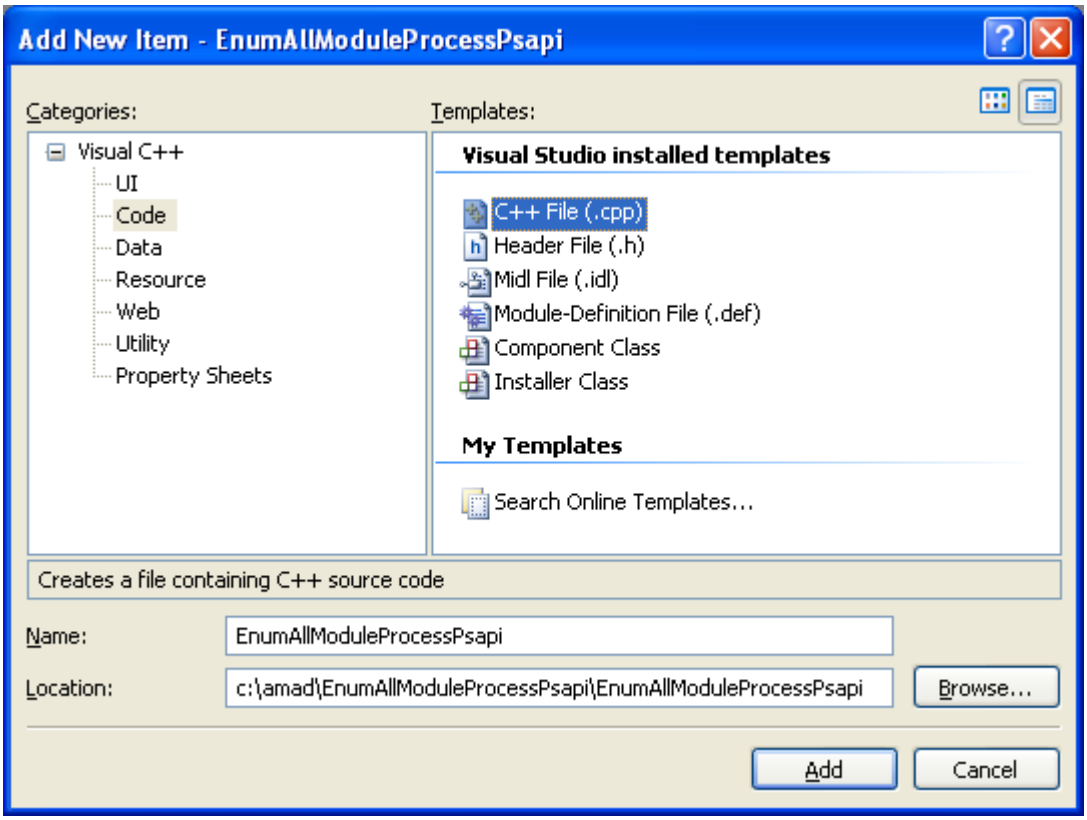
The main function obtains a list of processes by using the EnumProcesses() function. For each process, main calls the PrintProcessNameAndID() function, passing it the process identifier. PrintProcessNameAndID() in turn calls the OpenProcess() function to obtain the process handle. If OpenProcess() fails, the output shows the process name as <unknown>. For example, OpenProcess() fails for the Idle and CSRSS processes because their access restrictions prevent user-level code from opening them. Next, PrintProcessNameAndID() calls the EnumProcessModules() function to obtain the module handles. Finally, PrintProcessNameAndID() calls the GetModuleBaseName() function to obtain the name of the executable file and displays the name along with the process identifier.

Enumerating All Modules for a Process Program Example

To determine which processes have loaded a particular DLL, you must enumerate the modules for each process. The following sample code uses the EnumProcessModules() function to enumerate the modules of current processes in the system.
Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.



Then, add the source file and give it a suitable name.

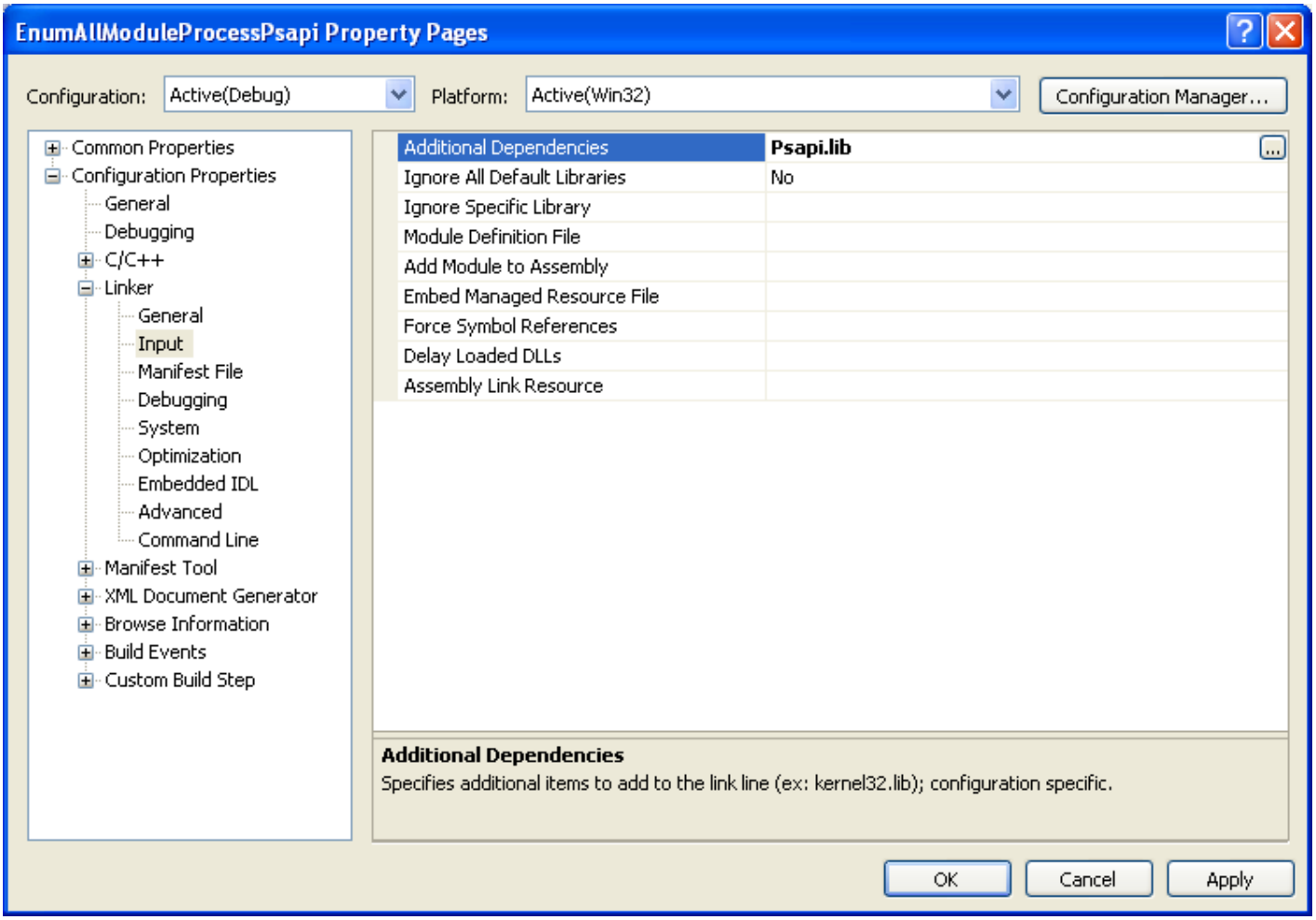


Next, add the following source code.

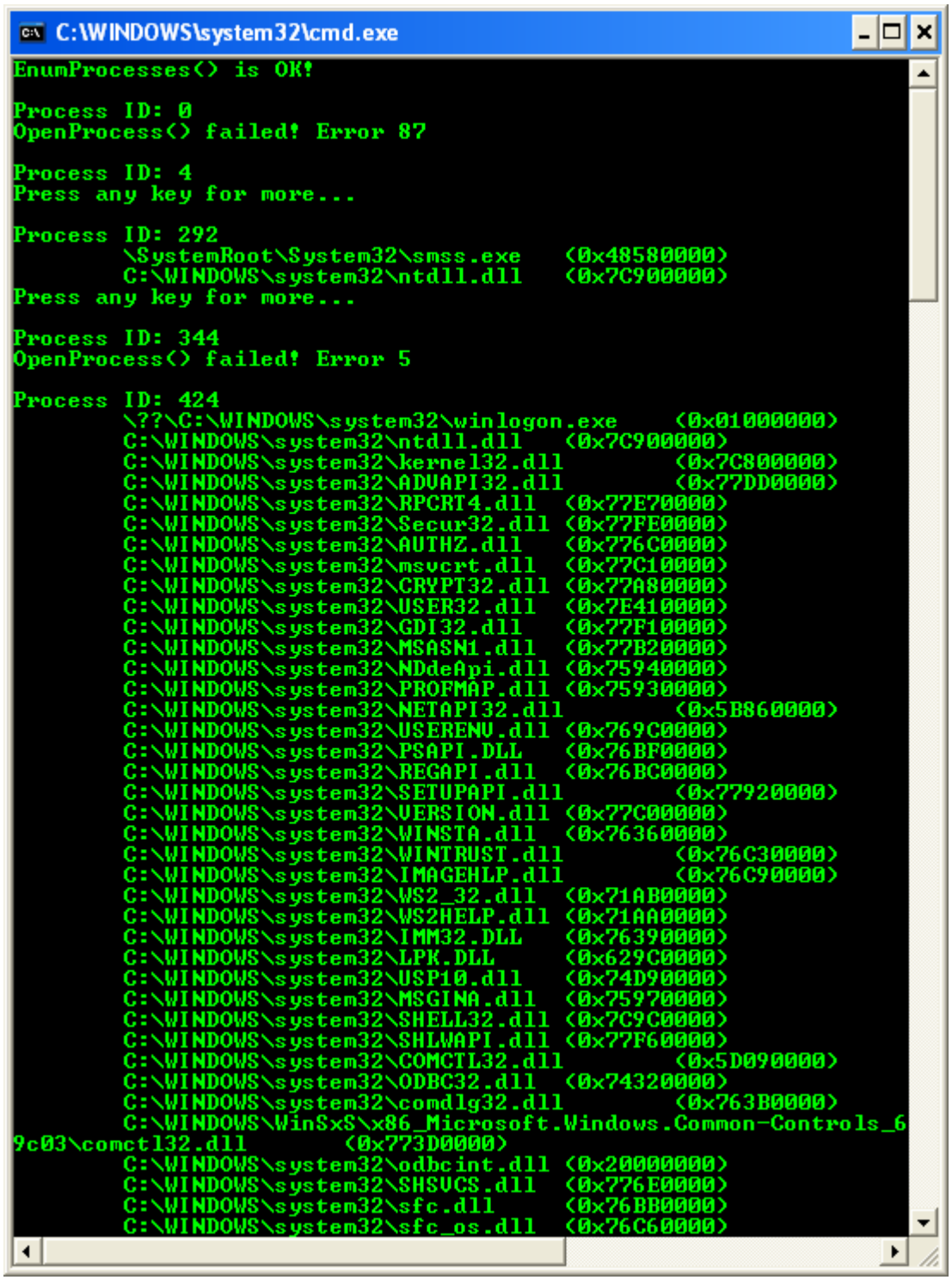
```
// Link to Psapi.lib
#include <windows.h>
#include <stdio.h>
#include <wchar.h>
#include <psapi.h>

void PrintModules(DWORD processID)
{
    HMODULE hMods[1024];
    HANDLE hProcess;
    DWORD cbNeeded;
    unsigned int i;
      WCHAR szModName[MAX_PATH];

    // Print the process identifier.
    wprintf(L"\nProcess ID: %u\n", processID);

    // Get a list of all the modules in this process.
    hProcess = OpenProcess( PROCESS_QUERY_INFORMATION |
                            PROCESS_VM_READ, FALSE, processID );
    if (hProcess == NULL)
      {
            wprintf(L"OpenProcess() failed! Error %d\n", GetLastError());
        return;
```

```
      }

    if(EnumProcessModules(hProcess, hMods, sizeof(hMods), &cbNeeded))
    {
        for(i = 0; i < (cbNeeded / sizeof(HMODULE)); i++)
        {
            // Get the full path to the module's file.
            if(GetModuleFileNameEx(hProcess, hMods[i], szModName,
sizeof(szModName) / sizeof(WCHAR)))
            {
                // Print the module name and handle value.
                wprintf(L"\t%s\t(0x%08X)\n", szModName, hMods[i]);
            }
        }
    }
    CloseHandle(hProcess);
      wprintf(L"Press any key for more...\n");
      _getwch();
}

int main(int argc, WCHAR **argv)
{
    // Get the list of process identifiers.
    DWORD aProcesses[1024], cbNeeded, cProcesses;
    unsigned int i;

    if (!EnumProcesses(aProcesses, sizeof(aProcesses), &cbNeeded))
        return 1;
      else
            wprintf(L"EnumProcesses() is OK!\n");

    // Calculate how many process identifiers were returned.
    cProcesses = cbNeeded / sizeof(DWORD);

    // Print the name of the modules for each process.
    for (i = 0; i < cProcesses; i++)
        PrintModules(aProcesses[i]);

      return 0;
}
```

Add the Additional Dependencies.



Build and run the project. The following screenshot is a sample output.

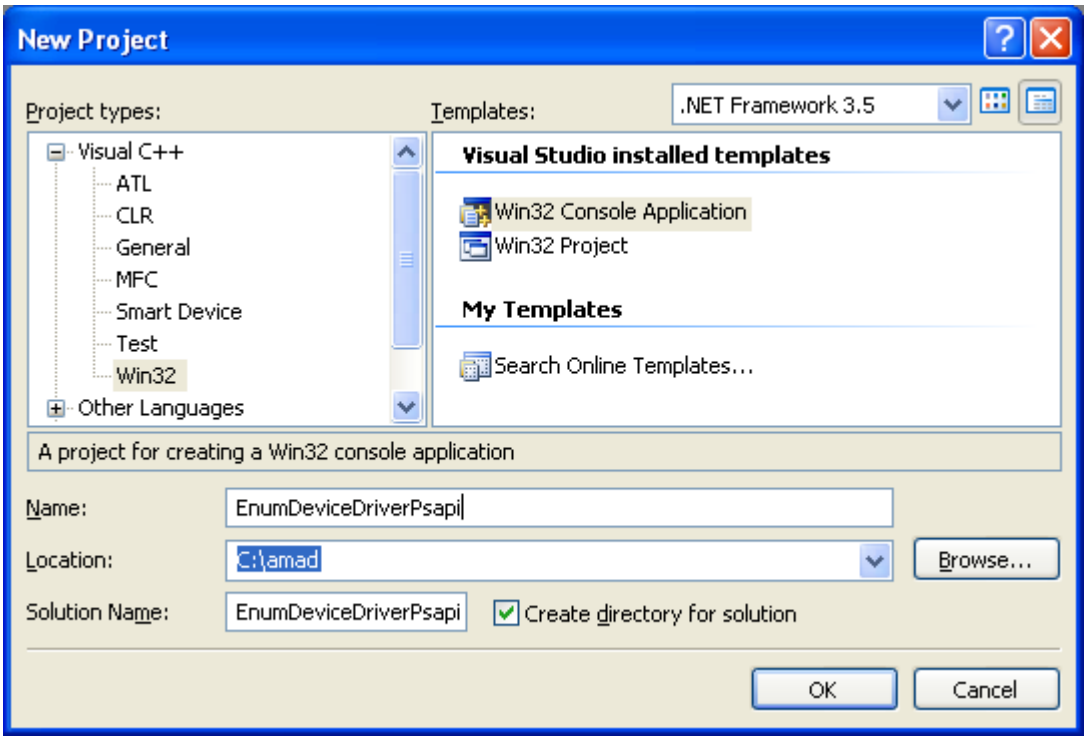


The main function obtains a list of processes by using the EnumProcesses() function. For each process, the main function calls the PrintModules() function, passing it the process identifier. PrintModules() in turn calls the OpenProcess() function to obtain the process handle. If

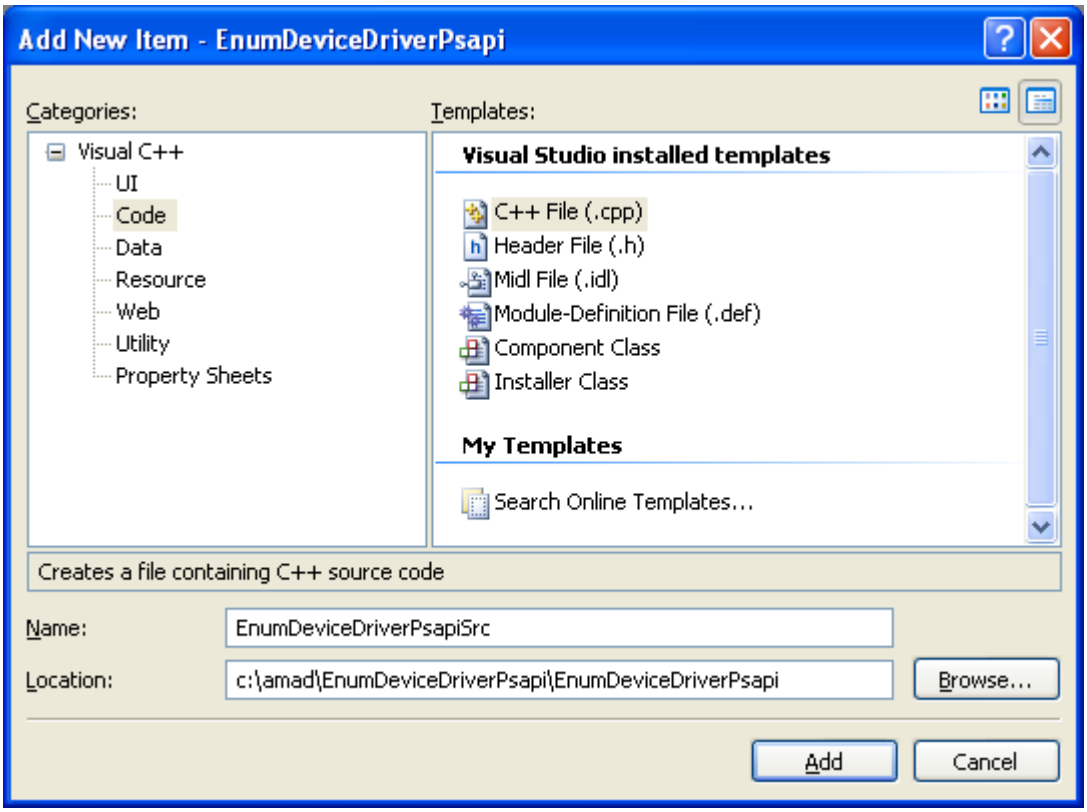
OpenProcess() fails, the output shows only the process identifier. For example, OpenProcess() fails for the Idle and CSRSS processes because their access restrictions prevent user-level code from opening them. Next, PrintModules() calls the EnumProcessModules() function to obtain the module handles function. Finally, PrintModules() calls the GetModuleFileNameEx() function, once for each module, to obtain the module names.

Enumerating All Device Drivers in the System Program Example

The following sample code uses the EnumDeviceDrivers() function to enumerate the current device drivers in the system. It passes the load addresses retrieved from this function call to the GetDeviceDriverBaseName() function to retrieve a name that can be displayed.
Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.



Then, add the source file and give it a suitable name.

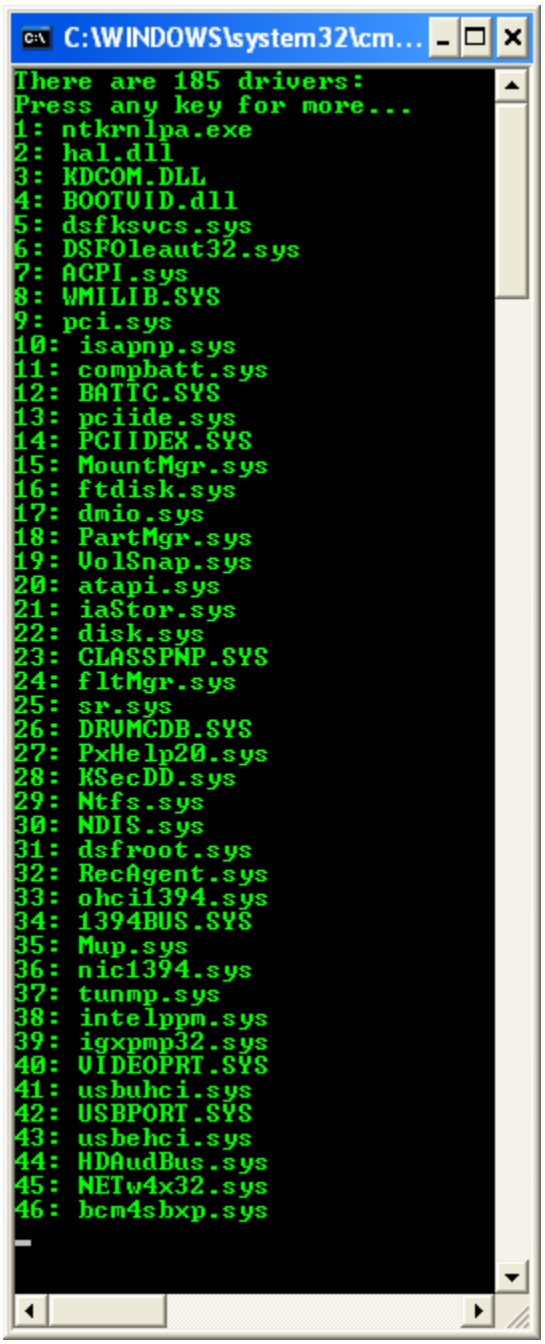


Next, add the following source code.

```
#include <windows.h>
#include <psapi.h>
#include <wchar.h>
#include <stdio.h>

// Link to Psapi.lib
#pragma comment(lib, "Psapi.lib")

#define ARRAY_SIZE 1024

int wmain(int argc, WCHAR **argv)
{
   LPVOID drivers[ARRAY_SIZE];
   DWORD cbNeeded;
   int cDrivers, i;
   WCHAR szDriver[ARRAY_SIZE];

   if(EnumDeviceDrivers(drivers, sizeof(drivers), &cbNeeded) && cbNeeded <
sizeof(drivers))
   {
         cDrivers = cbNeeded / sizeof(drivers[0]);

      wprintf(L"There are %d drivers:\n", cDrivers);
        wprintf(L"Press any key for more...\n");
```

```
      for (i=0; i < cDrivers; i++)
      {
         if(GetDeviceDriverBaseName(drivers[i], szDriver, sizeof(szDriver) /
sizeof(szDriver[0])))
            wprintf(L"%d: %s\n", i+1, szDriver);
             _getwch();
      }
   }
   else
         wprintf(L"EnumDeviceDrivers() failed; array size needed is %d\n",
cbNeeded / sizeof(LPVOID));

   return 0;
}
```

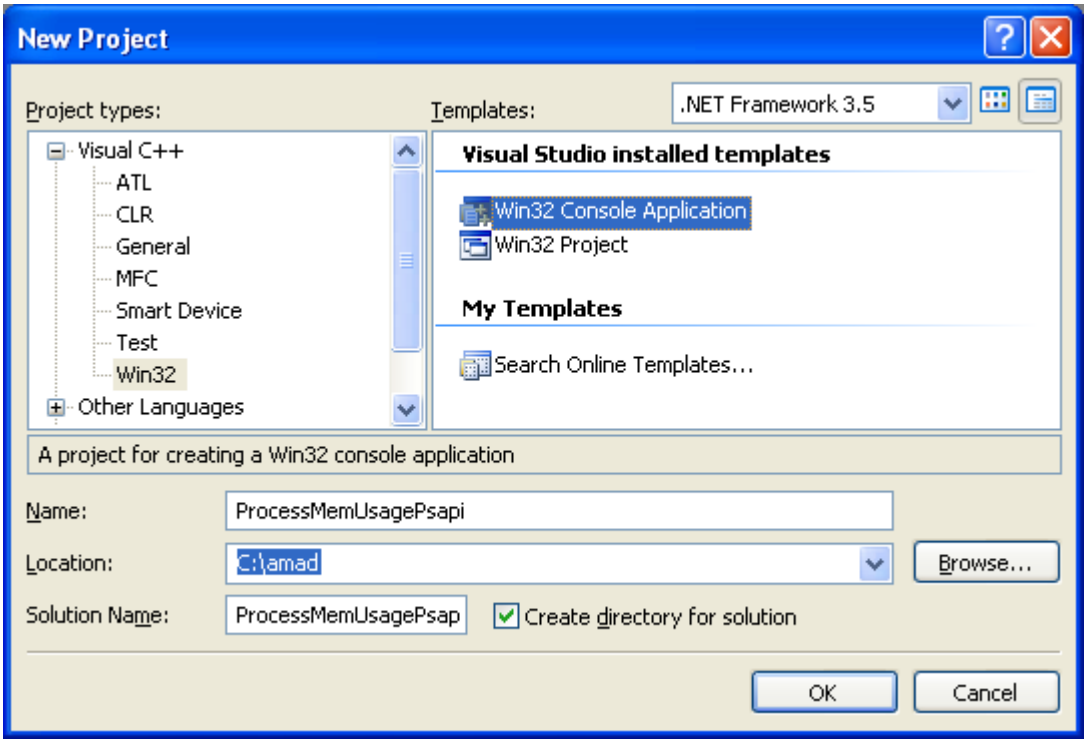
Build and run the project. The following screenshot is a sample output.



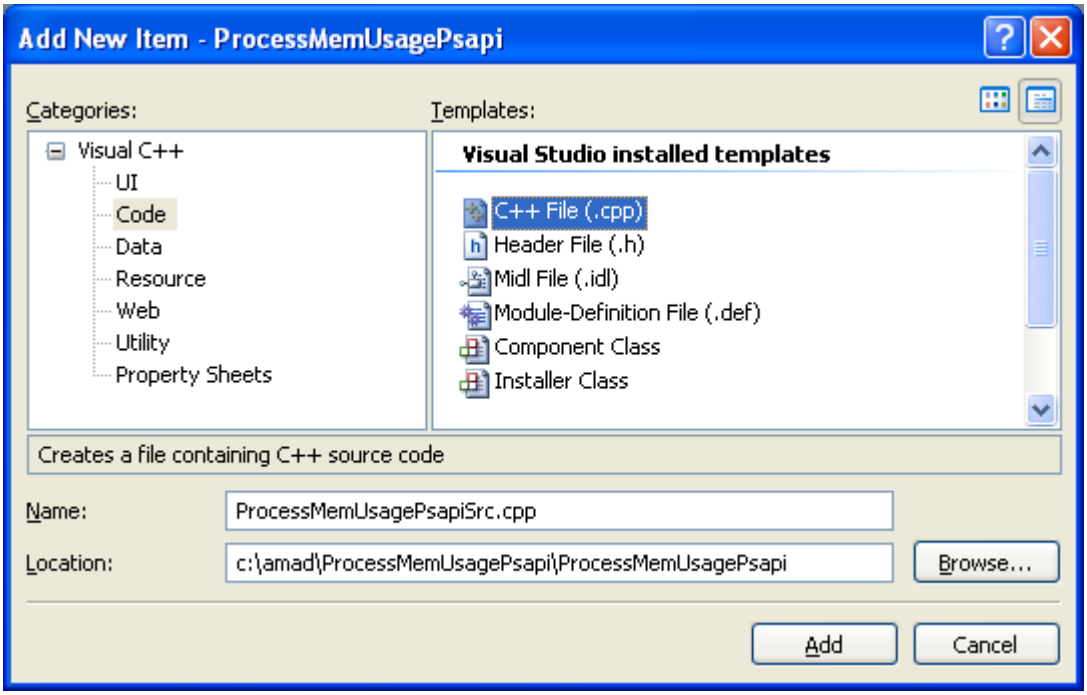
Collecting Memory Usage Information for a Process Program Example

To determine the efficiency of your application, you may want to examine its memory usage. The following sample code uses the GetProcessMemoryInfo() function to obtain information about the memory usage of a process.

Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.



Then, add the source file and give it a suitable name.

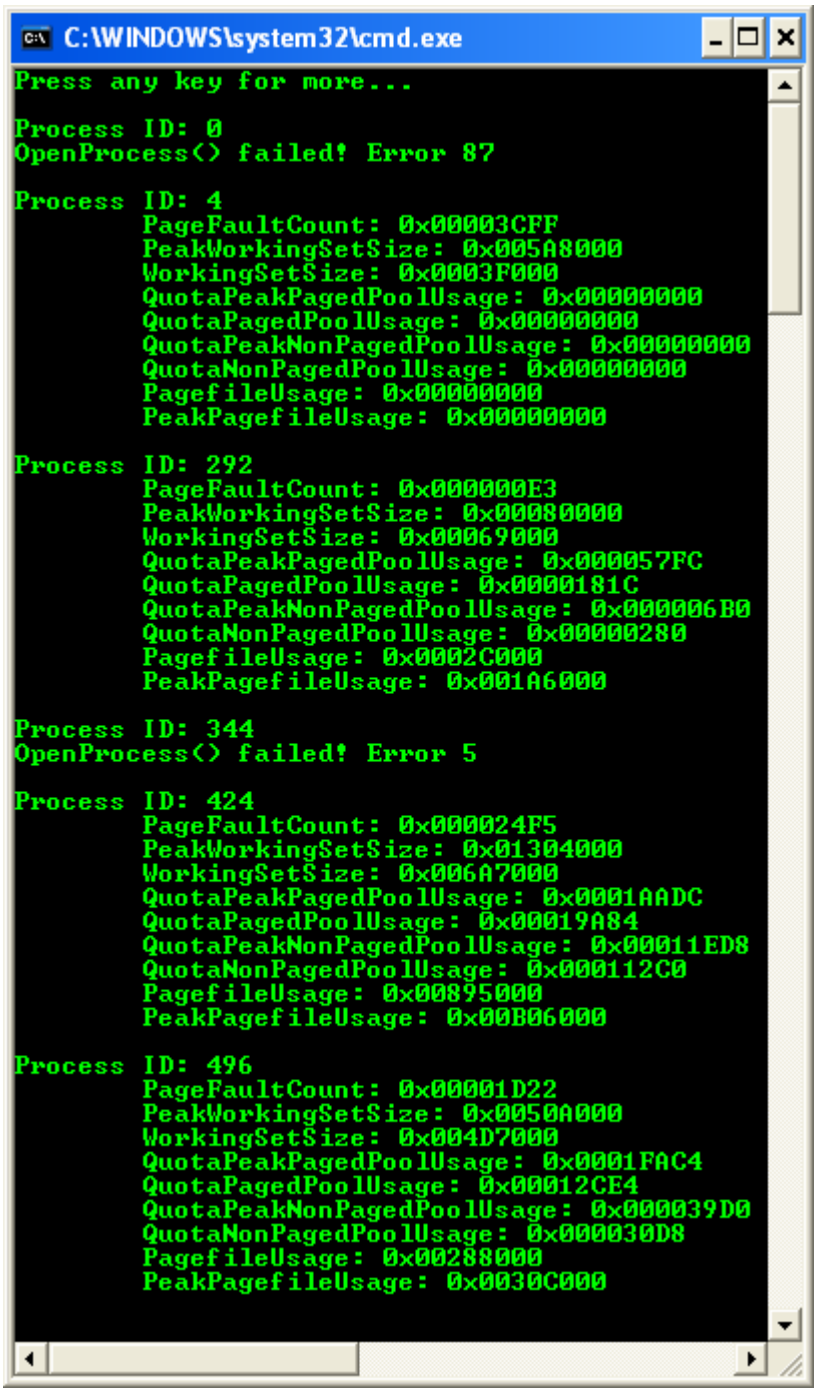


Next, add the following source code.

```
#include <windows.h>
#include <stdio.h>
#include <wchar.h>
#include <psapi.h>

// Link to Psapi.lib
#pragma comment(lib, "Psapi.lib")

void PrintMemoryInfo(DWORD processID)
{
    HANDLE hProcess;
    PROCESS_MEMORY_COUNTERS pmc;

    // Print the process identifier.
    printf( "\nProcess ID: %u\n", processID );

    // Print information about the memory usage of the process.
    hProcess = OpenProcess(  PROCESS_QUERY_INFORMATION |
                                    PROCESS_VM_READ,FALSE, processID );
    if (hProcess == NULL)
      {
            wprintf(L"OpenProcess() failed! Error %d\n", GetLastError());
        return;
      }

    if (GetProcessMemoryInfo(hProcess, &pmc, sizeof(pmc)))
    {
        wprintf(L"\tPageFaultCount: 0x%08X\n", pmc.PageFaultCount);
        wprintf(L"\tPeakWorkingSetSize: 0x%08X\n",  pmc.PeakWorkingSetSize);
        wprintf(L"\tWorkingSetSize: 0x%08X\n", pmc.WorkingSetSize);
        wprintf(L"\tQuotaPeakPagedPoolUsage: 0x%08X\n",
pmc.QuotaPeakPagedPoolUsage);
        wprintf(L"\tQuotaPagedPoolUsage: 0x%08X\n", pmc.QuotaPagedPoolUsage);
        wprintf(L"\tQuotaPeakNonPagedPoolUsage: 0x%08X\n",
pmc.QuotaPeakNonPagedPoolUsage);
        wprintf(L"\tQuotaNonPagedPoolUsage: 0x%08X\n",
pmc.QuotaNonPagedPoolUsage);
        wprintf(L"\tPagefileUsage: 0x%08X\n", pmc.PagefileUsage);
        wprintf(L"\tPeakPagefileUsage: 0x%08X\n", pmc.PeakPagefileUsage);
    }

    CloseHandle(hProcess);
}

int wmain(int argc, WCHAR **argv)
{
    // Get the list of process identifiers.
    DWORD aProcesses[1024], cbNeeded, cProcesses;
    unsigned int i;

    if (!EnumProcesses(aProcesses, sizeof(aProcesses), &cbNeeded))
        return 1;

    // Calculate how many process identifiers were returned.
```

```
    cProcesses = cbNeeded / sizeof(DWORD);

    // Print the memory usage for each process
      wprintf(L"Press any key for more...\n");
    for (i = 0; i < cProcesses; i++)
      {
        PrintMemoryInfo(aProcesses[i]);
            _getwch();
      }

      return 0;
}
```

Build and run the project. The following screenshot is a sample output.

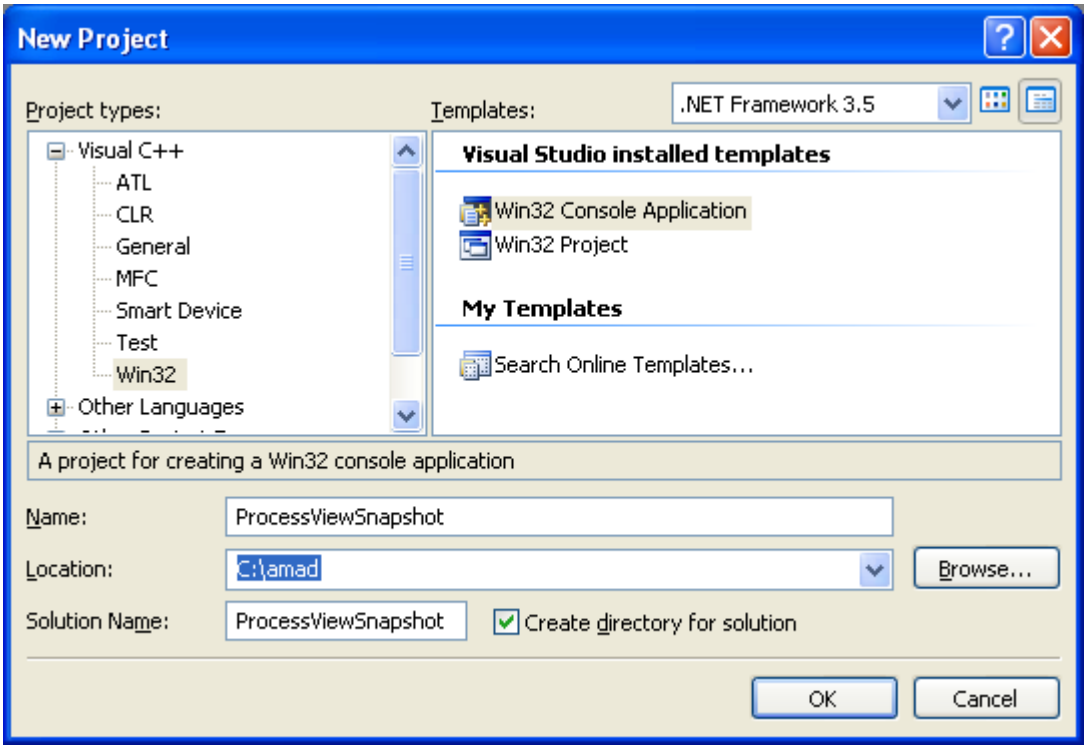


The main function obtains a list of processes by using the EnumProcesses() function. For each process, main calls the PrintMemoryInfo() function, passing the process identifier. PrintMemoryInfo() in turn calls the OpenProcess() function to obtain the process handle. If OpenProcess() fails, the output shows only the process identifier. For example, OpenProcess() fails for the Idle and CSRSS processes because their access restrictions prevent user-level code

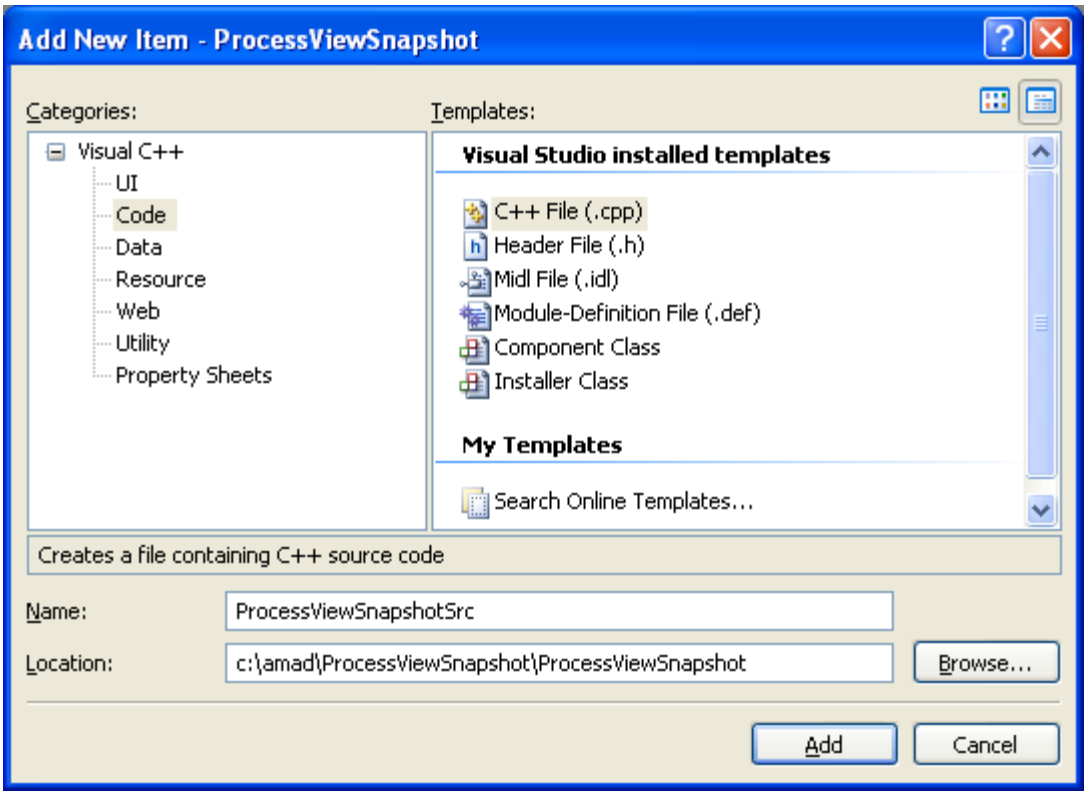
from opening them. Finally, PrintMemoryInfo() calls the GetProcessMemoryInfo() function to obtain the memory usage information.

Taking a Snapshot and Viewing Processes Program Example

The following simple console application obtains a list of running processes. First, the GetProcessList() function takes a snapshot of currently executing processes in the system using CreateToolhelp32Snapshot(), and then it walks through the list recorded in the snapshot using Process32First() and Process32Next(). For each process in turn, GetProcessList() calls the ListProcessModules() and the ListProcessThreads().
A simple error-reporting function, printError(), displays the reason for any failures, which usually result from security restrictions. For example, OpenProcess() fails for the Idle and CSRSS processes because their access restrictions prevent user-level code from opening them.
Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.



Then, add the source file and give it a suitable name.



Next, add the following source code.

```
// Taking a Snapshot and Viewing Processes
#include <windows.h>
#include <tlhelp32.h>
#include <tchar.h>
#include <stdio.h>

//  Prototypes
BOOL GetProcessList();
BOOL ListProcessModules(DWORD dwPID);
BOOL ListProcessThreads(DWORD dwOwnerPID);
void printError(WCHAR* msg);

int wmain(int argc, WCHAR **argv)
{
  GetProcessList();

  return 0;
}

BOOL GetProcessList()
{
  HANDLE hProcessSnap;
  HANDLE hProcess;
  PROCESSENTRY32 pe32;
  DWORD dwPriorityClass;
```

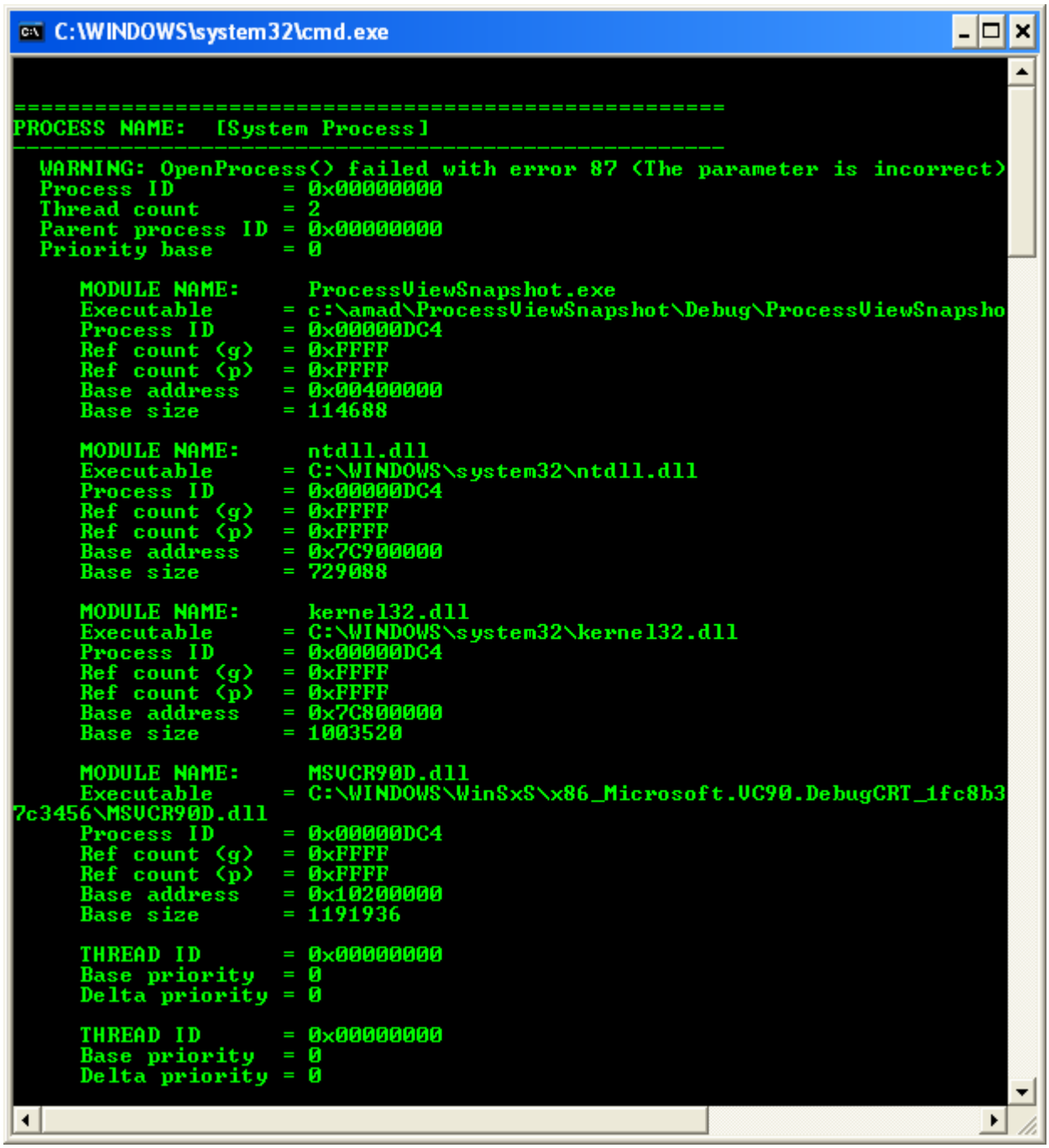
```
// Take a snapshot of all processes in the system.
hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
if(hProcessSnap == INVALID_HANDLE_VALUE)
{
  printError(L"CreateToolhelp32Snapshot() (of processes)");
  return (FALSE);
}

// Set the size of the structure before using it.
pe32.dwSize = sizeof(PROCESSENTRY32);

// Retrieve information about the first process,
// and exit if unsuccessful
if(!Process32First(hProcessSnap, &pe32))
{
  printError(L"Process32First()"); // show cause of failure
  CloseHandle(hProcessSnap);     // clean the snapshot object
  return (FALSE);
}

// Now walk the snapshot of processes, and
// display information about each process in turn
do
{
  printf( "\n\n=====================================================" );
  _tprintf( TEXT("\nPROCESS NAME:  %s"), pe32.szExeFile );
  printf( "\n-----------------------------------------------------" );

  // Retrieve the priority class.
  dwPriorityClass = 0;
  hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pe32.th32ProcessID);
  if(hProcess == NULL)
    printError(L"OpenProcess()");
  else
  {
    dwPriorityClass = GetPriorityClass(hProcess);
    if(!dwPriorityClass)
      printError(L"GetPriorityClass()");
    CloseHandle(hProcess);
  }

  wprintf(L"\n  Process ID        = 0x%08X", pe32.th32ProcessID);
  wprintf(L"\n  Thread count      = %d",   pe32.cntThreads);
  wprintf(L"\n  Parent process ID = 0x%08X", pe32.th32ParentProcessID);
  wprintf(L"\n  Priority base     = %d", pe32.pcPriClassBase );
  if(dwPriorityClass)
    wprintf(L"\n  Priority class    = %d", dwPriorityClass);

  // List the modules and threads associated with this process
  ListProcessModules(pe32.th32ProcessID);
  ListProcessThreads(pe32.th32ProcessID);

    // Press any key for more...
    _getwch();
```

```
  } while(Process32Next(hProcessSnap, &pe32));

  CloseHandle(hProcessSnap);
  return (TRUE);
}

BOOL ListProcessModules(DWORD dwPID)
{
  HANDLE hModuleSnap = INVALID_HANDLE_VALUE;
  MODULEENTRY32 me32;

  // Take a snapshot of all modules in the specified process.
  hModuleSnap = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, dwPID);
  if(hModuleSnap == INVALID_HANDLE_VALUE)
  {
    printError(L"CreateToolhelp32Snapshot() (of modules)");
    return (FALSE);
  }

  // Set the size of the structure before using it.
  me32.dwSize = sizeof(MODULEENTRY32);

  // Retrieve information about the first module,
  // and exit if unsuccessful
  if(!Module32First(hModuleSnap, &me32))
  {
    printError(L"Module32First()");  // show cause of failure
    CloseHandle(hModuleSnap);        // clean the snapshot object
    return (FALSE);
  }

  // Now walk the module list of the process,
  // and display information about each module
  do
  {
    wprintf(L"\n\n     MODULE NAME:     %s",   me32.szModule);
    wprintf(L"\n     Executable     = %s",     me32.szExePath);
    wprintf(L"\n     Process ID     = 0x%08X",         me32.th32ProcessID);
    wprintf(L"\n     Ref count (g)  = 0x%04X",     me32.GlblcntUsage);
    wprintf(L"\n     Ref count (p)  = 0x%04X",     me32.ProccntUsage);
    wprintf(L"\n     Base address   = 0x%08X", (DWORD) me32.modBaseAddr);
    wprintf(L"\n     Base size      = %d",             me32.modBaseSize);

  } while(Module32Next(hModuleSnap, &me32));

  CloseHandle(hModuleSnap);
  return (TRUE);
}

BOOL ListProcessThreads(DWORD dwOwnerPID)
{
  HANDLE hThreadSnap = INVALID_HANDLE_VALUE;
  THREADENTRY32 te32;
```

```
  // Take a snapshot of all running threads
  hThreadSnap = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);
  if(hThreadSnap == INVALID_HANDLE_VALUE)
    return (FALSE);

  // Fill in the size of the structure before using it.
  te32.dwSize = sizeof(THREADENTRY32);

  // Retrieve information about the first thread,
  // and exit if unsuccessful
  if(!Thread32First(hThreadSnap, &te32))
  {
    printError(L"Thread32First()"); // show cause of failure
    CloseHandle(hThreadSnap);     // clean the snapshot object
    return (FALSE);
  }

  // Now walk the thread list of the system,
  // and display information about each thread
  // associated with the specified process
  do
  {
    if(te32.th32OwnerProcessID == dwOwnerPID)
    {
      wprintf(L"\n\n     THREAD ID      = 0x%08X", te32.th32ThreadID);
      wprintf(L"\n     Base priority  = %d", te32.tpBasePri);
      wprintf(L"\n     Delta priority = %d", te32.tpDeltaPri);
    }
  } while(Thread32Next(hThreadSnap, &te32));

  CloseHandle(hThreadSnap);
  return (TRUE);
}

void printError(WCHAR* msg)
{
  DWORD eNum;
  WCHAR sysMsg[256];
  WCHAR* p;

  eNum = GetLastError();
  FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS,
         NULL, eNum,
         MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
         sysMsg, 256, NULL );

  // Trim the end of the line and terminate it with a null
  p = sysMsg;
  while( ( *p > 31 ) || ( *p == 9 ) )
    ++p;
  do { *p-- = 0; } while( ( p >= sysMsg ) &&
                          ( ( *p == '.' ) || ( *p < 33 ) ) );

  // Display the message
  wprintf(L"\n  WARNING: %s failed with error %d (%s)", msg, eNum, sysMsg );
```

```
}
```

Build and run the project. The following screenshot is a sample output. Press any key for more Module information.



PSAPI Reference

The following sections list the PSAPI functions and structures

PSAPI Functions

The following are the PSAPI functions.

Function	Description
EmptyWorkingSet()	- Removes as many pages as possible from the working set of the specified process.
EnumDeviceDrivers()	- Retrieves the load address for each device driver in the system.
EnumPageFiles()	- Calls the callback routine for each installed pagefile in the system.
EnumProcesses()	- Retrieves the process identifier for each process object in the system.
EnumProcessModules()	- Retrieves a handle for each module in the specified process. To control whether a 64-bit application enumerates 32-bit modules, 64-bit modules, or both types of modules, use the EnumProcessModulesEx() function.
EnumProcessModulesEx()	- Retrieves a handle for each module in the specified process that meets the specified filter criteria.
GetDeviceDriverBaseName()	- Retrieves the base name of the specified device driver.
GetDeviceDriverFileName()	- Retrieves the path available for the specified device driver.
GetMappedFileName()	- Checks whether the specified address is within a memory-mapped file in the address space of the specified process. If so, the function returns the name of the memory-mapped file.
GetModuleBaseName()	- Retrieves the base name of the specified module.
GetModuleFileNameEx()	- Retrieves the fully-qualified path for the file containing the specified module.
GetModuleInformation()	- Retrieves information about the specified module in the MODULEINFO structure.
GetPerformanceInfo()	- Retrieves the performance values contained in the PERFORMANCE_INFORMATION structure.
GetProcessImageFileName()	- Retrieves the name of the executable file for the specified process.
GetProcessMemoryInfo()	- Retrieves information about the memory usage of the specified process.
GetWsChanges()	- Retrieves information about the pages that have been added to the working set of the specified process since the last time this

	function or the InitializeProcessForWsWatch() function was called. To retrieve extended information, use the GetWsChangesEx() function.
GetWsChangesEx()	- Retrieves extended information about the pages that have been added to the working set of the specified process since the last time this function or the InitializeProcessForWsWatch() function was called.
InitializeProcessForWsWatch()	- Initiates monitoring of the working set of the specified process. You must call this function before calling the GetWsChanges() function.
QueryWorkingSet()	- Retrieves information about the pages currently added to the working set of the specified process.
QueryWorkingSetEx()	- Retrieves extended information about the pages currently added to the working set of the specified process.

PSAPI Structures

The following are the PSAPI structures.

Function	Description
ENUM_PAGE_FILE_INFORMATION	- Contains information about a pagefile.
MODULEINFO	- Contains the module load address, size, and entry point.
PERFORMANCE_INFORMATION	- Contains performance information.
PROCESS_MEMORY_COUNTERS	- Contains the memory statistics for a process.
PROCESS_MEMORY_COUNTERS_EX	- Contains extended memory statistics for a process.
PSAPI_WORKING_SET_BLOCK	- Contains working set information for a page.
PSAPI_WORKING_SET_EX_BLOCK	- Contains extended working set information for a page.
PSAPI_WORKING_SET_EX_INFORMATION	- Contains extended working set information for a process.
PSAPI_WORKING_SET_INFORMATION	- Contains working set information for a process.
PSAPI_WS_WATCH_INFORMATION	- Contains information about a page added to a process working set.
PSAPI_WS_WATCH_INFORMATION_EX	- Contains extended information about a page

	added to a process working set.