Snooping System Information

CHAPTER

14

IN THIS CHAPTER

In this chapter, you'll learn how to create a full-featured utility, called SysInfo, that's designed to browse the vital parameters of your system. Through the course of developing this application, you'll learn how to employ lesser-known APIs to gain access to low-level, systemwide information on processes, threads, modules, heaps, drivers, and pages. This chapter also covers how Windows 95/98 and Windows NT obtain this information differently. Additionally, SysInfo provides you with techniques for obtaining information on free memory resources, Windows version information, environment variable settings, and a list of loaded modules. Not only do you learn to use these nuts-and-bolts API functions, but you also learn how to integrate this information into a functional and aesthetically pleasing user interface. Additionally, you learn which of the Windows 3.*x* API functions the Win32 functions in this chapter are designed to replace.

You'd want to get such information from Windows for several reasons. Of course, the hacker in each of us would argue that being able to snoop around the operating system's backyard like some kind of cyber-voyeur is its own reward. Perhaps you're writing a program that needs to access environment variables in order to find certain files. Maybe you need to determine which modules are loaded in order to remove modules from memory manually. Possibly you need to come up with a killer chapter for a book you're writing. See, lots of valid reasons exist!

InfoForm: Obtaining General Information

To warm up, this section shows you how to obtain system information in an API that's consistent across Win32 versions. The code for this application will make a bit more sense if you learn about its user interface first. You'll learn about the user interface of this application a little bit backward, though, because we're going to explain one of the application's child forms first. This form, shown in Figure 14.1, is called `InfoForm`, and it's used to display various system and process settings, such as memory and hardware information, operating system (OS) version and directory information, and environment variables.

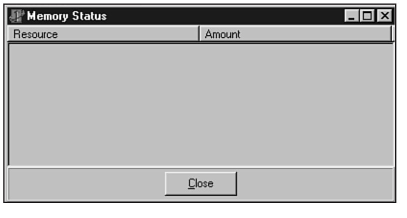

FIGURE 14.1

The `InfoForm` child form.

The contents of the form are quite simple. The form contains a `THeaderListbox` (a custom component covered in Chapter 21, "Writing Delphi Custom Components") and a `TButton`. To refresh your memory, the `THeaderListbox` control is a combination of a `THeader` control and a `TListBox` control. When the sections of the header are sized, the list box contents will also size appropriately. The `TheaderListbox` control, called `InfoLB`, displays the information mentioned earlier. The button dismisses the form.

Formatting the Strings

This application makes extensive use of the `Format()` function to format predefined strings with data retrieved from the OS at runtime. The strings that will be used are defined in a `const` section in the main unit:

```
const
  { Memory status strings }
  SMemUse   = 'Memory in useq%d%%';
  STotMem   = 'Total physical memoryq$%.8x bytes';
  SFreeMem  = 'Free physical memoryq$%.8x bytes';
  STotPage  = 'Total page file memoryq$%.8x bytes';
  SFreePage = 'Free page file memoryq$%.8x bytes';
  STotVirt  = 'Total virtual memoryq$%.8x bytes';
  SFreeVirt = 'Free virtual memoryq$%.8x bytes';
  { OS version info strings }
  SOSVer    = 'OS Versionq%d.%d';
  SBuildNo  = 'Build Numberq%d';
  SOSPlat   = 'Platformq%s';
  SOSWin32s = 'Windows 3.1x running Win32s';
  SOSWin95  = 'Windows 95/98';
  SOSWinNT  = 'Windows NT/2000';
  { System info strings }
  SProc     = 'Processor Arhitectureq%s';
  SPIntel   = 'Intel';
  SPageSize = 'Page Sizeq$%.8x bytes';
  SMinAddr  = 'Minimum Application Addressq$%p';
  SMaxAddr  = 'Maximum Application Addressq$%p';
  SNumProcs = 'Number of Processorsq%d';
  SAllocGra = 'Allocation Granularityq$%.8x bytes';
  SProcLevl = 'Processor Levelq%s';
  SIntel3   = '80386';
  SIntel4   = '80486';
  SIntel5   = 'Pentium';
  SIntel6   = 'Pentium Pro';
  SProcRev  = 'Processor Revisionq%.4x';
  { Directory strings }
  SWinDir   = 'Windows directoryq%s';
  SSysDir   = 'Windows system directoryq%s';
  SCurDir   = 'Current directoryq%s';
```

You're probably wondering about the conspicuous "q" in the middle of each of the strings. When displaying these strings, the `DelimChar` property of `InfoLB` is set to `q`, which means that the `InfoLB` component assumes that the character `q` defines the delimiter between each column in the list box.

There are three primary reasons for using `Format()` with predefined strings rather than individually formatting string literals:

- Because `Format()` accepts various types as parameters, you don't have to cloud your code with a bunch of varied calls to functions (such as `IntToStr()` and `IntToHex()`), which format different parameter types for display.
- `Format()` easily handles multiple data types. In this case, we use the `%s` and `%d` format strings to format string and numeric data so that it's more flexible.
- Keeping the strings in a separate location makes it easier to find, add, and change strings, if necessary. It's also more maintainable.

NOTE

Use a double percent sign (`%%`) to display a single percent symbol in a formatted string.

Obtaining Memory Status

The first bit of system information you can obtain to place in `InfoLB` is the memory status obtained by the `GlobalMemoryStatus()` API call. `GlobalMemoryStatus()` is a procedure that accepts one `var` parameter of type `TMemoryStatus`, which is defined as follows:

```
type
  TMemoryStatus = record
    dwLength: DWORD;
    dwMemoryLoad: DWORD;
    dwTotalPhys: DWORD;
    dwAvailPhys: DWORD;
    dwTotalPageFile: DWORD;
    dwAvailPageFile: DWORD;
    dwTotalVirtual: DWORD;
    dwAvailVirtual: DWORD;
  end;
```

- The first field in this record, `dwLength`, describes the length of the `TMemoryStatus` record. You should initialize this value to `SizeOf(TMemoryStatus)` prior to calling

GlobalMemoryStatus(). Doing this allows Windows to change the size of this record in future versions because it will be able to differentiate versions based on the value of the first field.

- dwMemoryLoad provides a number from 0 to 100 that's intended to give a general idea of memory usage. 0 means that no memory is being used, and 100 means that all memory is in use.
- dwTotalPhys indicates the total number of bytes of physical memory (the amount of RAM installed on the computer), and dwAvailPhys indicates how much of that total is currently unused.
- dwTotalPageFile indicates the total number of bytes that can be stored to hard disk page file(s). This number is not the same as the size of a page file on disk. dwAvailPageFile indicates how much of that total is available.
- dwTotalVirtual indicates the total number of bytes of usable virtual memory in the calling process. dwAvailVirtual indicates how much of this memory is available to the calling process.

The following code obtains the memory status and fills the list box with the status information:

```
procedure TInfoForm.ShowMemStatus;
var
  MS: TMemoryStatus;
begin
  InfoLB.DelimChar := 'q';
  MS.dwLength := SizeOf(MS);
  GlobalMemoryStatus(MS);
  with InfoLB.Items, MS do
  begin
    Clear;
    Add(Format(SMemUse, [dwMemoryLoad]));
    Add(Format(STotMem, [dwTotalPhys]));
    Add(Format(SFreeMem, [dwAvailPhys]));
    Add(Format(STotPage, [dwTotalPageFile]));
    Add(Format(SFreePage, [dwAvailPageFile]));
    Add(Format(STotVirt, [dwTotalVirtual]));
    Add(Format(SFreeVirt, [dwAvailVirtual]));
  end;
  InfoLB.Sections[0].Text := 'Resource';
  InfoLB.Sections[1].Text := 'Amount';
  Caption:= 'Memory Status';
end;
```

CAUTION

Don't forget to initialize the `dwLength` field of the `TMemoryStatus` structure before calling `GlobalMemoryStatus()`.

Figure 14.2 shows `InfoForm` displaying memory status information at runtime.

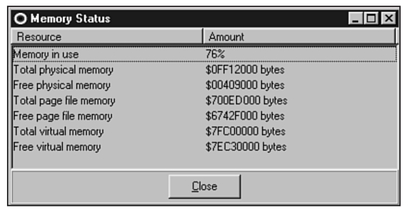

FIGURE 14.2

Viewing memory status information.

Getting the OS Version

You can find out what version of Windows and the Win32 OS you're running by making a call to the `GetVersionEx()` API function. `GetVersionEx()` accepts as its only parameter a `TOSVersionInfo` record, by reference. This record is defined as follows:

```
type
  TOSVersionInfo = record
    dwOSVersionInfoSize: DWORD;
    dwMajorVersion: DWORD;
    dwMinorVersion: DWORD;
    dwBuildNumber: DWORD;
    dwPlatformId: DWORD;
    szCSDVersion: array[0..126] of AnsiChar; {Maintenance string for PSS usage}
  end;
```

- The `dwOSVersionInfoSize` field should be initialized to `SizeOf(TOSVersionInfo)` prior to calling `GetVersionEx()`.
- `dwMajorVersion` indicates the major release number of the OS. In other words, if the OS version number is 4.0, the value of this field will be `4`.
- `dwMinorVersion` indicates the minor release number of the OS. In other words, if the OS version number is 4.0, the value of this field will be `0`.
- `dwBuildNumber` holds the build number of the OS in its low-order word.

- dwPlatformId describes the current Win32 platform. This parameter can have any one of the values in the following table:

Value	*Platform*
VER_PLATFORM_WIN32s	Win32s on Windows 3.1
VER_PLATFORM_WIN32_WINDOWS	Win32 on Windows 95 or Windows 98
VER_PLATFORM_WIN32_NT	Windows NT or Windows 2000

- szCSDVersion contains additional arbitrary OS information. This value is often an empty string.

The following procedure populates InfoLB with OS version information:

```
procedure TInfoForm.GetOSVerInfo;
var
  VI: TOSVersionInfo;
begin
  VI.dwOSVersionInfoSize := SizeOf(VI);
  GetVersionEx(VI);
  with InfoLB.Items, VI do
  begin
    Clear;
    Add(Format(SOSVer, [dwMajorVersion, dwMinorVersion]));
    Add(Format(SBuildNo, [LoWord(dwBuildNumber)]));
    case dwPlatformID of
      VER_PLATFORM_WIN32S: Add(Format(SOSPlat, [SOSWin32s]));
      VER_PLATFORM_WIN32_WINDOWS: Add(Format(SOSPlat, [SOSWin95]));
      VER_PLATFORM_WIN32_NT: Add(Format(SOSPlat, [SOSWinNT]));
    end;
  end;
end;
```


NOTE

In Windows 3.*x*, the GetVersion() function obtained similar version information. Because you're now in Win32 land, you should use the GetVersionEx() function; it provides more detailed information than GetVersion().

Obtaining Directory Information

The OS uses the Windows and System directories extensively to store shared DLLs, drivers, applications, and INI files. Additionally, Win32 also maintains a current directory for each process. Throughout the course of writing Win32 applications, it's likely that you'll encounter a situation where you need to obtain the location of one of these directories. When this happens, you'll be in luck because three functions in the Win32 API enable you to obtain that directory information.

The functions—`GetWindowsDirectory()`, `GetSystemDirectory()`, and `GetCurrentDirectory()`—are pretty straightforward. Each takes a pointer to a buffer where the directory string is copied as the first parameter and the buffer size is copied as the second parameter. The function copies into the buffer a null-terminated string containing the path. Hopefully, you can tell which directory each function returns by the name of the function. If not, well, let's just say we hope you don't rely on programming to eat.

This method uses a temporary array of `char` into which the directory information is stored. From there, the string is added to `InfoLB`, as you can see for yourself in the following code:

```
procedure TInfoForm.GetDirInfo;
var
  S: array[0..MAX_PATH] of char;
begin
  { Get Windows directory }
  GetWindowsDirectory(S, SizeOf(S));
  InfoLB.Items.Add(Format(SWinDir, [S]));
  { Get Windows system directory }
  GetSystemDirectory(S, SizeOf(S));
  InfoLB.Items.Add(Format(SSysDir, [S]));
  { Get Current directory for current process }
  GetCurrentDirectory(SizeOf(S), S);
  InfoLB.Items.Add(Format(SCurDir, [S]));
end;
```

NOTE

The `GetWindowsDir()` and `GetSystemDir()` functions from the Windows 3.*x* API are unavailable under Win32.

Getting System Information

The Win32 API provides a procedure called `GetSystemInfo()` that, in turn, provides some very low-level details on the operating system. This procedure accepts one parameter of type `TSystemInfo` by reference, and it fills the record with the proper values. The `TSystemInfo` record is defined as follows:

```
type
  PSystemInfo = ^TSystemInfo;
```

```
TSystemInfo = record
  case Integer of
    0: (
      dwOemId: DWORD);
    1: (
      wProcessorArchitecture: Word;
      wReserved: Word;
      dwPageSize: DWORD;
      lpMinimumApplicationAddress: Pointer;
      lpMaximumApplicationAddress: Pointer;
      dwActiveProcessorMask: DWORD;
      dwNumberOfProcessors: DWORD;
      dwProcessorType: DWORD;
      dwAllocationGranularity: DWORD;
      wProcessorLevel: Word;
      wProcessorRevision: Word);
end;
```

- The `dwOemId` field is used for Windows 95. This value is always set to `0` or `PROCESSOR_ARCHITECTURE_INTEL`.
- Under NT, the `wProcessorArchitecture` portion of the variant record is used. This field describes the type of processor architecture under which you're currently running. Because Delphi is designed for Intel only, however, it's the only type that matters at this point. For the sake of completeness, this field can have any one of the following values:

  `PROCESSOR_ARCHITECTURE_INTEL`

  `PROCESSOR_ARCHITECTURE_MIPS`

  `PROCESSOR_ARCHITECTURE_ALPHA`

  `PROCESSOR_ARCHITECTURE_PPC`
- The `wReserved` field is unused at this time.
- The `dwPageSize` field holds the page size in kilobytes (KB) and specifies the granularity of page protection and commitment. On Intel *x*86 machines, this value is 4KB.
- `lpMinimumApplicationAddress` returns the lowest memory address accessible to applications and DLLs. Attempts to access a memory address below this value is likely to result in an access violation. `lpMaximumApplicationAddress` returns the highest memory address accessible to applications and DLLs. Attempts to access a memory address above this value are likely to result in an access violation.
- `dwActiveProcessorMask` returns a mask representing the set of processors configured into the system. Bit 0 represents the first processor, and bit 31 represents the 32nd processor. Wouldn't having 32 processors be cool? Because Windows 95/98 supports only one processor, only bit 0 will be set under that implementation of Win32.

- `dwNumberOfProcessors` also returns the number of processors in the system. We're not sure why Microsoft bothered to put both this and the preceding field in the `TSystemInfo` record, but here they are.
- The `dwProcessorType` field is no longer relevant. It was retained for backward compatibility. This field can have any one of the following values:

  `PROCESSOR_INTEL_386`

  `PROCESSOR_INTEL_486`

  `PROCESSOR_INTEL_PENTIUM`

  `PROCESSOR_MIPS_R4000`

  `PROCESSOR_ALPHA_21064`

  Of course, under Windows 95/98, only the `PROCESSOR_INTEL_`*`x`* values are possible, whereas all are valid under Windows NT.
- `dwAllocationGranularity` returns the allocation granularity upon which memory will be allocated. In previous implementations of Win32, this value was hard-coded as 64KB. It's possible, however, that other hardware architectures may require different values.
- The `wProcessorLevel` field specifies the system's architecture-dependent processor level. This field can hold a variety of values for different processors. For Intel processors, this parameter can have any of the values in the following table:

Value	*Meaning*
3	Processor is an 80386
4	Processor is an 80486
5	Processor is a Pentium
6	Processor is a Pentium Pro or higher

- `wProcessorRevision` specifies an architecture-dependent processor revision. Like `wProcessorLevel`, this field can hold a variety of values for different processors. For Intel architectures, this field holds a number in the format *`xxyy`*. For Intel 386 and 486 chips, *`xx`* `+ $0A` is the stepping level and *`yy`* is the stepping (for example, `0300` is a D0 chip). For Intel Pentium or Cyrex/NextGen 486 chips, *`xx`* is the model number, and *`yy`* is the stepping (for example, `0201` is Model 2, Stepping 1).

The procedure used to obtain and add the formatted system information strings to `InfoLB` is as follows (note that this code is purposely slanted to display only Intel architecture information):

```
procedure TInfoForm.GetSysInfo;
var
  SI: TSystemInfo;
begin
  GetSystemInfo(SI);
```

```
  with InfoLB.Items, SI do
  begin
    Add(Format(SProc, [SPIntel]));
    Add(Format(SPageSize, [dwPageSize]));
    Add(Format(SMinAddr, [lpMinimumApplicationAddress]));
    Add(Format(SMaxAddr, [lpMaximumApplicationAddress]));
    Add(Format(SNumProcs, [dwNumberOfProcessors]));
    Add(Format(SAllocGra, [dwAllocationGranularity]));
    case wProcessorLevel of
      3: Add(Format(SProcLevl, [SIntel3]));
      4: Add(Format(SProcLevl, [SIntel4]));
      5: Add(Format(SProcLevl, [SIntel5]));
      6: Add(Format(SProcLevl, [SIntel6]));
    else Add(Format(SProcLevl, [IntToStr(wProcessorLevel)]));
    end;
  end;
end;
```

NOTE

The `GetSystemInfo()` function effectively replaces the `GetWinFlags()` function from the Windows 3.*x* API.

Figure 14.3 shows `InfoForm` displaying system information, including OS version and directory information, at runtime.

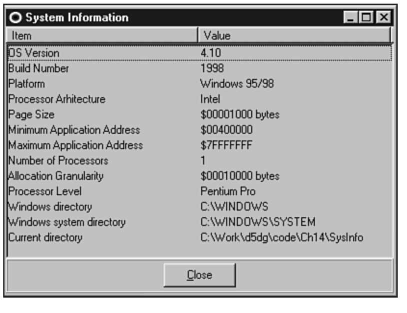

FIGURE 14.3

Viewing system information.

Checking Out the Environment

Obtaining the list of environment variables—things such as sets, path, and prompt—for the current process is an easy task, thanks to the `GetEnvironmentStrings()` API function. This function takes no parameters and returns a null-separated list of environment strings. The format of this list is a string, followed by a null, followed by a string, followed by a null, and so on until the entire string is terminated with a double null (`#0#0`). The following function is used in the `SysInfo` application to retrieve the output from the `GetEnvironmentStrings()` function and place it into `InfoLB`:

```
procedure TInfoForm.ShowEnvironment;
var
  EnvPtr, SavePtr: PChar;
begin
  InfoLB.DelimChar := '=';
  EnvPtr := GetEnvironmentStrings;
  SavePtr := EnvPtr;
  InfoLB.Items.Clear;
  repeat
    InfoLB.Items.Add(StrPas(EnvPtr));
    inc(EnvPtr, StrLen(EnvPtr) + 1);
  until EnvPtr^ = #0;
  FreeEnvironmentStrings(SavePtr);
  InfoLB.Sections[0].Text := 'Environment Variable';
  InfoLB.Sections[1].Text := 'Value';
  Caption:= 'Current Environment';
end;
```

NOTE

The `ShowEnvironment()` method takes advantage of Object Pascal's capability to perform pointer arithmetic on `PChar`-type strings. Notice how few lines of code are required to traverse the list of environment strings.

A couple of comments on this method are in order. First, notice that the `DelimChar` property of `InfoLB` is initially set to `'='`. Because each of the environment variable and value pairs are already separated by that character, it's very easy to display them properly in `InfoLB`. Also, when you're finished using the environment strings, you should call the `FreeEnvironmentStrings()` function to free the allocated block.

TIP

You can't obtain or set individual environment variables with the `GetEnvironmentStrings()` function. For getting and setting individual environment variables, see the `GetEnvironmentVariable()` and `SetEnvironmentVariable()` functions in the Win32 API help.

Figure 14.4 shows the `InfoForm` environment strings at runtime.

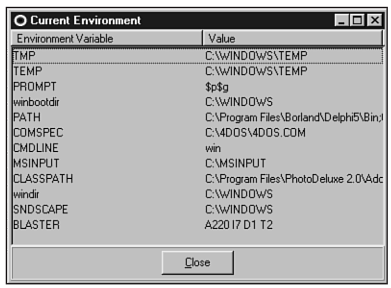

FIGURE 14.4

Viewing environment strings.

Listing 14.1 shows the entire source code for the `InfoU.pas` unit.

LISTING 14.1 The Source Code for the `InfoU.pas` Unit

```
unit InfoU;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  HeadList, StdCtrls, ExtCtrls, SysMain;

type
  TInfoVariety = (ivMemory, ivSystem, ivEnvironment);

  TInfoForm = class(TForm)
    InfoLB: THeaderListbox;
    Panel1: TPanel;
```


continues

LISTING 14.1 Continued

```
    OkBtn: TButton;
  private
    procedure GetOSVerInfo;
    procedure GetSysInfo;
    procedure GetDirInfo;
  public
    procedure ShowMemStatus;
    procedure ShowSysInfo;
    procedure ShowEnvironment;
  end;

procedure ShowInformation(Variety: TInfoVariety);

implementation

{$R *.DFM}

procedure ShowInformation(Variety: TInfoVariety);
begin
  with TInfoForm.Create(Application) do
    try
      Font := MainForm.Font;
      case Variety of
        ivMemory: ShowMemStatus;
        ivSystem: ShowSysInfo;
        ivEnvironment: ShowEnvironment;
      end;
      ShowModal;
    finally
      Free;
    end;
end;

const
  { Memory status strings }
  SMemUse   = 'Memory in useq%d%%';
  STotMem   = 'Total physical memoryq$%.8x bytes';
  SFreeMem  = 'Free physical memoryq$%.8x bytes';
  STotPage  = 'Total page file memoryq$%.8x bytes';
  SFreePage = 'Free page file memoryq$%.8x bytes';
  STotVirt  = 'Total virtual memoryq$%.8x bytes';
```

```
  SFreeVirt = 'Free virtual memoryq$%.8x bytes';

  { OS version info strings }
  SOSVer    = 'OS Versionq%d.%d';
  SBuildNo  = 'Build Numberq%d';
  SOSPlat   = 'Platformq%s';
  SOSWin32s = 'Windows 3.1x running Win32s';
  SOSWin95  = 'Windows 95/98';
  SOSWinNT  = 'Windows NT/2000';

  { System info strings }
  SProc     = 'Processor Arhitectureq%s';
  SPIntel   = 'Intel';
  SPageSize = 'Page Sizeq$%.8x bytes';
  SMinAddr  = 'Minimum Application Addressq$%p';
  SMaxAddr  = 'Maximum Application Addressq$%p';
  SNumProcs = 'Number of Processorsq%d';
  SAllocGra = 'Allocation Granularityq$%.8x bytes';
  SProcLevl = 'Processor Levelq%s';
  SIntel3   = '80386';
  SIntel4   = '80486';
  SIntel5   = 'Pentium';
  SIntel6   = 'Pentium Pro';
  SProcRev  = 'Processor Revisionq%.4x';

  { Directory strings }
  SWinDir   = 'Windows directoryq%s';
  SSysDir   = 'Windows system directoryq%s';
  SCurDir   = 'Current directoryq%s';

procedure TInfoForm.ShowMemStatus;
var
  MS: TMemoryStatus;
begin
  InfoLB.DelimChar := 'q';
  MS.dwLength := SizeOf(MS);
  GlobalMemoryStatus(MS);
  with InfoLB.Items, MS do
  begin
    Clear;
    Add(Format(SMemUse, [dwMemoryLoad]));
    Add(Format(STotMem, [dwTotalPhys]));
```


continues

LISTING 14.1 Continued

```
    Add(Format(SFreeMem, [dwAvailPhys]));
    Add(Format(STotPage, [dwTotalPageFile]));
    Add(Format(SFreePage, [dwAvailPageFile]));
    Add(Format(STotVirt, [dwTotalVirtual]));
    Add(Format(SFreeVirt, [dwAvailVirtual]));
  end;
  InfoLB.Sections[0].Text := 'Resource';
  InfoLB.Sections[1].Text := 'Amount';
  Caption:= 'Memory Status';
end;

procedure TInfoForm.GetOSVerInfo;
var
  VI: TOSVersionInfo;
begin
  VI.dwOSVersionInfoSize := SizeOf(VI);
  GetVersionEx(VI);
  with InfoLB.Items, VI do
  begin
    Clear;
    Add(Format(SOSVer, [dwMajorVersion, dwMinorVersion]));
    Add(Format(SBuildNo, [LoWord(dwBuildNumber)]));
    case dwPlatformID of
      VER_PLATFORM_WIN32S: Add(Format(SOSPlat, [SOSWin32s]));
      VER_PLATFORM_WIN32_WINDOWS: Add(Format(SOSPlat, [SOSWin95]));
      VER_PLATFORM_WIN32_NT: Add(Format(SOSPlat, [SOSWinNT]));
    end;
  end;
end;

procedure TInfoForm.GetSysInfo;
var
  SI: TSystemInfo;
begin
  GetSystemInfo(SI);
  with InfoLB.Items, SI do
  begin
    Add(Format(SProc, [SPIntel]));
    Add(Format(SPageSize, [dwPageSize]));
    Add(Format(SMinAddr, [lpMinimumApplicationAddress]));
```

```
    Add(Format(SMaxAddr, [lpMaximumApplicationAddress]));
    Add(Format(SNumProcs, [dwNumberOfProcessors]));
    Add(Format(SAllocGra, [dwAllocationGranularity]));
    case wProcessorLevel of
      3: Add(Format(SProcLevl, [SIntel3]));
      4: Add(Format(SProcLevl, [SIntel4]));
      5: Add(Format(SProcLevl, [SIntel5]));
      6: Add(Format(SProcLevl, [SIntel6]));
    else Add(Format(SProcLevl, [IntToStr(wProcessorLevel)]));
    end;
  end;
end;

procedure TInfoForm.GetDirInfo;
var
  S: array[0..MAX_PATH] of char;
begin
  { Get Windows directory }
  GetWindowsDirectory(S, SizeOf(S));
  InfoLB.Items.Add(Format(SWinDir, [S]));
  { Get Windows system directory }
  GetSystemDirectory(S, SizeOf(S));
  InfoLB.Items.Add(Format(SSysDir, [S]));
  { Get Current directory for current process }
  GetCurrentDirectory(SizeOf(S), S);
  InfoLB.Items.Add(Format(SCurDir, [S]));
end;

procedure TInfoForm.ShowSysInfo;
begin
  InfoLB.DelimChar := 'q';
  GetOSVerInfo;
  GetSysInfo;
  GetDirInfo;
  InfoLB.Sections[0].Text := 'Item';
  InfoLB.Sections[1].Text := 'Value';
  Caption:= 'System Information';
end;

procedure TInfoForm.ShowEnvironment;
var
  EnvPtr, SavePtr: PChar;
```


continues

LISTING 14.1 Continued

```
begin
  InfoLB.DelimChar := '=';
  EnvPtr := GetEnvironmentStrings;
  SavePtr := EnvPtr;
  InfoLB.Items.Clear;
  repeat
    InfoLB.Items.Add(StrPas(EnvPtr));
    inc(EnvPtr, StrLen(EnvPtr) + 1);
  until EnvPtr^ = #0;
  FreeEnvironmentStrings(SavePtr);
  InfoLB.Sections[0].Text := 'Environment Variable';
  InfoLB.Sections[1].Text := 'Value';
  Caption:= 'Current Environment';
end;

end.
```

Platform-Neutral Design

SysInfo is designed to function under both Windows 95/98 and Windows NT, even though the different versions of Win32 have very different ways of accessing low-level information such as processes and memory. The approach we took to enable platform-neutrality is to define an interface that contains methods that can obtain system information. This interface is then implemented for the two different operating systems. The interface is called `IWin32Info`; it's pretty simple and is shown here:

```
type
  IWin32Info = interface
    procedure FillProcessInfoList(ListView: TListView; ImageList: TImageList);
    procedure ShowProcessProperties(Cookie: Pointer);
  end;
```

- `FillProcessInfoList()` is responsible for filling a `TListView` and `TImageList` component with a list of running processes and their associated icons, if any.
- `ShowProcessProperties()` is called to obtain more information for a particular process selected in `TListView`.

In the SysInfo project, you'll find a unit called `W95Info` that contains a `TWin95Info` class that implements `IWin32Info` for Windows 95/98 using the ToolHelp32 API. Likewise, the project contains a `WNTInfo` unit with a `TWinNTInfo` class that takes advantage of PSAPI to implement `IWin32Info`. The following code segment, `SysMain` (which was taken from the project's main unit), shows how the proper class is created depending on the operating system:

```
if Win32Platform = VER_PLATFORM_WIN32_WINDOWS then
  FWinInfo := TWin95Info.Create
else if Win32Platform = VER_PLATFORM_WIN32_NT then
  FWinInfo := TWinNTInfo.Create
else
  raise Exception.Create('This application must be run on Win32');
```

Windows 95/98: Using ToolHelp32

ToolHelp32 is a collection of functions and procedures, part of the Win32 API, which enables you to see the status of some of the operating system's low-level operations. In particular, functions enable you to obtain information on all processes currently executing in the system and the threads, modules, and heaps that go with each of the processes. As you might guess, most of the information obtainable from ToolHelp32 is primarily used by applications that must look "inside" the OS, such as debuggers, although going through these functions gives even the average developer a better idea of how Win32 is put together.

NOTE

The ToolHelp32 API is available only under the Windows 95/98 implementation of Win32. This type of functionality would violate NT's robust process-protection and security features. Therefore, applications that use ToolHelp32 functions will function only under Windows 95/98 and not under Windows NT.

We say *ToolHelp32* to differentiate it from the 16-bit version of ToolHelp that was included in Windows 3.1*x*. Most of the functions in the previous version of ToolHelp no longer apply to Win32 and are therefore no longer supported. Also, under Windows 3.1*x*, the ToolHelp functions were physically located in a DLL called `TOOLHELP.DLL`, whereas ToolHelp32 functions reside in the kernel under Win32.

ToolHelp32 types and function definitions are located in the `TlHelp32` unit, so be sure to have that in your `uses` clause when working with these functions. To ensure that you receive a solid overview, the application you build in this chapter uses every function defined in the `TlHelp32` unit.

Figure 14.5 shows the main form for SysInfo. The user interface consists primarily of `TheaderListbox`, a custom control explained in detail in Chapter 11, "Writing Multithreaded Applications." The list contains important information for a given process. By double-clicking a process in the list, you can obtain more detailed information about it. This detail is shown in a child form similar to the main form.

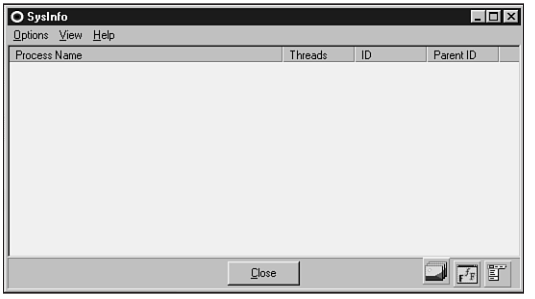

FIGURE 14.5

SysInfo's main form, `TMainForm`.

Snapshots

Due to the multitasking nature of the Win32 environment, objects such as processes, threads, modules, and heaps are constantly being created, destroyed, and modified. Because the status of the machine is constantly in a state of flux, system information that might be meaningful now may have no meaning a second from now. For example, suppose you want to write a program to enumerate through all the modules loaded systemwide. Because the operating system might preempt the thread executing your program at any time in order to provide time slices to other threads in the system, modules theoretically can be created and destroyed even as you enumerate through them.

In this dynamic environment, it would make more sense if you could freeze the system in time for a moment in order to obtain such system information. Although ToolHelp32 doesn't provide a means for freezing the system in time, it does provide a function that enables you to take a snapshot of the system at a particular moment. `CreateToolhelp32Snapshot()` is that function and is declared as follows:

```
function CreateToolhelp32Snapshot(dwFlags, th32ProcessID: DWORD): THandle;
  stdcall;
```

- The `dwFlags` parameter indicates what type of information should be included in the snapshot. This parameter can have any one of the values shown in the following table:

Value	*Meaning*
`TH32CS_INHERIT`	Indicates that the snapshot handle will be inheritable
`TH32CS_SNAPALL`	Equivalent to specifying the `TH32CS_SNAPHEAPLIST`, `TH32CS_SNAPMODULE`, `TH32CS_SNAPPROCESS`, and `TH32CS_SNAPTHREAD` values
`TH32CS_SNAPHEAPLIST`	Includes the heap list of the specified Win32 process in the snapshot

TH32CS_SNAPMODULE	Includes the module list of the specified Win32 process in the snapshot
TH32CS_SNAPPROCESS	Includes the Win32 process list in the snapshot
TH32CS_SNAPTHREAD	Includes the Win32 thread list in the snapshot

- The th32ProcessID parameter identifies the process for which you want to obtain information. Pass 0 in this parameter to indicate the current process. This parameter affects only module and heap lists because they are process-specific. The process and thread lists provided by ToolHelp32 are systemwide.
- The CreateToolhelp32Snapshot() function returns the handle to a snapshot or -1 in case of an error. The handle returned works just as other Win32 handles do regarding the processes and threads for which they're valid.

The following code creates a snapshot handle that contains information on all processes currently loaded systemwide (EToolHelpError is a programmer-defined exception):

```
var
  Snap: THandle;
begin
  Snap := CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
  if Snap = -1 then
    raise EToolHelpError.Create('CreateToolHelp32Snapshot failed');
end;
```

NOTE

When you're done using the handle, use the Win32 API CloseHandle() function to free resources associated with a handle created by CreateToolHelp32Snapshot().

Process Walking

Given a snapshot handle that includes process information, ToolHelp32 defines two functions that provide you with the capability of enumerating over (*walking*) processes. The functions, Process32First() and Process32Next(), are declared as follows:

```
function Process32First(hSnapshot: THandle;
    var lppe: TProcessEntry32): BOOL; stdcall;
function Process32Next(hSnapshot: THandle;
    var lppe: TProcessEntry32): BOOL; stdcall;
```

The first parameter to these functions, hSnapshot, is the snapshot handle returned by CreateToolHelp32Snapshot().

The second parameter, `lppe`, is a `TProcessEntry32` record that's passed by reference. As you go through the enumeration, the functions will fill this record with information on the next process. The `TProcessEntry32` record is defined as follows:

```
type
  TProcessEntry32 = record
    dwSize: DWORD;
    cntUsage: DWORD;
    th32ProcessID: DWORD;
    th32DefaultHeapID: DWORD;
    th32ModuleID: DWORD;
    cntThreads: DWORD;
    th32ParentProcessID: DWORD;
    pcPriClassBase: Longint;
    dwFlags: DWORD;
    szExeFile: array[0..MAX_PATH - 1] of Char;
  end;
```

- The `dwSize` field holds the size of the `TProcessEntry32` record. This should be initialized to `SizeOf(TProcessEntry32)` prior to using the record.
- The `cntUsage` field indicates the reference count of the process. When the reference count is zero, the operating system will unload the process.
- The `th32ProcessID` field contains the identification number of the process.
- The `th32DefaultHeapID` field contains an identifier for the process's default heap. The ID has meaning only within ToolHelp32, and it can't be used with other Win32 functions.
- The `thModuleID` field identifies the module associated with the process. This field has meaning only within ToolHelp32 functions.
- The `cntThreads` field indicates how many threads of execution the process has started.
- The `th32ParentProcessID` identifies the parent process to this process.
- The `pcPriClassBase` field holds the base priority of the process. The operating system uses this value to manage thread scheduling.
- The `dwFlags` field is reserved; don't use it.
- The `szExeFile` field is a null-terminated string that contains the pathname and filename of the EXE or driver associated with the process.

Once a snapshot containing process information has been taken, iterating over all processes is a matter of calling `Process32First()` and then calling `Process32Next()` until it returns `False`.

The process-walking code is encapsulated in the `TWin95Info` class, which implements the `IWin32Info` interface. The following code shows the private `Refresh()` method of the `TWin95Info` class, which iterates over the system processes and adds each to a list:

```
procedure TWin95Info.Refresh;
var
  PE: TProcessEntry32;
  PPE: PProcessEntry32;
begin
  FProcList.Clear;
  if FSnap > 0 then CloseHandle(FSnap);
  FSnap := CreateToolHelp32Snapshot(TH32CS_SNAPPROCESS, 0);
  if FSnap = -1 then
    raise Exception.Create('CreateToolHelp32Snapshot failed');
  PE.dwSize := SizeOf(PE);
  if Process32First(FSnap, PE) then               // get process
    repeat
      New(PPE);                                  // create new PPE
      PPE^ := PE;                                // fill it
      FProcList.Add(PPE);                        // add it to list
    until not Process32Next(FSnap, PE);           // get next process
end;
```

The `Refresh()` method is called by the `FillProcessInfoList()` method. As explained earlier, this method fills a `TListView` and `TImageList` component with information on all the running processes. It's shown here:

```
procedure TWin95Info.FillProcessInfoList(ListView: TListView;
  ImageList: TImageList);
var
  I: Integer;
  ExeFile: string;
  PE: TProcessEntry32;
  HAppIcon: HIcon;
begin
  Refresh;
  ListView.Columns.Clear;
  ListView.Items.Clear;
  for I := Low(ProcessInfoCaptions) to High(ProcessInfoCaptions) do
    with ListView.Columns.Add do
    begin
      if I = 0 then Width := 285
      else Width := 75;
      Caption := ProcessInfoCaptions[I];
    end;
  for I := 0 to FProcList.Count - 1 do
  begin
```

```
    PE := PProcessEntry32(FProcList.Items[I])^;
    HAppIcon := ExtractIcon(HInstance, PE.szExeFile, 0);
    try
      if HAppIcon = 0 then HAppIcon := FWinIcon;
      ExeFile := PE.szExeFile;
      if ListView.ViewStyle = vsList then
        ExeFile := ExtractFileName(ExeFile);
      // insert new item, set its caption, add subitems
      with ListView.Items.Add, SubItems do
      begin
        Caption := ExeFile;
        Data := FProcList.Items[I];
        Add(IntToStr(PE.cntThreads));
        Add(IntToHex(PE.th32ProcessID, 8));
        Add(IntToHex(PE.th32ParentProcessID, 8));
        if ImageList <> nil then
          ImageIndex := ImageList_AddIcon(ImageList.Handle, HAppIcon);
      end;
    finally
      if HAppIcon <> FWinIcon then DestroyIcon(HAppIcon);
    end;
  end;
end;
```

Figure 14.6 shows this code in action, displaying process information on a Windows 98 machine.

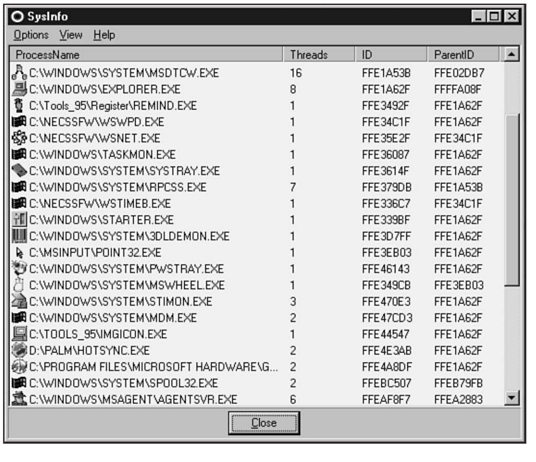

FIGURE 14.6

Viewing processes under Windows 98.

Not to be ignored is the code that obtains an icon for each process. Displaying the icon along with the application name gives the application a more professional appearance and a more native Windows feel. The `ExtractIcon()` API function from the `ShellAPI` unit attempts to extract the icon from the application file. If `ExtractIcon()` fails, `HWinIcon` is used instead. `HWinIcon` is the standard Windows icon, and it has been preloaded in the `OnCreate` event handler for this form using the `LoadImage()` API function:

```
FWinIcon := LoadImage(0, IDI_WINLOGO, IMAGE_ICON, LR_DEFAULTSIZE,
  LR_DEFAULTSIZE, LR_DEFAULTSIZE or LR_DEFAULTCOLOR or LR_SHARED);
```

When the user double-clicks one of the processes in the main form (refer to Figure 14.6), the `ShowProcessProperties()` method of `IWin32Info` is called, and the implementation of this method passes the parameter on to a method in the `Detail9x` unit called `ShowProcessDetails()`:

```
procedure TWin95Info.ShowProcessProperties(Cookie: Pointer);
begin
  ShowProcessDetails(PProcessEntry32(Cookie));
end;
```

`ShowProcessDetails()` must take another snapshot with `CreateToolHelp32Snapshot()` in order to obtain a snapshot of information for the selected process. This is done by passing the `Cookie` parameter, which holds the process (ID in this case) to the chosen process as the `th32ProcessID` field for `CreateToolHelp32Snapshot()`. The `TH32CS_SNAPALL` flag is passed as the `dwFlags` parameter to put all the information into the snapshot, as shown in the following snippet:

```
{ Create a snapshot for the current process }
FCurSnap := CreateToolhelp32Snapshot(TH32CS_SNAPALL, P^.th32ProcessID);
if FCurSnap = -1 then
  raise EToolHelpError.Create('CreateToolHelp32Snapshot failed');
```


The `TDetailForm` object displays only one list at a time. An enumerated type keeps track of which list is which:

```
type
  TListType = (ltThread, ltModule, ltHeap);
```

`TDetailForm` also maintains three separate `TStringList` components for each of the threads, modules, and heaps. These lists are defined as part of an array called `DetailLists`:

```
DetailLists: array[TListType] of TStringList;
```

Thread Walking

To walk a process's thread list, ToolHelp32 provides two functions similar to those for process walking: `Thread32First()` and `Thread32Next()`. These functions are declared as follows:

```
function Thread32First(hSnapshot: THandle;
  var lpte: TThreadEntry32): BOOL; stdcall;

function Thread32Next(hSnapshot: THandle;
  var lpte: TThreadENtry32): BOOL; stdcall;
```

In addition to the usual `hSnapshot` parameter, these functions also accept a parameter by reference of type `TThreadEntry32`. As for the process functions, the calling function fills in this record. The `TThreadEntry32` record is defined as follows:

```
type
  TThreadEntry32 = record
    dwSize: DWORD;
    cntUsage: DWORD;
    th32ThreadID: DWORD;
    th32OwnerProcessID: DWORD;
    tpBasePri: Longint;
    tpDeltaPri: Longint;
    dwFlags: DWORD;
  end;
```

- `dwSize` is the size of the record, and it should be initialized to `SizeOf(TThreadEntry32)` prior to using the record.
- `cntUsage` is the reference count of the thread. When this value reaches zero, the thread is unloaded by the operating system.
- `th32ThreadID` is the identification number of the thread. This value has meaning only within the ToolHelp32 functions.
- `th32OwnerProcessID` is the identifier of the process that owns this thread. This ID can be used with other Win32 functions.
- `tpBasePri` is the base priority class of the thread. This value is the same for all threads of a given process. The possible values for this field are usually in the range of `4` through `24`. The following table lists the meaning of each value:

Value	*Meaning*
`4`	Idle
`8`	Normal
`13`	High
`24`	Real time

- `tpDeltaPri` is the *delta* (change in) *priority* from `tpBasePri`. It's a signed number that, when combined with the base priority class, reveals the overall priority of the thread. The following table shows the constants defined for each possible value:

Constant	*Value*
THREAD_PRIORITY_IDLE	-15
THREAD_PRIORITY_LOWEST	-2
THREAD_PRIORITY_BELOW_NORMAL	-1
THREAD_PRIORITY_NORMAL	0
THREAD_PRIORITY_ABOVE_NORMAL	1
THREAD_PRIORITY_HIGHEST	2
THREAD_PRIORITY_TIME_CRITICAL	15

- `dwFlags` is currently reserved and shouldn't be used.

The `WalkThreads()` method of `TDetailForm` is used to walk the thread list. As the thread list is traversed, important information about the thread is added to the thread element of the `DetailLists` array. Here's the code for this method:

```
procedure TWin95DetailForm.WalkThreads;
{ Uses ToolHelp32 functions to walk list of threads }
var
  T: TThreadEntry32;
begin
  DetailLists[ltThread].Clear;
  T.dwSize := SizeOf(T);
  if Thread32First(FCurSnap, T) then
    repeat
      { Make sure thread is for current process }
      if T.th32OwnerProcessID = FCurProc.th32ProcessID then
        DetailLists[ltThread].Add(Format(SThreadStr, [T.th32ThreadID,
          GetClassPriorityString(T.tpBasePri),
          GetThreadPriorityString(T.tpDeltaPri), T.cntUsage]));
    until not Thread32Next(FCurSnap, T);
end;
```

NOTE

The following line of code in the `WalkThreads()` method is important because ToolHelp32 thread lists are not process-specific:

```
if T.th32OwnerProcessID = FCurProc.th32ProcessID then
```

You must therefore do a manual comparison as you iterate through the threads to determine which threads are associated with the process in question.

Figure 14.7 shows the detail form with the thread list visible.

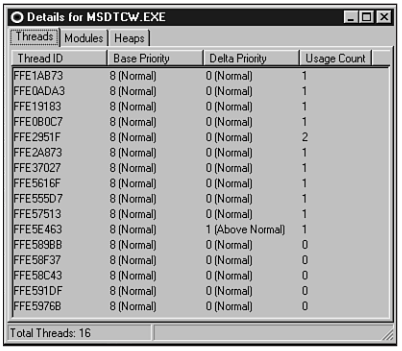

FIGURE 14.7

Viewing threads in the detail form under Windows 98.

Module Walking

Module walking works much the same as process and thread walking. ToolHelp32 provides two functions that do the work: `Module32First()` and `Module32Next()`. These functions are declared as follows:

```
function Module32First(hSnapshot: THandle;
  var lpme: TModuleEntry32): BOOL; stdcall;

function Module32Next(hSnapshot: THandle;
  var lpme: TModuleEntry32): BOOL; stdcall;
```

Again, the snapshot handle is the first parameter to the functions. The second `var` parameter, `lpme`, is a `TModuleEntry32` record. This record is defined as follows:

```
type
  TModuleEntry32 = record
    dwSize: DWORD;
    th32ModuleID: DWORD;
    th32ProcessID: DWORD;
    GlblcntUsage: DWORD;
    ProccntUsage: DWORD;
    modBaseAddr: PBYTE;
    modBaseSize: DWORD;
    hModule: HMODULE;
```

```
    szModule: array[0..MAX_MODULE_NAME32 + 1] of Char;
    szExePath: array[0..MAX_PATH - 1] of Char;
  end;
```

- `dwSize` is the size of the record, and it should be initialized to `SizeOf(TModuleEntry32)` prior to using the record.
- `th32ModuleID` is the identifier of the module. This value has meaning only with ToolHelp32 functions.
- `th32ProcessID` is the identifier of the process being examined. This value can be used with other Win32 functions.
- `GlblcntUsage` is the global reference count of the module.
- `ProccntUsage` is the reference count of the module within the context of the owning process.
- `modBaseAddr` is the base address of the module in memory. This value is valid only within the context of `th32ProcessID`'s context.
- `modBaseSize` is the size in bytes of the module in memory.
- `hModule` is the module handle. This value is valid only within `th32ProcessID`'s context.
- `szModule` is a null-terminated string containing the module name.
- `szExepath` is a null-terminated string containing the full path of the module.

The `WalkModules()` method of `TDetailForm` is very similar to its `WalkThreads()` method. As shown in the following code, this method traverses the module list and adds it to the module list portion of the `DetailLists` array:

```
procedure TWin95DetailForm.WalkModules;
{ Uses ToolHelp32 functions to walk list of modules }
var
  M: TModuleEntry32;
begin
  DetailLists[ltModule].Clear;
  M.dwSize := SizeOf(M);
  if Module32First(FCurSnap, M) then
    repeat
      DetailLists[ltModule].Add(Format(SModuleStr, [M.szModule, M.ModBaseAddr,
        M.ModBaseSize, M.ProcCntUsage]));
    until not Module32Next(FCurSnap, M);
end;
```

Figure 14.8 shows the detail form with the module list visible.

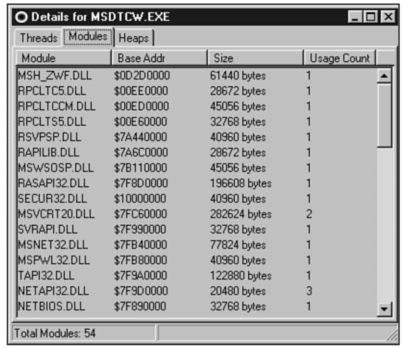

FIGURE 14.8

Viewing modules in the detail form under Windows 98.

Heap Walking

Heap walking is slightly more complicated than the other types of enumeration you've learned about in this chapter. ToolHelp32 provides four functions that enable heap walking. The first two functions, `Heap32ListFirst()` and `Heap32ListNext()`, enable you to iterate over each of a process's heaps. The other two functions, `Heap32First()` and `Heap32Next()`, enable you to obtain more detailed information on all the blocks within an individual heap.

`Heap32ListFirst()` and `Heap32ListNext()` are defined as follows:

```
function Heap32ListFirst(hSnapshot: THandle;
  var lphl: THeapList32): BOOL; stdcall;

function Heap32ListNext(hSnapshot: THandle;
  var lphl: THeapList32): BOOL; stdcall;
```

Again, the first parameter is the customary snapshot handle. The second parameter, `lphl`, is a `THeapList32` record that's passed by reference. This record is defined as follows:

```
type
  THeapList32 = record
    dwSize: DWORD;
    th32ProcessID: DWORD;
    th32HeapID: DWORD;
    dwFlags: DWORD;
  end;
```

- `dwSize` is the size of the record, and it should be initialized to `SizeOf(THeapList32)` prior to using the record.

- `th32ProcessID` is the identifier of the owning process.
- `th32HeapID` is the identifier of the heap. This value has meaning only for the specified process and within ToolHelp32.
- `dwFlags` holds a flag that determines the heap type. The value of this field can be either `HF32_DEFAULT`, which means that the current heap is the process's default heap, or `HF32_SHARED`, which means that the current heap is a normal shared heap.

The `Heap32First()` and `Heap32Next()` functions are defined as follows:

```
function Heap32First(var lphe: THeapEntry32; th32ProcessID,
  th32HeapID: DWORD): BOOL; stdcall;

function Heap32Next(var lphe: THeapEntry32): BOOL; stdcall;
```

Notice that the parameter lists of these functions are a bit of a departure from the process, thread, module, and heap list enumeration functions that you've learned about in this chapter. These functions are designed to enumerate the blocks of a given heap in a given process rather than enumerating over some properties of just a process. When calling `Heap32First()`, the `th32ProcessID` and `th32HeapID` parameters should be set to the values of the field of the same name of the `THeapList32` record filled by `Heap32ListFirst()` or `Heap32ListNext()`. The `lphe var` parameter of `Heap32First()` and `Heap32Next()` is of type `THeapEntry32`. This record contains descriptive information pertaining to the heap block and is defined as follows:

```
type
  THeapEntry32 = record
    dwSize: DWORD;
    hHandle: THandle;     // Handle of this heap block
    dwAddress: DWORD;     // Linear address of start of block
    dwBlockSize: DWORD;   // Size of block in bytes
    dwFlags: DWORD;
    dwLockCount: DWORD;
    dwResvd: DWORD;
    th32ProcessID: DWORD; // owning process
    th32HeapID: DWORD;    // heap block is in
  end;
```

- `dwSize` is the size of the record, and it should be initialized to `SizeOf(THeapEntry32)` prior to using the record.
- `hHandle` is the handle of the heap block.
- `dwAddress` is the linear address of the start of the heap block.
- `dwBlockSize` is the size, in bytes, of this heap block.
- `dwFlags` describes the type of heap block. This field can have any of the values shown in the following table:

Value	*Meaning*
LF32_FIXED	The memory block has a fixed (unmovable) location.
LF32_FREE	The memory block is not used.
LF32_MOVEABLE	The memory block location can be moved.

- `dwLockCount` is the lock count of the memory block. This value is increased by one every time the process calls `GlobalLock()` or `LocalLock()` on this block.
- `dwResvd` is reserved at this time and shouldn't be used.
- `th32ProcessID` is the identifier of the owning process.
- `th32HeapID` is the identifier of the heap to which the block belongs.

Because you must first walk the list of heap lists before you can walk the heap block list, the code for heap block walking is a bit—but not much—more complex than what you've seen so far. As you see in the `TDetailForm.WalkHeaps()` method that follows, the trick is to nest the `Heap32First()`/`Heap32Next()` loop within the `Heap32ListFirst()`/`Heap32ListNext()` loop. The method adds an additional level of complexity by adding a `PHeapEntry32` record pointer to the objects in the heap list portion of the `DetailLists` array. This is done so that information on the heap is available later when viewing heap contents:

```
procedure TWin95DetailForm.WalkHeaps;
{ Uses ToolHelp32 functions to walk list of heaps }
var
  HL: THeapList32;
  HE: THeapEntry32;
  PHE: PHeapEntry32;
begin
  DetailLists[ltHeap].Clear;
  HL.dwSize := SizeOf(HL);
  HE.dwSize := SizeOf(HE);
  if Heap32ListFirst(FCurSnap, HL) then
    repeat
      if Heap32First(HE, HL.th32ProcessID, HL.th32HeapID) then
        repeat
          New(PHE);       // need to make copy of THeapList32 record so we
          PHE^ := HE;     // have enough info to view heap later
          DetailLists[ltHeap].AddObject(Format(SHeapStr, [HL.th32HeapID,
            Pointer(HE.dwAddress), HE.dwBlockSize,
            GetHeapFlagString(HE.dwFlags)]), TObject(PHE));
        until not Heap32Next(HE);
    until not Heap32ListNext(FCurSnap, HL);
  HeapListAlloc := True;
end;
```

Figure 14.9 shows the detail form with the heap block list visible.

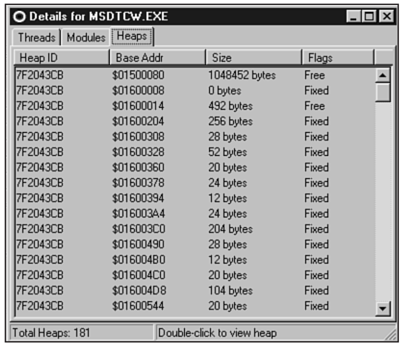

FIGURE 14.9

Viewing Windows heap blocks in the detail form under Windows 98.

Heap Viewing

Up to this point, you've learned about every function in the ToolHelp32 API except for one: `ToolHelp32ReadProcessMemory()`. To make sure you finish this chapter with a warm, fuzzy feeling, you'll also learn about this function.

`ToolHelp32ReadProcessMemory()` is declared this way:

```
function Toolhelp32ReadProcessMemory(th32ProcessID: DWORD;
  lpBaseAddress: Pointer; var lpBuffer; cbRead: DWORD;
  var lpNumberOfBytesRead: DWORD): BOOL; stdcall;
```

This function is arguably the most powerful and definitely the most fun in ToolHelp32 because it actually allows you to peek into the memory space of another process. The parameters for this function are as follows:

- `th32ProcessID` is the identifier of the process whose memory you want to read. You can obtain this value by any of the ToolHelp32 enumeration functions. You can pass zero in this parameter to indicate the current process.
- `lpBaseAddress` is the linear address of the first byte of memory you want to read in process `th32ProcessID`. You need to use the right process with the right address because any given linear address is meaningful only to a particular process.
- `lpBuffer` is the buffer to which you want to copy process `th32ProcessID`'s memory. You must ensure that memory is allocated for this buffer.
- `cbRead` is the number of bytes to read from process `th32ProcessID`, starting at `lpBaseAddress`.
- `lpNumberOfBytesRead` is filled in by the function before it returns. This is the number of bytes actually read from process `th32ProcessID`.

Once the memory of a particular process is copied to a local buffer using this function, SysInfo shows another modal form, `HeapViewForm`, which formats the memory dump for viewing. To handle the formatting, `HeapViewForm` makes use of a custom component called `TMemView` for viewing a memory dump. Because discussing the internals of the `TMemView` control is beyond the focus of this chapter (and because the control isn't terribly complex), you can browse the source code for the control on this book's CD-ROM. The following method of `TDetailForm`, `DetailLBDblClick()`, is called when the user double-clicks in the `THeaderListbox` component's `DetailLB`:

```
procedure TWin95DetailForm.DetailLBDblClick(Sender: TObject);
{ This procedure is called when the user double clicks on an item }
{ in DetailLB.  If the current tab page is heaps, a heap view     }
{ form is presented to the user. }
var
  NumRead: DWORD;
  HE: THeapEntry32;
  MemSize: integer;
begin
  inherited;
  if DetailTabs.TabIndex = 2 then
  begin
    HE := PHeapEntry32(DetailLB.Items.Objects[DetailLB.ItemIndex])^;
    MemSize := HE.dwBlockSize;         // get heap size
    { if heap is too big, use ProcMemMaxSize }
    if MemSize > ProcMemMaxSize then MemSize := ProcMemMaxSize;
    ProcMem := AllocMem(MemSize);     // allocate a temp buffer
    Screen.Cursor := crHourGlass;
    try
      { Copy heap into temp buffer }
      if Toolhelp32ReadProcessMemory(FCurProc.th32ProcessID,
        Pointer(HE.dwAddress), ProcMem^, MemSize, NumRead) then
        { point HeapView control at temp buffer }
        ShowHeapView(ProcMem, MemSize)
      else
        MessageDlg(SHeapReadErr, mtInformation, [mbOk], 0);
    finally
      Screen.Cursor := crDefault;
      FreeMem(ProcMem, MemSize);
    end;
  end;
end;
```

This method first checks to see whether the current tab page is the heap list page. If so, it allocates a temporary buffer and passes it to the `ToolHelp32ReadProcessMemory()` function to be filled. Once the buffer is filled, it's displayed in the `TMemView` control `HeapView`, and `HeapViewForm` is shown modally. When the form returns from the `ShowModal()` call, the buffer is freed. Figure 14.10 shows a heap view in action.

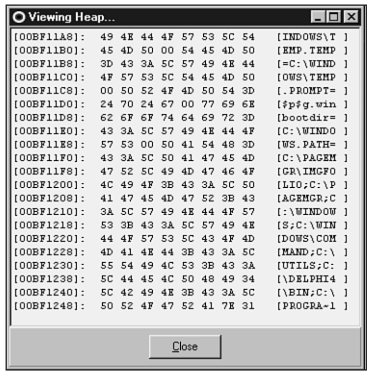

FIGURE 14.10

Viewing the heap of another Windows 98 process.

The Source

Listings 14.2 and 14.3 show the complete source for the `W9xInfo.pas` and `Detail9x.pas` units, respectively.

LISTING 14.2 `W9xInfo.pas`, Obtaining Process Information Under Windows 95/98

```
unit W9xInfo;

interface

uses Windows, InfoInt, Classes, TlHelp32, Controls, ComCtrls;

type
  TWin9xInfo = class(TInterfacedObject, IWin32Info)
  private
    FProcList: TList;
    FWinIcon: HICON;
    FSnap: THandle;
    procedure Refresh;
  public
    constructor Create;
    destructor Destroy; override;
    procedure FillProcessInfoList(ListView: TListView; ImageList: TImageList);
```

continues

LISTING 14.2 Continued

```
    procedure ShowProcessProperties(Cookie: Pointer);
  end;

implementation

uses ShellAPI, CommCtrl, SysUtils, Detail9x;

const
  ProcessInfoCaptions: array[0..3] of string = (
    'ProcessName', 'Threads', 'ID', 'ParentID');

{ TProcList }

type
  TProcList = class(TList)
    procedure Clear; override;
  end;

procedure TProcList.Clear;
var
  I: Integer;
begin
  for I := 0 to Count - 1 do Dispose(PProcessEntry32(Items[I]));
  inherited Clear;
end;

{ TWin95Info }

constructor TWin9xInfo.Create;
begin
  FProcList := TProcList.Create;
  FWinIcon := LoadImage(0, IDI_WINLOGO, IMAGE_ICON, LR_DEFAULTSIZE,
    LR_DEFAULTSIZE, LR_DEFAULTSIZE or LR_DEFAULTCOLOR or LR_SHARED);
end;

destructor TWin9xInfo.Destroy;
begin
  DestroyIcon(FWinIcon);
  if FSnap > 0 then CloseHandle(FSnap);
  FProcList.Free;
  inherited Destroy;
end;

procedure TWin9xInfo.FillProcessInfoList(ListView: TListView;
  ImageList: TImageList);
```

```
var
  I: Integer;
  ExeFile: string;
  PE: TProcessEntry32;
  HAppIcon: HIcon;
begin
  Refresh;
  ListView.Columns.Clear;
  ListView.Items.Clear;
  for I := Low(ProcessInfoCaptions) to High(ProcessInfoCaptions) do
    with ListView.Columns.Add do
    begin
      if I = 0 then Width := 285
      else Width := 75;
      Caption := ProcessInfoCaptions[I];
    end;
  for I := 0 to FProcList.Count - 1 do
  begin
    PE := PProcessEntry32(FProcList.Items[I])^;
    HAppIcon := ExtractIcon(HInstance, PE.szExeFile, 0);
    try
      if HAppIcon = 0 then HAppIcon := FWinIcon;
      ExeFile := PE.szExeFile;
      if ListView.ViewStyle = vsList then
        ExeFile := ExtractFileName(ExeFile);
      // insert new item, set its caption, add subitems
      with ListView.Items.Add, SubItems do
      begin
        Caption := ExeFile;
        Data := FProcList.Items[I];
        Add(IntToStr(PE.cntThreads));
        Add(IntToHex(PE.th32ProcessID, 8));
        Add(IntToHex(PE.th32ParentProcessID, 8));
        if ImageList <> nil then
          ImageIndex := ImageList_AddIcon(ImageList.Handle, HAppIcon);
      end;
    finally
      if HAppIcon <> FWinIcon then DestroyIcon(HAppIcon);
    end;
  end;
end;

procedure TWin9xInfo.Refresh;
var
  PE: TProcessEntry32;
  PPE: PProcessEntry32;
```


continues

LISTING 14.2 Continued

```
begin
  FProcList.Clear;
  if FSnap > 0 then CloseHandle(FSnap);
  FSnap := CreateToolHelp32Snapshot(TH32CS_SNAPPROCESS, 0);
  if FSnap = INVALID_HANDLE_VALUE then
    raise Exception.Create('CreateToolHelp32Snapshot failed');
  PE.dwSize := SizeOf(PE);
  if Process32First(FSnap, PE) then               // get process
    repeat
      New(PPE);                                   // create new PPE
      PPE^ := PE;                                 // fill it
      FProcList.Add(PPE);                         // add it to list
    until not Process32Next(FSnap, PE);           // get next process
end;

procedure TWin9xInfo.ShowProcessProperties(Cookie: Pointer);
begin
  ShowProcessDetails(PProcessEntry32(Cookie));
end;

end.
```

LISTING 14.3 `Detail9x.pas`, Obtaining Process Details Under Windows 95/98

```
unit Detail9x;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ComCtrls, HeadList, TlHelp32, Menus, SysMain, DetBase;

type
  TListType = (ltThread, ltModule, ltHeap);

  TWin9xDetailForm = class(TBaseDetailForm)
    procedure DetailTabsChange(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure DetailLBDblClick(Sender: TObject);
  private
    FCurSnap: THandle;
    FCurProc: TProcessEntry32;
    DetailLists: array[TListType] of TStringList;
```

```
    ProcMem: PByte;
    HeapListAlloc: Boolean;
    procedure FreeHeapList;
    procedure ShowList(ListType: TListType);
    procedure WalkThreads;
    procedure WalkHeaps;
    procedure WalkModules;
  public
    procedure NewProcess(P: PProcessEntry32);
  end;

procedure ShowProcessDetails(P: PProcessEntry32);

implementation

{$R *.DFM}

uses ProcMem;

const
  { Array of strings which goes into the header of each respective list. }
  HeaderStrs: array[TListType] of TDetailStrings = (
      ('Thread ID', 'Base Priority', 'Delta Priority', 'Usage Count'),
      ('Module', 'Base Addr', 'Size', 'Usage Count'),
      ('Heap ID', 'Base Addr', 'Size', 'Flags'));

  { Array of strings which goes into the footer of each list. }
  ACountStrs: array[TListType] of string[31] = (
      'Total Threads: %d', 'Total Modules: %d', 'Total Heaps: %d');

  TabStrs: array[TListType] of string[7] = ('Threads', 'Modules', 'Heaps');

  SCaptionStr  = 'Details for %s';       // form caption
  SThreadStr   = '%x'#1'%s'#1'%s'#1'%d'; // id, base pri, delta pri, usage
  SModuleStr   = '%s'#1'$%p'#1'%d bytes'#1'%d'; // name, addr, size, usage
  SHeapStr     = '%x'#1'$%p'#1'%d bytes'#1'%s'; // ID, addr, size, flags
  SHeapReadErr = 'This heap is not accessible for read access.';

  ProcMemMaxSize = $7FFE;                // max size of heap view

procedure ShowProcessDetails(P: PProcessEntry32);
var
  I: TListType;
begin
  with TWin9xDetailForm.Create(Application) do
    try
```


continues

LISTING 14.3 Continued

```
      for I := Low(TabStrs) to High(TabStrs) do
        DetailTabs.Tabs.Add(TabStrs[I]);
      NewProcess(P);
      Font := MainForm.Font;
      ShowModal;
    finally
      Free;
    end;
end;

function GetThreadPriorityString(Priority: Integer): string;
{ Returns string describing thread priority }
begin
  case Priority of
    THREAD_PRIORITY_IDLE:          Result := '%d (Idle)';
    THREAD_PRIORITY_LOWEST:        Result := '%d (Lowest)';
    THREAD_PRIORITY_BELOW_NORMAL:  Result := '%d (Below Normal)';
    THREAD_PRIORITY_NORMAL:        Result := '%d (Normal)';
    THREAD_PRIORITY_ABOVE_NORMAL:  Result := '%d (Above Normal)';
    THREAD_PRIORITY_HIGHEST:       Result := '%d (Highest)';
    THREAD_PRIORITY_TIME_CRITICAL: Result := '%d (Time critical)';
  else
    Result := '%d (unknown)';
  end;
  Result := Format(Result, [Priority]);
end;

function GetClassPriorityString(Priority: DWORD): String;
{ returns string describing process priority class }
begin
  case Priority of
    4:   Result := '%d (Idle)';
    8:   Result := '%d (Normal)';
    13:  Result := '%d (High)';
    24:  Result := '%d (Real time)';
  else
    Result := '%d (non-standard)';
  end;
  Result := Format(Result, [Priority]);
end;

function GetHeapFlagString(Flag: DWORD): String;
{ Returns a string describing a heap flag }
begin
  case Flag of
```

```
    LF32_FIXED:    Result := 'Fixed';
    LF32_FREE:     Result := 'Free';
    LF32_MOVEABLE: Result := 'Moveable';
  end;
end;

procedure TWin9xDetailForm.ShowList(ListType: TListType);
{ Shows appropriate thread, heap, or module list in DetailLB }
var
  i: Integer;
begin
  Screen.Cursor := crHourGlass;
  try
    with DetailLB do
    begin
      for i := 0 to 3 do
        Sections[i].Text := HeaderStrs[ListType, i];
      Items.Clear;
      Items.Assign(DetailLists[ListType]);
    end;
     DetailSB.Panels[0].Text := Format(ACountStrs[ListType],
       [DetailLists[ListType].Count]);
     if ListType = ltHeap then
       DetailSB.Panels[1].Text := 'Double-click to view heap'
     else
       DetailSB.Panels[1].Text := '';
  finally
    Screen.Cursor := crDefault;
  end;
end;

procedure TWin9xDetailForm.WalkThreads;
{ Uses ToolHelp32 functions to walk list of threads }
var
  T: TThreadEntry32;
begin
  DetailLists[ltThread].Clear;
  T.dwSize := SizeOf(T);
  if Thread32First(FCurSnap, T) then
    repeat
      { Make sure thread is for current process }
      if T.th32OwnerProcessID = FCurProc.th32ProcessID then
        DetailLists[ltThread].Add(Format(SThreadStr, [T.th32ThreadID,
          GetClassPriorityString(T.tpBasePri),
          GetThreadPriorityString(T.tpDeltaPri), T.cntUsage]));
    until not Thread32Next(FCurSnap, T);
```

continues

LISTING 14.3 Continued

```
end;

procedure TWin9xDetailForm.WalkModules;
{ Uses ToolHelp32 functions to walk list of modules }
var
  M: TModuleEntry32;
begin
  DetailLists[ltModule].Clear;
  M.dwSize := SizeOf(M);
  if Module32First(FCurSnap, M) then
    repeat
      DetailLists[ltModule].Add(Format(SModuleStr, [M.szModule, M.ModBaseAddr,
        M.ModBaseSize, M.ProcCntUsage]));
    until not Module32Next(FCurSnap, M);
end;

procedure TWin9xDetailForm.WalkHeaps;
{ Uses ToolHelp32 functions to walk list of heaps }
var
  HL: THeapList32;
  HE: THeapEntry32;
  PHE: PHeapEntry32;
begin
  DetailLists[ltHeap].Clear;
  HL.dwSize := SizeOf(HL);
  HE.dwSize := SizeOf(HE);
  if Heap32ListFirst(FCurSnap, HL) then
    repeat
      if Heap32First(HE, HL.th32ProcessID, HL.th32HeapID) then
        repeat
          New(PHE);      // need to make copy of THeapList32 record so we
          PHE^ := HE;    // have enough info to view heap later
          DetailLists[ltHeap].AddObject(Format(SHeapStr, [HL.th32HeapID,
            Pointer(HE.dwAddress), HE.dwBlockSize,
            GetHeapFlagString(HE.dwFlags)]), TObject(PHE));
        until not Heap32Next(HE);
    until not Heap32ListNext(FCurSnap, HL);
  HeapListAlloc := True;
end;

procedure TWin9xDetailForm.FreeHeapList;
{ Since special allocation of PHeapList32 objects are added to the list, }
{ these must be freed. }
var
  i: integer;
```

```
begin
  for i := 0 to DetailLists[ltHeap].Count - 1 do
    Dispose(PHeapEntry32(DetailLists[ltHeap].Objects[i]));
end;

procedure TWin9xDetailForm.NewProcess(P: PProcessEntry32);
{ This procedure is called from the main form to show the detail }
{ form for a particular process. }
begin
  { Create a snapshot for the current process }
  FCurSnap := CreateToolhelp32Snapshot(TH32CS_SNAPALL, P^.th32ProcessID);
  if FCurSnap = INVALID_HANDLE_VALUE then
    raise Exception.Create('CreateToolHelp32Snapshot failed');
  HeapListAlloc := False;
  Screen.Cursor := crHourGlass;
  try
    FCurProc := P^;
    { Include module name in detail form caption }
    Caption := Format(SCaptionStr, [ExtractFileName(FCurProc.szExeFile)]);
    WalkThreads;                        // walk ToolHelp32 lists
    WalkModules;
    WalkHeaps;
    DetailTabs.TabIndex := 0;           // 0 = thread tab
    ShowList(ltThread);                 // show thread page first
  finally
    Screen.Cursor := crDefault;
    if HeapListAlloc then FreeHeapList;
    CloseHandle(FCurSnap);              // close snapshot handle
  end;
end;

procedure TWin9xDetailForm.DetailTabsChange(Sender: TObject);
{ OnChange event handler for tab set.  Sets visible list to jive with tabs. }
begin
  inherited;
  ShowList(TListType(DetailTabs.TabIndex));
end;

procedure TWin9xDetailForm.FormCreate(Sender: TObject);
var
  LT: TListType;
begin
  inherited;
  { Dispose of lists }
  for LT := Low(TListType) to High(TListType) do
    DetailLists[LT] := TStringList.Create;
```


continues

LISTING 14.3 Continued

```
end;

procedure TWin9xDetailForm.FormDestroy(Sender: TObject);
var
  LT: TListType;
begin
  inherited;
  { Dispose of lists }
  for LT := Low(TListType) to High(TListType) do
    DetailLists[LT].Free;
end;

procedure TWin9xDetailForm.DetailLBDblClick(Sender: TObject);
{ This procedure is called when the user double clicks on an item }
{ in DetailLB.  If the current tab page is heaps, a heap view     }
{ form is presented to the user. }
var
  NumRead: DWORD;
  HE: THeapEntry32;
  MemSize: integer;
begin
  inherited;
  if DetailTabs.TabIndex = 2 then
  begin
    HE := PHeapEntry32(DetailLB.Items.Objects[DetailLB.ItemIndex])^;
    MemSize := HE.dwBlockSize;          // get heap size
    { if heap is too big, use ProcMemMaxSize }
    if MemSize > ProcMemMaxSize then MemSize := ProcMemMaxSize;
    ProcMem := AllocMem(MemSize);     // allocate a temp buffer
    Screen.Cursor := crHourGlass;
    try
      { Copy heap into temp buffer }
      if Toolhelp32ReadProcessMemory(FCurProc.th32ProcessID,
        Pointer(HE.dwAddress), ProcMem^, MemSize, NumRead) then
        { point HeapView control at temp buffer }
        ShowHeapView(ProcMem, MemSize)
      else
        MessageDlg(SHeapReadErr, mtInformation, [mbOk], 0);
    finally
      Screen.Cursor := crDefault;
      FreeMem(ProcMem, MemSize);
    end;
  end;
end;

end.
```

Windows NT/2000: PSAPI

As we mentioned earlier, the ToolHelp32 API does not exist under Windows NT/2000. The Windows Platform SDK, however, provides a DLL called `PSAPI.DLL` from which you can obtain the same types of information as with ToolHelp32 under Windows NT/2000, including

- Running processes
- Modules loaded per process
- Loaded device drivers
- Process memory information
- Files memory mapped per process

Later versions of Windows NT and all versions of Windows 2000 include `PSAPI.DLL`, although you can redistribute this file if you wish to deploy it to the users of your applications. Delphi provides an interface unit for this DLL called `PSAPI.pas`, which loads all its functions dynamically. Therefore, applications that use this unit will run on machines with or without `PSAPI.DLL` (of course, the functions won't work without `PSAPI.DLL` installed, but the application will run).

The first step in obtaining process information using PSAPI is to call `EnumProcesses()`, which is defined as follows:

```
function EnumProcesses(lpidProcess: LPDWORD; cb: DWORD;
  var cbNeeded: DWORD): BOOL;
```

- `lpidProcess` is a pointer to an array of DWORDs that will be filled in with process IDs by the function.
- `cb` contains the number of DWORDs in the array passed in `lpidProcess`.
- Upon return, `cbNeeded` will hold the number of bytes copied into `lpidProcess`. The expression `cbNeeded div SizeOf(DWORD)` will provide the number of elements copied into the array and therefore the number of running processes.

After calling this function, the array passed in `lpidProcess` will contain a bunch of process IDs. Process IDs aren't particularly useful on their own, but you can pass a process ID to the `OpenProcess()` API function in order to obtain a process handle. Once you have a process handle, you can call other PSAPI functions or even other Win32 API functions that call for process handles.

PSAPI provides a similar function for obtaining information on loaded device drivers called—we'll give you one guess—`EnumDeviceDrivers()`. This method is defined as follows:

```
function EnumDeviceDrivers(lpImageBase: PPointer; cb: DWORD;
  var lpcbNeeded: DWORD): BOOL;
```

- `lpImageBase` is a pointer to an array of `Pointers` that will be filled with the base address of each device driver.

- cb contains the number of Pointers in the array passed in lpImageBase.
- Upon return, lpcbNeeded will hold the number of bytes copied to lpImageBase.

In the SysInfo project ID is a unit called WNTInfo.pas, which contains a class called TWinNTInfo that implements IWin32Info. This class contains a private method called Refresh(), which obtains process and device driver information:

```
procedure TWinNTInfo.Refresh;
var
  Count: DWORD;
  BigArray: array[0..$3FFF - 1] of DWORD;
begin
  // Get array of process IDs
  if not EnumProcesses(@BigArray, SizeOf(BigArray), Count) then
    raise Exception.Create(SFailMessage);
  SetLength(FProcList, Count div SizeOf(DWORD));
  Move(BigArray, FProcList[0], Count);
  // Get array of Driver addresses
  if not EnumDeviceDrivers(@BigArray, SizeOf(BigArray), Count) then
    raise Exception.Create(SFailMessage);
  SetLength(FDrvList, Count div SizeOf(DWORD));
  Move(BigArray, FDrvList[0], Count);
end;
```

This method initially passes a local called BigArray to EnumProcesses() and EnumDeviceDrivers() and then moves the data from BigArray into dynamic arrays called FProcList and FDrvList. The reason for this ungainly implementation of these functions is that neither EnumProcesses() nor EnumDeviceDrivers() provide a means for determining how many elements will be returned before allocating an array. We are therefore stuck passing a large array (that we hope is large enough) to the methods and copying the result to an appropriately sized dynamic array.

The FillProcessInfoList() method for TWinNTInfo calls two helper methods—FillProcesses() and FillDrivers()—to fill the contents of the TListView on the main form. FillProcesses() is shown here:

```
procedure TWinNTInfo.FillProcesses(ListView: TListView;
  ImageList: TImageList);
var
  I: Integer;
  Count: DWORD;
  ProcHand: THandle;
  ModHand: HMODULE;
```

```
  HAppIcon: HICON;
  ModName: array[0..MAX_PATH] of char;
begin
  for I := Low(FProcList) to High(FProcList) do
  begin
    ProcHand := OpenProcess(PROCESS_QUERY_INFORMATION or PROCESS_VM_READ,
      False, FProcList[I]);
    if ProcHand > 0 then
      try
        EnumProcessModules(Prochand, @ModHand, 1, Count);
        if GetModuleFileNameEx(Prochand, ModHand, ModName,
          SizeOf(ModName)) > 0 then
        begin
          HAppIcon := ExtractIcon(HInstance, ModName, 0);
          try
            if HAppIcon = 0 then HAppIcon := FWinIcon;
            with ListView.Items.Add, SubItems do
            begin
              Caption := ModName;                    // file name
              Data := Pointer(FProcList[I]);         // save ID
              Add(SProcName);                        // "process"
              Add(IntToStr(FProcList[I]));           // process ID
              Add('$' + IntToHex(ProcHand, 8));      // process handle
              // priority class
              Add(GetPriorityClassString(GetPriorityClass(ProcHand)));
              // icon
              if ImageList <> nil then
                ImageIndex := ImageList_AddIcon(ImageList.Handle,
                  HAppIcon);
            end;
          finally
            if HAppIcon <> FWinIcon then DestroyIcon(HAppIcon);
          end;
        end;
      finally
        CloseHandle(ProcHand);
      end;
  end;
end;
```

This method uses `OpenProcess()` to convert each process ID into a process handle. Several flags can be passed to this method in the first parameter, but for purposes of querying information with PSAPI, `PROCESS_QUERY_INFORMATION` and `PROCESS_VM_READ` together work best.

Given a process handle, the code then calls `EnumProcessModules()` to obtain the filename for the process. This method is defined as follows:

```
function EnumProcessModules(hProcess: THandle; lphModule: LPDWORD;
  cb: DWORD; var lpcbNeeded: DWORD): BOOL;
```

This method works in a manner similar to the other PSAPI functions: `hProcess` is a process handle, `lphModule` is a pointer to an array of module handles, `cb` indicates the number of elements in the array, and the final parameter returns the number of bytes copied to `lphModule`.

Because we're only interested in the primary module for this process right now, we only pass an array of one element. The first module returned by `EnumProcessModules()` is the primary module for the process. All the process information is then added to the `TListView` component in a manner similar to that shown in `TWin9xInfo`.

`FillDrivers()` functions in a like manner, except that it uses the `GetDeviceDriverFileName()` method shown here:

```
function GetDeviceDriverFileName(ImageBase: Pointer; lpFileName: PChar;
  nSize: DWORD): DWORD;
```

This method takes the image base of the device driver as the first parameter, a pointer to a string buffer as the second parameter, and the size of the buffer in the last parameter. Upon successful return, `lpFileName` will contain the filename of the device driver. Our use of this method is shown in the following code:

```
procedure TWinNTInfo.FillDrivers(ListView: TListView;
  ImageList: TImageList);
var
  I: Integer;
  DrvName: array[0..MAX_PATH] of char;
begin
  for I := Low(FDrvList) to High(FDrvList) do
    if GetDeviceDriverFileName(FDrvList[I], DrvName, SizeOf(DrvName)) > 0 then
      with ListView.Items.Add do
      begin
        Caption := DrvName;
        SubItems.Add(SDrvName);
        SubItems.Add('$' + IntToHex(Integer(FDrvList[I]), 8));
      end;
end;
```

Figure 14.11 shows the SysInfo application running on a Windows NT 4.0 machine.

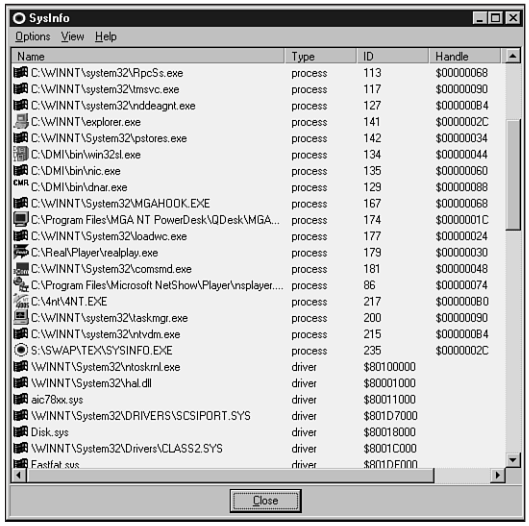

FIGURE 14.11

Browsing Windows NT processes and drivers.

Like `TWin95Info`'s implementation of `ShowProcessProperties()`, `TWinNTInfo` calls out to another unit to display a form containing more process information. In particular, the additional information pertains to process modules and memory usage. The method that does the work of obtaining this information resides in the `TWinNTDetailForm` class in the `DetailNT` unit, and it's shown in the following code:

```
procedure TWinNTDetailForm.NewProcess(ProcessID: DWORD);
const
  AddrMask = DWORD($FFFFF000);
var
  I, Count: Integer;
  ProcHand: THandle;
  WSPtr: Pointer;
  ModHandles: array[0..$3FFF - 1] of DWORD;
  WorkingSet: array[0..$3FFF - 1] of DWORD;
  ModInfo: TModuleInfo;
  ModName, MapFileName: array[0..MAX_PATH] of char;
begin
  ProcHand := OpenProcess(PROCESS_QUERY_INFORMATION or PROCESS_VM_READ, False,
    ProcessID);
```

```
  if ProcHand = 0 then
    raise Exception.Create('No information available for this process/driver');
  try
    EnumProcessModules(ProcHand, @ModHandles, SizeOf(ModHandles), Count);
    for I := 0 to (Count div SizeOf(DWORD)) - 1 do
      if (GetModuleFileNameEx(ProcHand, ModHandles[I], ModName,
        SizeOf(ModName)) > 0) and GetModuleInformation(ProcHand,
        ModHandles[I], @ModInfo, SizeOf(ModInfo)) then
        with ModInfo do
          DetailLists[ltModules].Add(Format(SModuleStr, [ModName, lpBaseOfDll,
            SizeOfImage, EntryPoint]));
    if QueryWorkingSet(ProcHand, @WorkingSet, SizeOf(WorkingSet)) then
      for I := 1 to WorkingSet[0] do
      begin
        WSPtr := Pointer(WorkingSet[I] and AddrMask);
        GetMappedFileName(ProcHand, WSPtr, MapFileName, SizeOf(MapFileName));
        DetailLists[ltMemory].Add(Format(SMemoryStr, [WSPtr,
          MemoryTypeToString(WorkingSet[I]), MapFileName]));
      end;
  finally
    CloseHandle(ProcHand);
  end;
end;
```

As you can see, this method makes calls to `OpenProcess()` and `EnumProcessModules()`, about which you've already learned. This method also calls a PSAPI function called `QueryWorkingSet()`, however, to obtain memory information for a process. This function is defined as follows:

```
function QueryWorkingSet(hProcess: THandle; pv: Pointer; cb: DWORD): BOOL;
```

`hProcess` is the process handle. `pv` is a pointer to an array of DWORDs, and `cb` holds the number of elements in the array. Upon return, `pv` will point to an array of DWORDs. The upper 20 bits of this DWORD hold the base address of a memory page, and the lower 12 bits of each DWORD hold flags that indicate whether the page is readable, writable, executable, and so on.

Figures 14.12 and 14.13 show module and memory details under Windows NT. Listings 14.4 and 14.5 show the `WNTInfo.pas` and `DetailNT.pas` units, respectively.

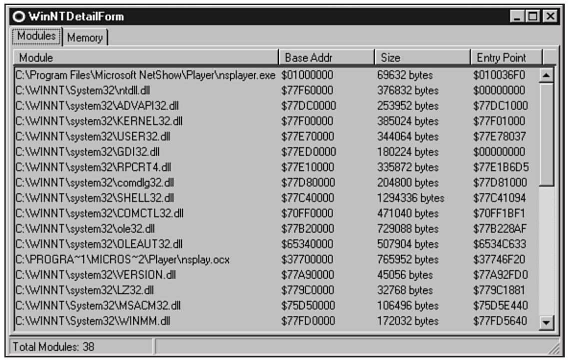

FIGURE 14.12
Viewing Windows NT process modules.

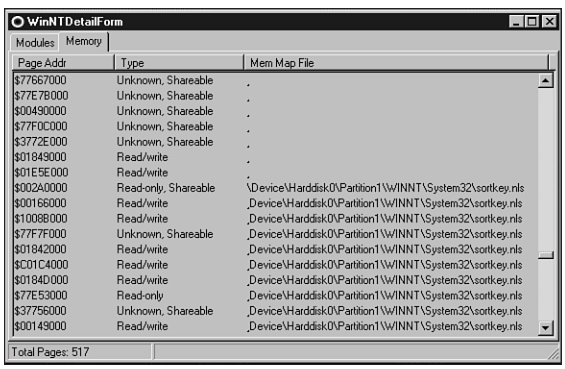

FIGURE 14.13
Viewing Windows NT process memory details.

LISTING 14.4 `WNTInfo.pas`, Obtaining Process Information Under Windows NT/2000

```
unit WNTInfo;

interface

uses InfoInt, Windows, Classes, ComCtrls, Controls;

type
  TWinNTInfo = class(TInterfacedObject, IWin32Info)
  private
```

continues

LISTING 14.4 Continued

```
    FProcList: array of DWORD;
    FDrvlist: array of Pointer;
    FWinIcon: HICON;
    procedure FillProcesses(ListView: TListView; ImageList: TImageList);
    procedure FillDrivers(ListView: TListView; ImageList: TImageList);
    procedure Refresh;
  public
    constructor Create;
    destructor Destroy; override;
    procedure FillProcessInfoList(ListView: TListView;
      ImageList: TImageList);
    procedure ShowProcessProperties(Cookie: Pointer);
  end;

implementation

uses SysUtils, PSAPI, ShellAPI, CommCtrl, DetailNT;

const
  SFailMessage = 'Failed to enumerate processes or drivers.  Make sure '+
    'PSAPI.DLL is installed on your system.';
  SDrvName = 'driver';
  SProcname = 'process';
  ProcessInfoCaptions: array[0..4] of string = (
    'Name', 'Type', 'ID', 'Handle', 'Priority');

function GetPriorityClassString(PriorityClass: Integer): string;
begin
  case PriorityClass of
    HIGH_PRIORITY_CLASS: Result := 'High';
    IDLE_PRIORITY_CLASS: Result := 'Idle';
    NORMAL_PRIORITY_CLASS: Result := 'Normal';
    REALTIME_PRIORITY_CLASS: Result := 'Realtime';
  else
    Result := Format('Unknown ($%x)', [PriorityClass]);
  end;
end;

{ TWinNTInfo }

constructor TWinNTInfo.Create;
```

```
begin
  FWinIcon := LoadImage(0, IDI_WINLOGO, IMAGE_ICON, LR_DEFAULTSIZE,
    LR_DEFAULTSIZE, LR_DEFAULTSIZE or LR_DEFAULTCOLOR or LR_SHARED);
end;

destructor TWinNTInfo.Destroy;
begin
  DestroyIcon(FWinIcon);
  inherited Destroy;
end;

procedure TWinNTInfo.FillDrivers(ListView: TListView;
  ImageList: TImageList);
var
  I: Integer;
  DrvName: array[0..MAX_PATH] of char;
begin
  for I := Low(FDrvList) to High(FDrvList) do
    if GetDeviceDriverFileName(FDrvList[I], DrvName,
      SizeOf(DrvName)) > 0 then
      with ListView.Items.Add do
      begin
        Caption := DrvName;
        SubItems.Add(SDrvName);
        SubItems.Add('$' + IntToHex(Integer(FDrvList[I]), 8));
      end;
end;

procedure TWinNTInfo.FillProcesses(ListView: TListView;
  ImageList: TImageList);
var
  I: Integer;
  Count: DWORD;
  ProcHand: THandle;
  ModHand: HMODULE;
  HAppIcon: HICON;
  ModName: array[0..MAX_PATH] of char;
begin
  for I := Low(FProcList) to High(FProcList) do
  begin
    ProcHand := OpenProcess(PROCESS_QUERY_INFORMATION or PROCESS_VM_READ,
      False, FProcList[I]);
```


continues

LISTING 14.4 Continued

```
    if ProcHand > 0 then
      try
        EnumProcessModules(Prochand, @ModHand, 1, Count);
        if GetModuleFileNameEx(Prochand, ModHand, ModName,
          SizeOf(ModName)) > 0 then
        begin
          HAppIcon := ExtractIcon(HInstance, ModName, 0);
          try
            if HAppIcon = 0 then HAppIcon := FWinIcon;
            with ListView.Items.Add, SubItems do
            begin
              Caption := ModName;                    // file name
              Data := Pointer(FProcList[I]);         // save ID
              Add(SProcName);                        // "process"
              Add(IntToStr(FProcList[I]));           // process ID
              Add('$' + IntToHex(ProcHand, 8));      // process handle
              // priority class
              Add(GetPriorityClassString(GetPriorityClass(ProcHand)));
              // icon
              if ImageList <> nil then
                ImageIndex := ImageList_AddIcon(ImageList.Handle,
                  HAppIcon);
            end;
          finally
            if HAppIcon <> FWinIcon then DestroyIcon(HAppIcon);
          end;
        end;
      finally
        CloseHandle(ProcHand);
      end;
  end;
end;

procedure TWinNTInfo.FillProcessInfoList(ListView: TListView;
  ImageList: TImageList);
var
  I: Integer;
begin
  Refresh;
  ListView.Columns.Clear;
  ListView.Items.Clear;
  for I := Low(ProcessInfoCaptions) to High(ProcessInfoCaptions) do
```

```
    with ListView.Columns.Add do
    begin
      if I = 0 then Width := 285
      else Width := 75;
      Caption := ProcessInfoCaptions[I];
    end;
  FillProcesses(ListView, ImageList);  // Add processes to listview
  FillDrivers(ListView, ImageList);    // Add device drivers to listview
end;

procedure TWinNTInfo.Refresh;
var
  Count: DWORD;
  BigArray: array[0..$3FFF - 1] of DWORD;
begin
  // Get array of process IDs
  if not EnumProcesses(@BigArray, SizeOf(BigArray), Count) then
    raise Exception.Create(SFailMessage);
  SetLength(FProcList, Count div SizeOf(DWORD));
  Move(BigArray, FProcList[0], Count);
  // Get array of Driver addresses
  if not EnumDeviceDrivers(@BigArray, SizeOf(BigArray), Count) then
    raise Exception.Create(SFailMessage);
  SetLength(FDrvList, Count div SizeOf(DWORD));
  Move(BigArray, FDrvList[0], Count);
end;

procedure TWinNTInfo.ShowProcessProperties(Cookie: Pointer);
begin
  ShowProcessDetails(DWORD(Cookie));
end;

end.
```


LISTING 14.5 `DetailNT.pas`, Obtaining Process Details Under Windows NT/2000

```
unit DetailNT;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
```

continues

LISTING 14.5 Continued

```
  DetBase, ComCtrls, HeadList;

type
  TListType = (ltModules, ltMemory);

  TWinNTDetailForm = class(TBaseDetailForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure DetailTabsChange(Sender: TObject);
  private
    FProcHand: THandle;
    DetailLists: array[TListType] of TStringList;
    procedure ShowList(ListType: TListType);
  public
    procedure NewProcess(ProcessID: DWORD);
  end;

procedure ShowProcessDetails(ProcessID: DWORD);

implementation

uses PSAPI;

{$R *.DFM}

const
  TabStrs: array[0..1] of string[7] = ('Modules', 'Memory');

  { Array of strings that goes into the footer of each list. }
  ACountStrs: array[TListType] of string[31] = (
      'Total Modules: %d', 'Total Pages: %d');

  { Array of strings that goes into the header of each respective list. }
  HeaderStrs: array[TListType] of TDetailStrings = (
    ('Module', 'Base Addr', 'Size', 'Entry Point'),
    ('Page Addr', 'Type', 'Mem Map File', ''));

  SCaptionStr  = 'Details for %s';               // form caption
  SModuleStr   = '%s'#1'$%p'#1'%d bytes'#1'$%p'; // name, addr, size, entry pt
  SMemoryStr   = '$%p'#1'%s'#1'%s';              // addr, type, mem map file

procedure ShowProcessDetails(ProcessID: DWORD);
```

```
var
  I: Integer;
begin
  with TWinNTDetailForm.Create(Application) do
    try
      for I := Low(TabStrs) to High(TabStrs) do
        DetailTabs.Tabs.Add(TabStrs[I]);
      NewProcess(ProcessID);
      ShowList(ltModules);
      ShowModal;
    finally
      Free;
    end;
end;

function MemoryTypeToString(Value: DWORD): string;
const
  TypeMask = DWORD($0000000F);
begin
  Result := '';
  case Value and TypeMask of
    1: Result := 'Read-only';
    2: Result := 'Executable';
    4: Result := 'Read/write';
    5: Result := 'Copy on write';
  else
    Result := 'Unknown';
  end;
  if Value and $100 <> 0 then
    Result := Result + ', Shareable';
end;

procedure TWinNTDetailForm.FormCreate(Sender: TObject);
var
  LT: TListType;
begin
  inherited;
  { Dispose of lists }
  for LT := Low(TListType) to High(TListType) do
    DetailLists[LT] := TStringList.Create;
end;

procedure TWinNTDetailForm.FormDestroy(Sender: TObject);
```


continues

LISTING 14.5 Continued

```
var
  LT: TListType;
begin
  inherited;
  { Dispose of lists }
  for LT := Low(TListType) to High(TListType) do
    DetailLists[LT].Free;
end;

procedure TWinNTDetailForm.NewProcess(ProcessID: DWORD);
const
  AddrMask = DWORD($FFFFF000);
var
  I, Count: Integer;
  ProcHand: THandle;
  WSPtr: Pointer;
  ModHandles: array[0..$3FFF - 1] of DWORD;
  WorkingSet: array[0..$3FFF - 1] of DWORD;
  ModInfo: TModuleInfo;
  ModName, MapFileName: array[0..MAX_PATH] of char;
begin
  ProcHand := OpenProcess(PROCESS_QUERY_INFORMATION or PROCESS_VM_READ, False,
    ProcessID);
  if ProcHand = 0 then
    raise Exception.Create('No information available for this process/driver');
  try
    EnumProcessModules(ProcHand, @ModHandles, SizeOf(ModHandles), Count);
    for I := 0 to (Count div SizeOf(DWORD)) - 1 do
      if (GetModuleFileNameEx(ProcHand, ModHandles[I], ModName,
        SizeOf(ModName)) > 0) and GetModuleInformation(ProcHand,
        ModHandles[I], @ModInfo, SizeOf(ModInfo)) then
        with ModInfo do
          DetailLists[ltModules].Add(Format(SModuleStr, [ModName, lpBaseOfDll,
            SizeOfImage, EntryPoint]));
    if QueryWorkingSet(ProcHand, @WorkingSet, SizeOf(WorkingSet)) then
      for I := 1 to WorkingSet[0] do
      begin
        WSPtr := Pointer(WorkingSet[I] and AddrMask);
        GetMappedFileName(ProcHand, WSPtr, MapFileName, SizeOf(MapFileName));
        DetailLists[ltMemory].Add(Format(SMemoryStr, [WSPtr,
          MemoryTypeToString(WorkingSet[I]), MapFileName]));
      end;
```

```
  finally
    CloseHandle(ProcHand);
  end;
end;

procedure TWinNTDetailForm.ShowList(ListType: TListType);
var
  I: Integer;
begin
  Screen.Cursor := crHourGlass;
  try
    with DetailLB do
    begin
      for I := 0 to 3 do
        Sections[I].Text := HeaderStrs[ListType, i];
      Items.Clear;
      Items.Assign(DetailLists[ListType]);
    end;
     DetailSB.Panels[0].Text := Format(ACountStrs[ListType],
       [DetailLists[ListType].Count]);
  finally
    Screen.Cursor := crDefault;
  end;
end;

procedure TWinNTDetailForm.DetailTabsChange(Sender: TObject);
begin
  inherited;
  ShowList(TListType(DetailTabs.TabIndex));
end;

end.
```


Summary

This chapter demonstrated techniques for accessing system information from within your Delphi programs. It focused on the proper usage of the ToolHelp32 functions provided by Windows 95/98 and the PSAPI functions found on Windows NT. You learned how to use a few Win32 API functions to obtain other types of system information, including memory information, environment variables, and version information. Additionally, you learned how to incorporate the `TListView`, `TImageList`, `THeaderListbox`, and `TMemView` custom components into your applications. The next chapter, “Porting to Delphi 5,” discusses migrating your applications from previous versions of Delphi.