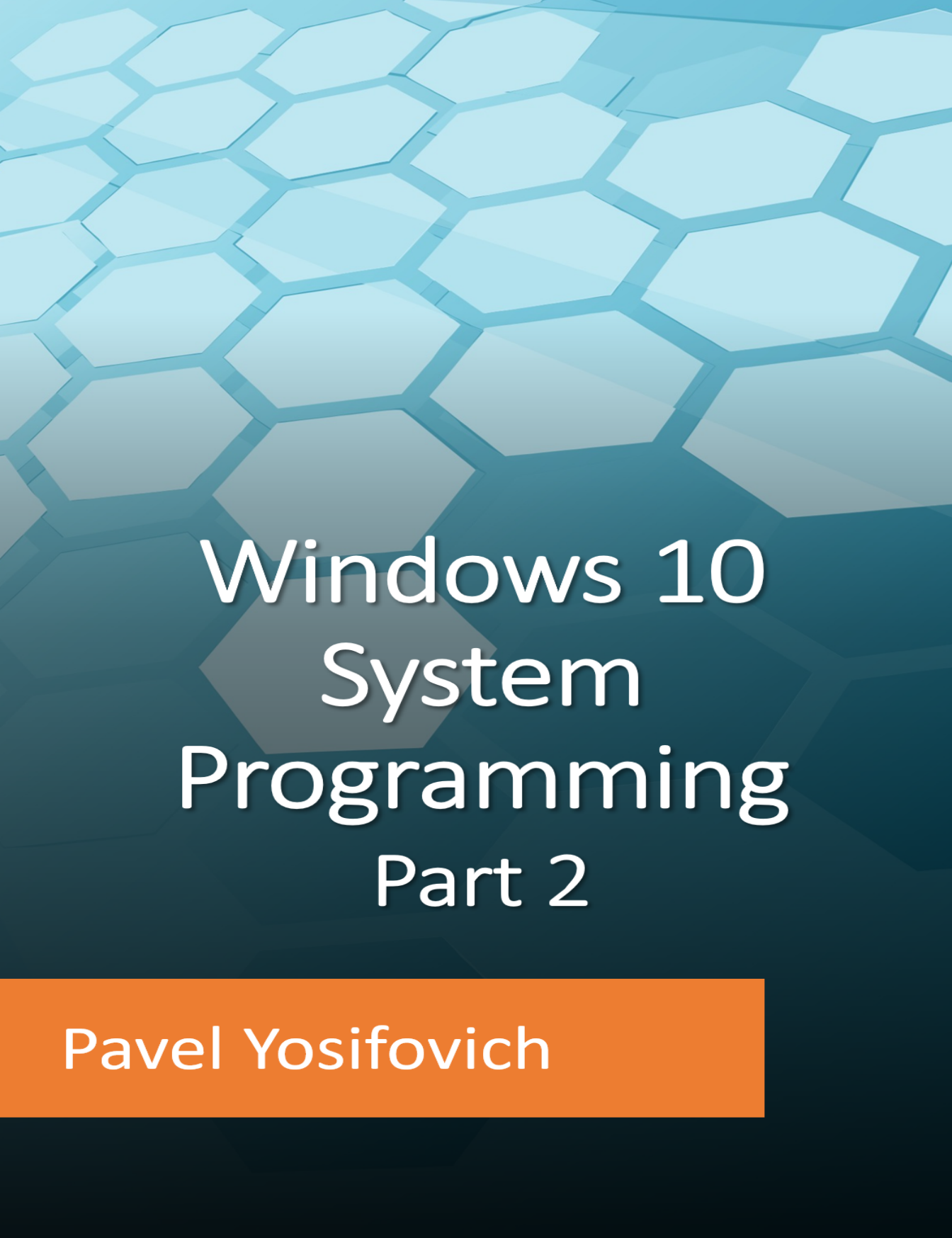

Windows 10
System
Programming
Part 2
Pavel Yosifovich

Windows 10 System Programming, Part 2

Pavel Yosifovich

This book is available at https://leanpub.com/windows10systemprogrammingpart2

This version was published on 2025-10-19

Contents

Introduction

The term *System Programming* refers to programming close to an operating system level. *Windows 10 System Programming* provides guidance for system programmers targeting modern Windows systems, from Windows 7 up to the latest Windows 10 versions.

The book uses the documented Windows *Application Programming Interface* (API) to leverage system-level facilities, including processes, threads, synchronization primitives, virtual memory, I/O, security and more. The book is presented in two parts, due to the sheer size of the Windows API and the Windows system facilities breadth. You're holding in your hands (or your screen of choice) part 2.

Who Should Read This Book

The book is intended for software developers that target the Windows platform, and need to have a level of control not achievable by higher-level frameworks and libraries. The book uses C and C++ for code examples, as the Windows API is mostly C-based. C++ is used where it makes sense, where its advantages are obvious in terms of maintenance, clarity, resource management, or any combination of the above. The book does not use non-trivial C++ constructs, such as template metaprogramming. This book is not about C++, it's about Windows.

That said, other languages can be used to target the Windows API through their specialized interoperability mechanisms. For example, .NET languages (C#, VB, F#, etc.) can use *Platform Invoke* (P/Invoke) to make calls to the Windows API. Other languages, such as Python, Rust, Java, and many others have their own equivalent facilities.

What You Should Know to Use This Book

Readers should be very comfortable with the C programming language, especially with pointers, structures, and the C standard library, as these occur very frequently in the Windows APIs. Basic C++ knowledge is highly recommended, although it is possible to traverse the book with C proficiency only.

Sample Code

All the sample code from the book is freely available in the book's Github repository at https://github.com/zodiacon/Win10SysProgBookSamples. Updates to the code samples will be pushed to this repository. It's recommended readers clone the repository on their local machine, so it's easy to experiment with the code directly.

All code samples have been compiled with Visual Studio 2019. It's possible to compile most code samples with earlier versions of Visual Studio if desired. There might be few features of the latest C++ standards that may not be supported in earlier versions, but these should be easy to fix.

Happy reading!

Pavel Yosifovich
September 2021

Chapter 13: Working With Memory

In chapter 12, we looked at the basics of virtual and physical memory. In this chapter, we'll discuss the various APIs available to developers for managing memory. Some APIs are better to use for large allocations, while others are more suited to managing small allocations. After you complete this chapter, you should have a good understanding of the various APIs and their capabilities, allowing you to choose the right tool for the job where memory is involved.

In this chapter:

- **Memory APIs**
- **The `VirtualAlloc*` Functions**
- **Reserving and Committing Memory**
- **Working Sets**
- **Heaps**
- **Other `Virtual` Functions**
- **Writing and Reading to/from Other Processes**
- **Large Pages**
- **Address Windowing Extensions**
- **NUMA**
- **The `VirtualAlloc2` Function**

Memory APIs

Windows provides several sets of APIs to work with memory. Figure 13-1 shows the available sets and their dependency relationship.

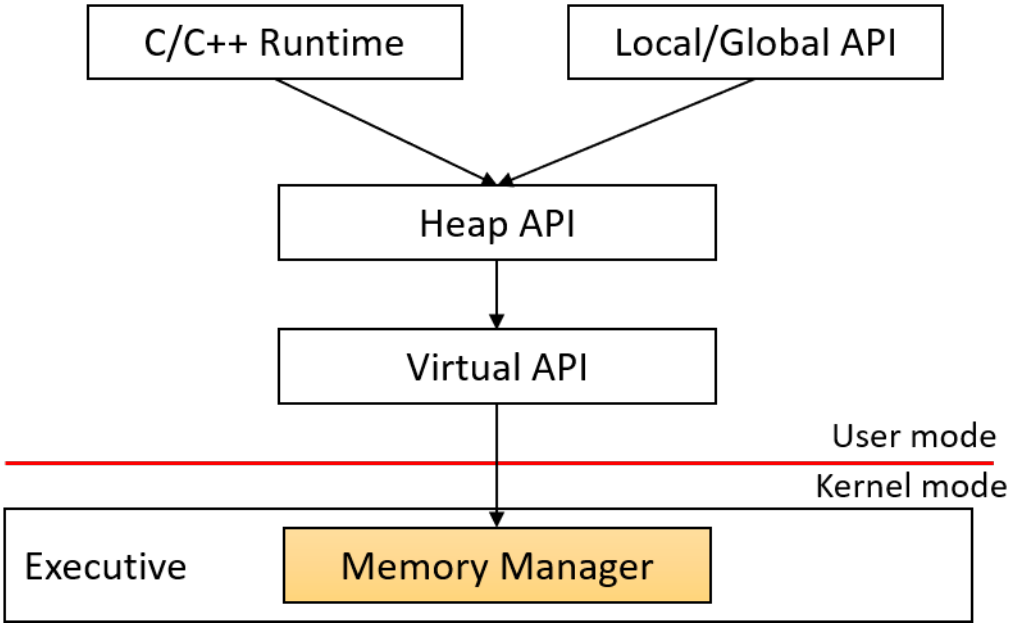


Figure 13-1: Windows user-mode APIs

We'll look at the APIs from the lowest level to the highest. Each API set has its strengths and shortcomings.

The `VirtualAlloc*` Functions

The lowest layer - the `Virtual` API is the closest to the memory manager, which has several implications:

- It's the most powerful API, providing practically everything that can be done with virtual memory.
- It always works in units of pages and on page boundaries.
- It's used by higher-level APIs, as we'll see throughout this chapter.

The most fundamental function that allows reserving and/or committing memory is `VirtualAlloc`:

```
LPVOID VirtualAlloc(
    _In_opt_ LPVOID lpAddress,
    _In_ SIZE_T dwSize,
    _In_ DWORD flAllocationType,
    _In_ DWORD flProtect);
```

An extended function, `VirtualAllocEx`, works on a potentially different process:

```
LPVOID VirtualAllocEx(
    _In_ HANDLE hProcess,
    _In_opt_ LPVOID lpAddress,
    _In_ SIZE_T dwSize,
    _In_ DWORD flAllocationType,
    _In_ DWORD flProtect);
```

`VirtualAllocEx` is identical to `VirtualAlloc` except for the process handle parameter, that must have the `PROCESS_VM_OPERATION` access mask.

`VirtualAlloc(Ex)` cannot be called from a UWP process. Windows 10 added a variant of `VirtualAlloc` that can be called from a UWP process:

```
PVOID VirtualAllocFromApp(
    _In_opt_ PVOID BaseAddress,
    _In_ SIZE_T Size,
    _In_ ULONG AllocationType,
    _In_ ULONG Protection);
```

To make things simpler for UWP processes, `VirtualAlloc` is defined inline and calls `VirtualAllocFromApp`, so technically you can call `VirtualAlloc` from a UWP process.

There is another `VirtualAlloc` variant introduced in Windows 10 version 1803 called `VirtualAlloc2`. It's dealt with in its own section. There is yet another `VirtualAlloc` variant (`VirtualAllocExNuma`) that is used specifically with *Non-Uniform Memory Architecture* (NUMA). We'll discuss NUMA in its own section as well.

We'll start by describing the basic `VirtualAlloc` function upon which all the rest are built. `VirtualAlloc`'s main purpose is to reserve and/or commit a block of memory.

The first parameter to `VirtualAlloc` is an optional pointer where the reservation/committing should take place. If it's a new allocation, `NULL` is typically passed-in, indicating that the memory manager should find some free address. If the region is already reserved, and a commitment inside the region is needed, then `lpAddress` indicates where the committing should start. In any case, the address is rounded down to the nearest page. For new reservations, it's rounded down to the allocation granularity.

> Allocation granularity is currently 64 KB on all Windows architectures and versions. You can always get the value dynamically by calling `GetSystemInfo`.

`dwSize` is the size of the block to reserve/commit. If `lpAddress` is `NULL`, the size is rounded up to the nearest page boundary. For example, 1 KB is rounded to 4 KB, 50 KB is rounded to 52 KB. If `lpAddress` is not `NULL`, then all pages in the range of `lpAddress` to `lpAddress+dwSize-1` are included.

`flAllocationType` indicates the type of operation to perform. The most common flags are `MEM_RESERVE` and `MEM_COMMIT`. With `MEM_RESERVE`, the region is reserved, although the function fails if `lpAddress` specifies an already reserved region.

`MEM_COMMIT` commits a region (or part of a region) previously reserved. This means `lpAddress` cannot be `NULL` in this case. However, it is possible to reserve and commit memory at the same time by combining both flags. For example, the following code reserves and commits 128 KB of memory:

```
void* p = ::VirtualAlloc(nullptr, 128 << 10, MEM_COMMIT | MEM_RESERVE,
    PAGE_READWRITE);
if(!p) {
    // some error occurred
}
```

A `VirtualAlloc` Bug

Technically, you can commit and reserve memory at the same time by using `MEM_COMMIT` alone. Strictly speaking, this is incorrect. The reason this works goes back to a bug in the API that allowed it. Unfortunately, many developers (knowingly or not) abused this bug, and so Microsoft decided not to fix it so that existing code would not break. You should always use both flags if reserving and committing at the same time.

Any committed pages are guaranteed to be filled with zeros. The reason has to do with a security requirement stating that a process can never see any memory belonging to another process, even if that process no longer exists. To make it explicit, the memory is always zeroed out.

This is not the case with functions such as `malloc` and similar. The reason will be clear later in this chapter.

Reserving a region of memory that is already reserved is an error. On the other hand, committing memory that is already committed succeeds implicitly.

The last parameter to `VirtualAlloc` is the page protection to set for the reserved/committed memory (see chapter 12 in part 1 for more on protection flags). For committed memory, it's the page protection to set. For reserved memory, this sets the initial protection (`AllocationProtect` member in `MEMORY_BASIC_INFORMATION`), although it can change when memory is later committed. The protection flag has no effect on reserved memory, since reserved memory is inaccessible. Still, a valid value must be supplied even in this case.

The return value of `VirtualAlloc` is the base address for the operation if successful, or `NULL` otherwise. If `lpAddress` is not `NULL`, the returned value may or may not equal `lpAddress`, depending on its page or allocation granularity alignment (as described earlier).

There are other possible flags to `VirtualAlloc` except `MEM_RESERVE` and `MEM_COMMIT`:

- `MEM_RESET` is a flag, that if used, must be the only one. It indicates to the memory manager that the committed memory in the range is no longer needed, and so the memory manager should not

bother writing it to a page file. The committed memory cannot be backed by a mapped file, only by a page file. Note that this is not the same as decommitting the memory; the memory is still committed and can be used later (see next flag).

- `MEM_RESET_UNDO` is the opposite of `MEM_RESET`, stating that the committed memory region is of interest again. The values in the range are not necessarily zero, since the memory manager may or may not have reused the mapped physical pages.
- `MEM_LARGE_PAGES` indicates the operation should use large pages rather than small pages. We'll discuss this option in the "Large Pages" section, later in this chapter.
- `MEM_PHYSICAL` is a flag that can only be specified with `MEM_RESERVE`, for use with *Address Windowing Extensions* (AWE), described later in this chapter.
- `MEM_TOP_DOWN` is an advisory flag to the memory manager to prefer high addresses rather than low ones.
- `MEM_WRITE_WATCH` is a flag that must be specified with `MEM_RESERVE`. This flag indicates the system should track memory writes to this region (once committed, of course). This is described further in the "Memory Tracking" section.

Decommitting / Releasing Memory

`VirtualAlloc` must have an opposite function that can de-commit and/or release (the opposite of reserve) a block of memory. This is the role of `VirtualFree` and `VirtualFreeEx`:

```
BOOL VirtualFree(
    _in_ LPVOID lpAddress,
    _In_ SIZE_T dwSize,
    _In_ DWORD dwFreeType);

BOOL VirtualFreeEx(
    _In_ HANDLE hProcess,
    _In_ LPVOID lpAddress,
    _In_ SIZE_T dwSize,
    _In_ DWORD dwFreeType);
```

`VirtualFreeEx` is an extended version of `VirtualFree` that performs the requested operation in the process specified by `hProcess`, which must have the `PROCESS_VM_OPERATION` access mask (just like `VirtualAllocEx`). Only two flags are supported by the `dwFreeType` parameter - `MEM_DECOMMIT` and `MEM_RELEASE` - one of which (and only one) must be specified.

`MEM_DECOMMIT` decommits the pages that span `lpAddress` to `lpAddress+dwSize-1`, returning the memory region to the reserved state. `MEM_RELEASE` indicates the region should be freed completely. `lpAddress` must be the base address of the region originally reserved, and `dwSize` must be zero. If any memory in the region is committed, it's first decommitted and then the entire region is released (pages become free).

Reserving and Committing Memory

Using the `VirtualAlloc` function to reserve and commit memory is a good idea when large allocations are needed, since this function works on page granularity. For small allocations, using `VirtualAlloc` is too wasteful, since every new allocation would be on a new page. For small allocations, it's better to use the heap functions (described later in this chapter).

Committed Memory and RAM

Committing memory does not mean that RAM is immediately allocated for that memory. Committing memory increases the total system commit, meaning it guarantees that the committed memory will be available when accessed. Once a page is accessed, the system provides the page in RAM and the access can go through. Providing this page in RAM may be at the expense of another page in RAM that would be pushed to disk if the system is low on memory. Regardless, this process is transparent to the application.

Suppose you want to create an application that works similarly to Microsoft *Excel*, where a grid of cells is available for data entry of some sort. Let's further suppose that you want the user to have a large grid at her disposal, say 1024 by 1024 cells, and each cell can contain a 1 KB of data. How would you manage the cells in such an application?

One way is to use some linked list or map implementation, where each element is a 1 KB chunk of memory. When the user accesses a cell, the element is retrieved based on the managed data structure and used. Since every cell has the same size, another alternative might be to allocate a large enough memory chunk and then access a particular cell with a quick calculation. Here is an example:

```
int cellSize = 1 << 10;      // 1 KB
int maxx = 1 << 10, maxy = 1 << 10; // 1024 x 1024 cells
void* data = malloc(maxx * maxy * cellSize);

// locate cell (x,y) address
void* pCell = (BYTE*)data + (y * maxx + x) * cellSize;
// perform access to pCell...
```

This works, and locating a cell is very fast. The problem is that 1 GB of memory is committed upfront. This is wasteful because the user is unlikely to use all the available cells. Also, if we later decide to allow more cells to be used, the committed memory would have to be larger.

What we want is a way to continue locating cells very fast, but not allocate the cell data unless it's being used. This is where reserving memory and committing in chunks can help solve this issue. We start with reserving a 1 GB of address space:

```
void* data = ::VirtualAlloc(nullptr, maxx * maxy * cellSize,
    MEM_RESERVE, PAGE_READWRITE);
```

The reserving operation is very cheap - the commit size of the system is not modified. Whenever an access is required for a cell, we can compute its address within the block as before, and then commit the required cell:

```
// commit page for cell (x, y)
void* pCell = (BYTE*)data + (y * maxx + x) * cellSize;
::VirtualAlloc(pCell, cellSize, MEM_COMMIT, PAGE_READWRITE);
// access cell...
```

The code calculates the cell's address like before, but because the memory is initially reserved, there is nothing there. Accessing that memory in any way causes an access violation exception. By using `VirtualAlloc` with `MEM_COMMIT`, the page where the cell resides is committed, making it "real". Accessing this memory must now succeed.

> `VirtualAlloc` always works with pages, so the above code commits 4 cells (remember each cell is 1 KB in size), not just one. There is no way around that when using the `Virtual*` functions.

Every time a cell needs to be used, the same code runs, committing that cell (and 3 adjacent cells), so that there are accessible. The consumption of memory is only for those cells that are used (and the adjacent cells even if they are not used). This scheme allows us to use a very large number of potential cells without wasting more memory. For example, we can increase the maximum grid size to 2048 by 2048. The only change is the amount of reserved memory, to keep the address space contiguous.

The main downside of this approach is that large portions of the address space are taken. In 64-bit processes (where the address space is 128 TB), this is not an issue. In 32-bit processes, this might fail. For example, using a 2048 by 2048 grid with 1 KB of memory per cell requires 4 GB of address space, which is beyond the capabilities of a 32-bit process (even on WOW64 with the `LARGEADDRESSAWARE` set, since the address space must also contain DLLs, thread stacks, etc.). The address range must also be contiguous, which even with a smaller size could be a problem to get in a 32-bit process.

Committing an already committed memory is not an issue, but it does incur a system call, that perhaps could be avoided if the cell memory in question is already committed. How can that be done?

One way is to use the `VirtualQuery` function described in chapter 12 to query the memory region and decide whether to commit memory or not. But that requires a system call in itself, so it's actually worse than just committing before any access. The alternative is to access the memory "blindly" - if it's already committed, it just works; if not, an exception is raised, which can be caught and handled by committing the required memory. Here is the basic idea in code:

```
void DoWork(void* data, int x, int y) {
    // in some function
    void* pCell = (BYTE*)data + (y * maxx + x) * cellSize;
    __try
        // access the cell memory
        ::strcpy((char*)pCell, "some text data");
        // if we get here, all is well
    }
    __except(FixMemory(pCell, GetExceptionCode())) {
        // no code needed here
    }
}

int FixMemory(void* p, DWORD code) {
    if(code == EXCEPTION_ACCESS_VIOLATION) {
        // we can fix it by comitting the memory
        ::VirtualAlloc(p, cellSize, MEM_COMMIT, PAGE_READWRITE);
        // tell the CPU to try again
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    // some other exception, look elsewhere for a handler
    return EXCEPTION_CONTINUE_SEARCH;
}
```

If an exception is raised in the `strcpy` call, the `__except` expression is evaluated by calling the `FixMemory` function with the address in question and the exception code. The purpose of this function is to return one of three possible values indicating to the exception handling mechanism what to do next. If the exception code is an access violation, then we can do something about it and commit the required memory before returning `EXCEPTION_CONTINUE_EXECUTION`, indicating the processor should try the original instruction again. If it's some other exception, we don't handle it and return `EXCEPTION_CONTINUE_SEARCH` to continue searching up the call stack for a handler.

> The other possible return value is `EXCEPTION_EXECUTE_HANDLER`. Exception handling is detailed in chapter 23.

The *Micro Excel* Application

The *Micro Excel* application demonstrates the above technique. Running it shows the dialog in figure 13-2.

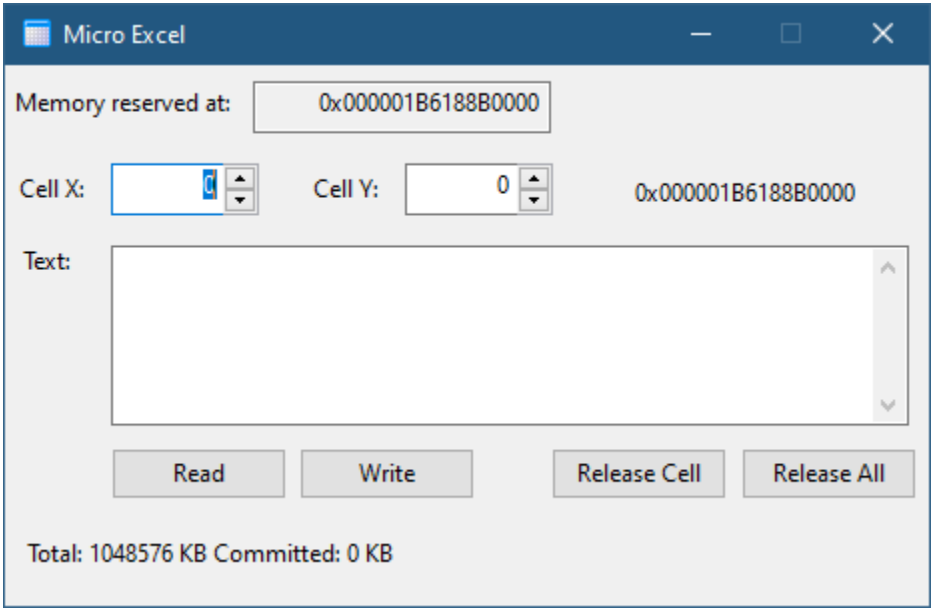


Figure 13-2: The *Micro Excel* application

The application reserves a 1 GB memory range, starting at the address shown at the top. The *Cell X* and *Cell Y* edit boxes allow selecting a cell in either direction (0-1023). By typing something in the large edit box and clicking *Write*, the text is written to the requested cell. If the memory is not committed (which must be the case the first time), it's committed by handling an access violation exception. After adding one string to cell (0,0), the application window looks like figure 13-3.

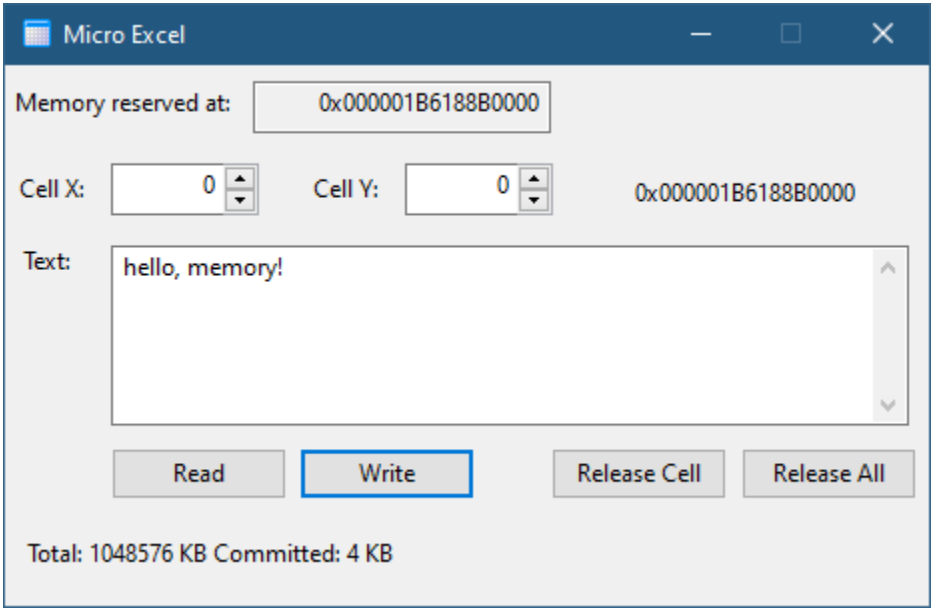


Figure 13-3: *Micro Excel* application with one allocation

The bottom text indicates 4 KB have been committed, which is what we expect, as a single page is required for the 1 KB cell. If we set cell X to 1 and write something, what would be the committed size? It would remain the same, because the first committed page covers 4 cells (figure 13-4).

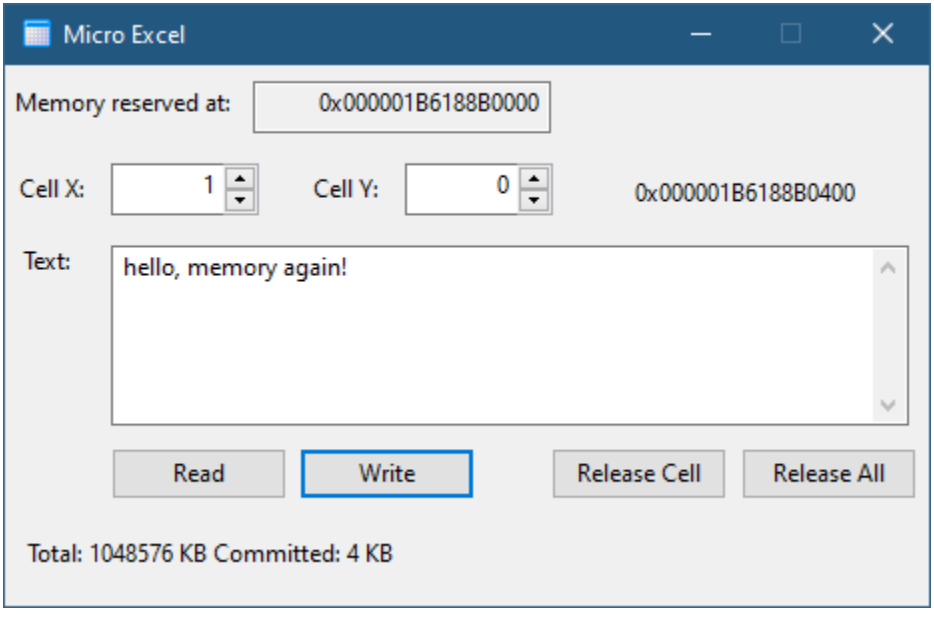


Figure 13-4: Writing to cell (1,0)

What would happen if you write something to cell (0,1)? Try it and find out! You can read back from any

cell by using the *Read* button. If the cell is not committed, an error is returned.

It's interesting to see the allocated memory region with *VMMap*. Each time a new page is committed, it "punches a hole" in the large reserved region. Figure 13-5 shows the memory region used by the application with several "punched holes" of committed memory.

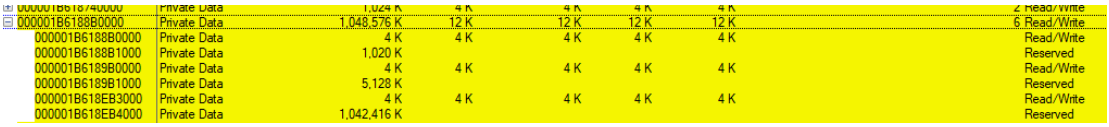


Figure 13-5: *VMMap* showing the committed regions inside the large reserved region

The application is built as a standard WTL dialog-based application. The `CMainDlg` class holds several data members related to the managed memory:

```
class CMainDlg : public CDialogImpl<CMainDlg> {
//...
    const int CellSize = 1024, SizeX = 1024, SizeY = 1024;
    const size_t TotalSize = CellSize * SizeX * SizeY;

    void* m_Address{ nullptr };
};
```

The various sizes are declared as constants, but it wouldn't be much different if these were dynamically set. `OnInitDialog` calls `AllocateRegion` to reserve the initial region of memory:

```
bool CMainDlg::AllocateRegion() {
    m_Address = ::VirtualAlloc(nullptr, TotalSize,
        MEM_RESERVE, PAGE_READWRITE);
    if (!m_Address) {
        AtlMessageBox(nullptr,
            L"Available address space is not large enough",
            IDR_MAINFRAME, MB_ICONERROR);
        EndDialog(IDCANCEL);
        return false;
    }

    // update the UI
    CString addr;
    addr.Format(L"0x%p", m_Address);
    SetDlgItemText(IDC_ADDRESS, addr);
    SetDlgItemText(IDC_CELLADDR, addr);

    return true;
}
```

Once the user clicks any of the buttons, the address of the cell needs to be retrieved. This is accomplished with `GetCell`:

```
void* CMainDlg::GetCell(int& x, int& y, bool reportError) const {
    // get indices from UI
    x = GetDlgItemInt(IDC_CELLX);
    y = GetDlgItemInt(IDC_CELLY);
    // check range validity
    if (x < 0 || x >= SizeX || y < 0 || y >= SizeY) {
        if(reportError)
            AtlMessageBox(*this, L"Indices out of range",
                IDR_MAINFRAME, MB_ICONEXCLAMATION);
        return nullptr;
    }
    return (BYTE*)m_Address + CellSize * ((size_t)x + SizeX * y);
}
```

The interesting code is in the *Write* button handler. First, the cell address is retrieved:

```
LRESULT CMainDlg::OnWrite(WORD, WORD, HWND, BOOL&) {
    int x, y;
    auto p = GetCell(x, y);
    if(!p)
        return 0;
```

Next, the edit box context is read and written to the memory block. If an exception occurs, the helper `FixMemory` is called:

```
WCHAR text[512];
GetDlgItemText(IDC_TEXT, text, _countof(text));

__try {
    ::wcscpy_s((WCHAR*)p, CellSize / sizeof(WCHAR), text);
}
__except (FixMemory(p, GetExceptionCode())) {
    // nothing to do: this code is never reached
}
```

`FixMemory` will attempt to correct the error if indeed this was an access violation:

```
int CMainDlg::FixMemory(void* address, DWORD exceptionCode) {
    if (exceptionCode == EXCEPTION_ACCESS_VIOLATION) {
        // commit the cell
        ::VirtualAlloc(address, CellSize, MEM_COMMIT, PAGE_READWRITE);
        // tell the CPU to try again
        return EXCEPTION_CONTINUE_EXECUTION;
    }

    // some other error, continue to search a handler up the call stack
    return EXCEPTION_CONTINUE_SEARCH;
}
```

> Technically, the `VirtualAlloc` call inside `FixMemory` could fail if the system is at its maximum commit limit. In that case, `VirtualAlloc` returns `NULL`, and the return value from `FixMemory` should be `EXCEPTION_CONTINUE_SEARCH`.

The *Read* button handler is similar, except that no committing is attempted if the cell is not committed, and an error is displayed instead:

```
LRESULT CMainDlg::OnRead(WORD, WORD, HWND, BOOL&) {
    int x, y;
    auto p = GetCell(x, y);
    if(!p)
        return 0;

    WCHAR text[512];
    __try {
        ::wcscpy_s(text, _countof(text), (PCWSTR)p);
        SetDlgItemText(IDC_TEXT, text);
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        AtlMessageBox(nullptr, L"Cell memory is not committed",
            IDR_MAINFRAME, MB_ICONWARNING);
    }
    return 0;
}
```

The *Release* and *Release All* buttons decommit a cell and release the entire memory block, respectively:

```
LRESULT CMainDlg::OnRelease(WORD, WORD, HWND, BOOL&) {
    int x, y;
    auto p = GetCell(x, y);
    if (p) {
        ::VirtualFree(p, CellSize, MEM_DECOMMIT);
    }
    return 0;
}

LRESULT CMainDlg::OnReleaseAll(WORD, WORD wID, HWND, BOOL&) {
    ::VirtualFree(m_Address, 0, MEM_RELEASE);
    // allocate a new reserved region
    AllocateRegion();

    return 0;
}
```

Finally, the bottom output indicating the amount of committed memory is displayed using a timer that uses the `VirtualQuery` API from chapter 12 to walk the entire memory region and count the size of the committed memory:

```
LRESULT CMainDlg::OnTimer(UINT, WPARAM id, LPARAM, BOOL&) {
    if (id == 1) {
        MEMORY_BASIC_INFORMATION mbi;
        auto p = (BYTE*)m_Address;
        size_t committed = 0;
        while (p < (BYTE*)m_Address + TotalSize) {
            ::VirtualQuery(p, &mbi, sizeof(mbi));
            if (mbi.State == MEM_COMMIT)
                committed += mbi.RegionSize;
            p += mbi.RegionSize;
        }
        CString text;
        text.Format(L"Total: %llu KB Committed: %llu KB",
            TotalSize >> 10, committed >> 10);
        SetDlgItemText(IDC_STATS, text);
    }
    return 0;
}
```

Working Sets

The term *Working Set* represents memory accessible without incurring a page fault. Naturally, a process would like all of its committed memory be in its working set. The memory manager must balance the needs of a process with the needs of all other processes. Memory that was not accessed for a long time might be removed from a process' working set. This does not mean it's automatically discarded - the memory manager has elaborate algorithms to keep physical pages that used to be part of a process' working set in RAM longer than might be necessary, so that if the process in question decided to access that memory, it can be immediately faulted into the working set (this is known as a *soft page fault*).

> The details of the way the memory manager manages physical memory is beyond the scope of this book. Interested readers should consult chapter 5 in the *Windows Internals, 7th ed. Part 1* book.

The current and peak woking sets of a process can be obtained by calling `GetProcessMemoryInfo`, described in chapter 12 (part 1), repeated here for convenience:

```
BOOL GetProcessMemoryInfo(
    HANDLE Process,
    PPROCESS_MEMORY_COUNTERS ppsmemCounters,
    DWORD cb);
```

The members `WorkingSetSize` and `PeakWorkingSetSize` of `PROCESS_MEMORY_COUNTERS` give the current and peak working set of the specified process. Other members of this (and the extended) structure provide other useful metrics. Refer to chapter 12 for the full details.

A process has a minimum and maximum working set. These limits are soft by default, so that a process can consume more RAM than its maximum working set if memory is abundant, and can use less RAM than its minimum working set if memory is scarce. You can query for these limits with `GetProcessWorkingSetSize`:

```
BOOL GetProcessWorkingSetSize(
    _In_  HANDLE hProcess,
    _Out_ PSIZE_T lpMinimumWorkingSetSize,  // bytes
    _Out_ PSIZE_T lpMaximumWorkingSetSize); // bytes
```

The process handle must have the `PROCESS_QUERY_INFORMATION` or `PROCESS_QUERY_LIMITED_INFORMATION` access mask. The default values are a minimum of 50 pages (200 KB), and a maximum of 345 pages (1380 KB), but these limits can be altered by calling `SetProcessWorkingSetSize`:

```
BOOL SetProcessWorkingSetSize(
    _In_ HANDLE hProcess,
    _In_ SIZE_T dwMinimumWorkingSetSize,    // bytes
    _In_ SIZE_T dwMaximumWorkingSetSize);   // bytes
```

The process handle must have the `PROCESS_SET_QUOTA` access mask for the operation to have a chance to succeed. If the minimum or the maximum working set values is higher than the current maximum working set, the caller's token must have the `SE_INC_WORKING_SET_NAME` privilege. This is not normally an issue, as all users have this privilege.

The minimum value for the minimum working set size is 20 pages (80 KB), and the minimum value for the maximum working set size is 13 pages (52 KB). The maximum for the maximum working set size is 512 pages less than the available memory. Specifying zero for any of the values is an error.

One special case is specifying `(SIZE_T)-1` for both values. This causes the system to remove as many pages as possible from the process' working set. The same can be accomplished with the `EmptyWorkingSet` function:

```
BOOL WINAPI EmptyWorkingSet(_In_ HANDLE hProcess);
```

The process handle must have the `PROCESS_SET_QUOTA` access mask and either `PROCESS_QUERY_INFORMATION` or `PROCESS_QUERY_LIMITED_INFORMATION`.

As mentioned, the minimum and maximum working set sizes are soft limits by default. This can be changed with an extended version of `SetProcessWorkingSetSize`:

```
BOOL SetProcessWorkingSetSizeEx(
    _In_ HANDLE hProcess,
    _In_ SIZE_T dwMinimumWorkingSetSize,
    _In_ SIZE_T dwMaximumWorkingSetSize,
    _In_ DWORD Flags);
```

The extra flags parameter can be a combination of the values listed in table 13-1.

Table 13-1: Flags for `SetProcessWorkingSetSizeEx`

Flag	Description
`QUOTA_LIMITS_HARDWS_MIN_ENABLE` (1)	Hard limit on the minimum working set
`QUOTA_LIMITS_HARDWS_MIN_DISABLE` (2)	Soft limit on the minimum working set
`QUOTA_LIMITS_HARDWS_MAX_ENABLE` (4)	Hard limit on the maximum working set
`QUOTA_LIMITS_HARDWS_MAX_DISABLE` (8)	Soft limit on the maximum working set

If a process is configured to use a maximum working set hard limit, any extra committed memory the process might like to have as part of its working set causes other pages to be removed from its working

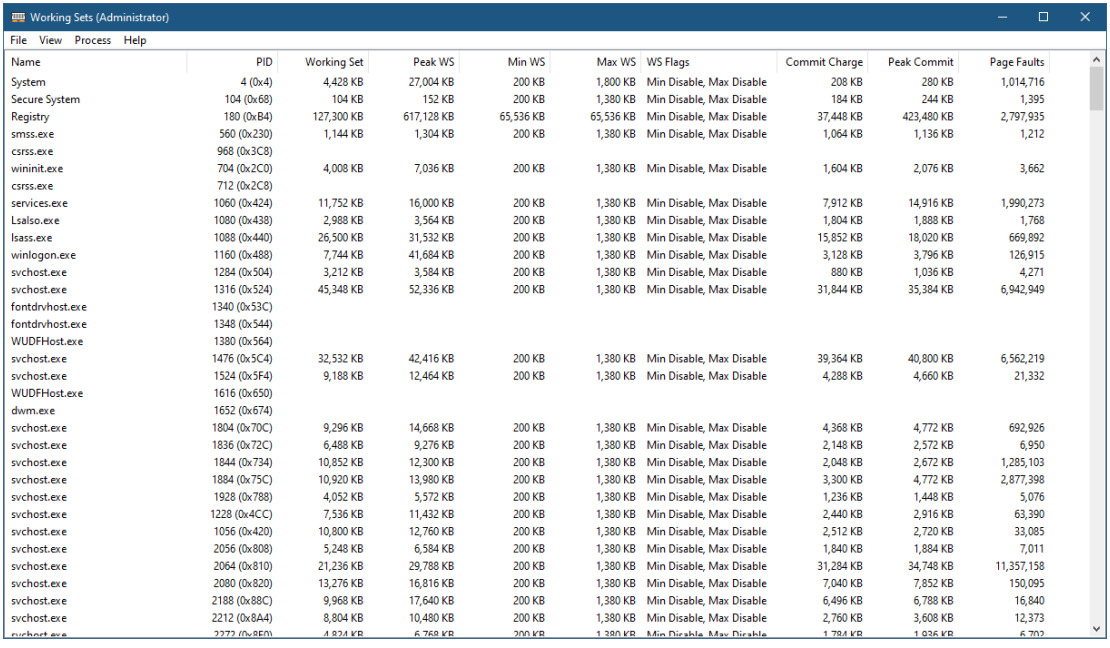
set, essentially making page faults against itself. Specifying zero for the flags does not change the limit flags, essentially making the call equivalent to `SetProcessWorkingSetSize`. The opposite function for getting the current limits and the flags exists as well:

```
BOOL GetProcessWorkingSetSizeEx(
    _In_ HANDLE hProcess,
    _Out_ PSIZE_T lpMinimumWorkingSetSize,
    _Out_ PSIZE_T lpMaximumWorkingSetSize,
    _Out_ PDWORD Flags);
```

The *Working Sets* Application

The *Working Sets* application shows all processes with memory-related counters, such as the working set size, the peak working set size, the minimum and maximum, and more. It's based on the APIs discussed in the previous section. It's interesting (and sometime necessary) to get a sense of how various processes use memory.

When the application is first launched, many processes do not show any memory counters. This is because a handle could not be successfully open to them. Selecting the *View* / *Accessible Processes Only* leaves only accessible processes in the display. Alternatively, selecting *File* / *Run as Administrator* (or launching the application with admin rights) shows many more processes as accessible. Figure 13-6 shows what this might look like.



Name	PID	Working Set	Peak WS	Min WS	Max WS	WS Flags	Commit Charge	Peak Commit	Page Faults
System	4 (0x4)	4,428 KB	27,004 KB	200 KB	1,800 KB	Min Disable, Max Disable	208 KB	280 KB	1,014,716
Secure System	104 (0x68)	104 KB	152 KB	200 KB	1,380 KB	Min Disable, Max Disable	184 KB	244 KB	1,395
Registry	180 (0xB4)	127,300 KB	617,128 KB	65,536 KB	65,536 KB	Min Disable, Max Disable	37,448 KB	423,480 KB	2,797,935
smss.exe	560 (0x230)	1,144 KB	1,304 KB	200 KB	1,380 KB	Min Disable, Max Disable	1,064 KB	1,136 KB	1,212
csrss.exe	968 (0x3C8)								
wininit.exe	704 (0x2C0)	4,008 KB	7,036 KB	200 KB	1,380 KB	Min Disable, Max Disable	1,604 KB	2,076 KB	3,662
csrss.exe	712 (0x2C8)								
services.exe	1060 (0x424)	11,752 KB	16,000 KB	200 KB	1,380 KB	Min Disable, Max Disable	7,912 KB	14,916 KB	1,990,273
Lsalso.exe	1080 (0x438)	2,988 KB	3,564 KB	200 KB	1,380 KB	Min Disable, Max Disable	1,804 KB	1,888 KB	1,768
lsass.exe	1088 (0x440)	26,500 KB	31,532 KB	200 KB	1,380 KB	Min Disable, Max Disable	15,852 KB	18,020 KB	669,892
winlogon.exe	1160 (0x488)	7,744 KB	41,684 KB	200 KB	1,380 KB	Min Disable, Max Disable	3,128 KB	3,796 KB	126,915
svchost.exe	1284 (0x504)	3,212 KB	3,584 KB	200 KB	1,380 KB	Min Disable, Max Disable	880 KB	1,036 KB	4,271
svchost.exe	1316 (0x524)	45,348 KB	52,336 KB	200 KB	1,380 KB	Min Disable, Max Disable	31,844 KB	35,384 KB	6,942,949
fontdrvhost.exe	1340 (0x53C)								
fontdrvhost.exe	1348 (0x544)								
WUDFHost.exe	1380 (0x564)								
svchost.exe	1476 (0x5C4)	32,532 KB	42,416 KB	200 KB	1,380 KB	Min Disable, Max Disable	39,364 KB	40,800 KB	6,562,219
svchost.exe	1524 (0x5F4)	9,188 KB	12,464 KB	200 KB	1,380 KB	Min Disable, Max Disable	4,288 KB	4,660 KB	21,332
WUDFHost.exe	1616 (0x650)								
dwm.exe	1652 (0x674)								
svchost.exe	1804 (0x70C)	9,296 KB	14,668 KB	200 KB	1,380 KB	Min Disable, Max Disable	4,368 KB	4,772 KB	692,926
svchost.exe	1836 (0x72C)	6,488 KB	9,276 KB	200 KB	1,380 KB	Min Disable, Max Disable	2,148 KB	2,572 KB	6,950
svchost.exe	1844 (0x734)	10,852 KB	12,300 KB	200 KB	1,380 KB	Min Disable, Max Disable	2,048 KB	2,672 KB	1,285,103
svchost.exe	1884 (0x75C)	10,920 KB	13,980 KB	200 KB	1,380 KB	Min Disable, Max Disable	3,300 KB	4,772 KB	2,877,398
svchost.exe	1928 (0x788)	4,052 KB	5,572 KB	200 KB	1,380 KB	Min Disable, Max Disable	1,236 KB	1,448 KB	5,076
svchost.exe	1228 (0x4CC)	7,536 KB	11,432 KB	200 KB	1,380 KB	Min Disable, Max Disable	2,440 KB	2,916 KB	63,390
svchost.exe	1056 (0x420)	10,800 KB	12,760 KB	200 KB	1,380 KB	Min Disable, Max Disable	2,512 KB	2,720 KB	33,085
svchost.exe	2056 (0x808)	5,248 KB	6,584 KB	200 KB	1,380 KB	Min Disable, Max Disable	1,840 KB	1,884 KB	7,011
svchost.exe	2064 (0x810)	21,236 KB	29,788 KB	200 KB	1,380 KB	Min Disable, Max Disable	31,284 KB	34,748 KB	11,357,158
svchost.exe	2080 (0x820)	13,276 KB	16,816 KB	200 KB	1,380 KB	Min Disable, Max Disable	7,040 KB	7,852 KB	150,095
svchost.exe	2188 (0x88C)	9,968 KB	17,640 KB	200 KB	1,380 KB	Min Disable, Max Disable	6,496 KB	6,788 KB	16,840
svchost.exe	2212 (0x8A4)	8,804 KB	10,480 KB	200 KB	1,380 KB	Min Disable, Max Disable	2,760 KB	3,608 KB	12,373
svchost.exe	2272 (0x8E0)	4,824 KB	6,768 KB	200 KB	1,380 KB	Min Disable, Max Disable	1,784 KB	1,936 KB	6,702

Figure 13-6: The *Working Sets* application

The display refreshes automatically every second. You can try working with a certain process and watch its working set and committed memory usage change dynamically. You can sort by any column. Figure

13-7 shows the processes whose peak working set usage is the highest (at least from the processes that can be queried).

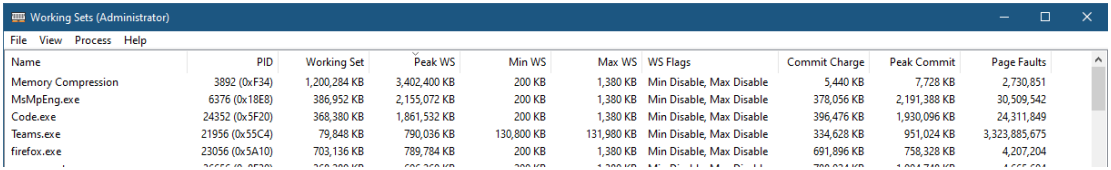


Figure 13-7: *Working Sets* sorted by peak working set

The *Memory Compression* process has the highest peak working set, which is no real surprise, since it holds compressed memory, thus saving physical memory. Next in the list is *Windows Defender*, which consumes way too much memory for an anti-malware software. *Visual Studio Code* is next, and so on.

You can select the *Process* / *Empty Working Set* menu item to force a process to relinquish most of its physical memory usage (at least for a while). For some processes, this will fail, as `PROCESS_SET_QUOTA` is not possible to get for every process.

The application is built as a standard WTL *Single Document Interface* (SDI), which a view that is based on a list view control. The most important function is `CView::Refresh`, called initially and every second when the timer expires. Each process information is stored in the following structure (defined as a nested type in *view.h*):

```
struct ProcessInfo {
    DWORD Id;           // process ID
    CString ImageName;
    SIZE_T MinWorkingSet, MaxWorkingSet;
    DWORD WorkingSetFlags;
    PROCESS_MEMORY_COUNTERS_EX Counters;
    bool CountersAvailable{ false };
};

std::vector<ProcessInfo> m_Items;
```

The first order of business in the `Refresh` method is to start process enumeration:

```
void CView::Refresh() {
    wil::unique_handle hSnapshot(
        ::CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0));
    if (!hSnapshot)
        return;

    PROCESSENTRY32 pe;
    pe.dwSize = sizeof(pe);
    // skip the idle process
```

```
    ::Process32First(hSnapshot.get(), &pe);

    m_Items.clear();
    m_Items.reserve(512);
```

The *Toolhelp* functions introduced in chapter 3 are used. For each process, a `ProcessInfo` object is filled up:

```
while (::Process32Next(hSnapshot.get(), &pe)) {
    // attempt to open a handle to the process
    wil::unique_handle hProcess(
        ::OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION,
        FALSE, pe.th32ProcessID));
    if (!hProcess && m_ShowOnlyAccessibleProcesses)
        continue;

    ProcessInfo pi;
    pi.Id = pe.th32ProcessID;
    pi.ImageName = pe.szExeFile;
```

If a handle cannot be obtained and the "show only accessible processes* option is set, then skip this process. Otherwise, fill in the only two values that are guaranteed to be available for all processes - the process ID and process image name.

Next, if a proper handle could be opened, the memory APIs are invoked to get the information and store it in the `ProcessInfo` instance:

```
if (hProcess) {
    ::GetProcessMemoryInfo(hProcess.get(),
        (PROCESS_MEMORY_COUNTERS*)&pi.Counters,
        sizeof(pi.Counters));
    ::GetProcessWorkingSetSizeEx(hProcess.get(), &pi.MinWorkingSet,
        &pi.MaxWorkingSet, &pi.WorkingSetFlags);
    pi.CountersAvailable = true;
}
```

If a handle is available, the `CountersAvailable` field is set to `true`, indicating the memory information in the structure is valid. Finally, the object is added to the vector and the loop continues for all processes:

```
    m_Items.push_back(pi);
}
```

All that's left to do is sort the items (if needed) and update the list view:

```
    DoSort(GetSortInfo(*this));
    SetItemCountEx(static_cast<int>(m_Items.size()),
        LVSICF_NOSCROLL | LVSICF_NOINVALIDATEALL);
    RedrawItems(GetTopIndex(), GetTopIndex() + GetCountPerPage());
}
```

> I'm not discussing the other functions in the `CView` class, since they don't use any memory-related APIs. The full source code is available for the interested reader to browse through.

To empty the working, `SetProcessWorkingSetSize` is called with both values as `-1`. The `EmptyWorkingSet` API could have been called instead:

```
LRESULT CView::OnEmptyWorkingSet(WORD, WORD, HWND, BOOL&) {
    auto index = GetSelectedIndex();
    ATLASSERT(index >= 0);

    const auto& item = m_Items[index];
    wil::unique_handle hProcess(
        ::OpenProcess(PROCESS_SET_QUOTA, FALSE, item.Id));
    if (!hProcess) {
        AtlMessageBox(*this, L"Failed to open process",
            IDR_MAINFRAME, MB_ICONERROR);
        return 0;
    }

    if (!::SetProcessWorkingSetSize(hProcess.get(),
        (SIZE_T)-1, (SIZE_T)-1)) {
        AtlMessageBox(*this, L"Failed to empty working set",
            IDR_MAINFRAME, MB_ICONERROR);
    }
    return 0;
}
```

Add a menu option to set a minimum and/or maximum working set sizes (perhaps with a dialog box), along with the optional flags available with `SetProcessWorkingSetSizeEx` and do some experimentation on how these values affect running processes.

Heaps

The `VirtualAlloc` set of functions are very powerful, since they are very close to the memory manager. There is a downside, however. These functions only work in page chunks: if you allocate 10 bytes, you get back a page. If you allocate 10 more bytes, you get a *different* page. This is too wasteful for managing small allocations, which are so common in applications. This is exactly where heaps come in.

The *Heap Manager* is a component layered on top of the *Virtual* API, that knows how to manage small allocations efficiently. A *heap*, in this context, is a memory block managed by the heap manager. Every process starts with a single heap, called the *default process heap*. A handle to that heap is obtained with `GetProcessHeap`:

```
HANDLE GetProcessHeap();
```

More heaps can be created, described in the section *Other Heaps*. With a heap in hand, allocating (committing) memory is done with `HeapAlloc`:

```
LPVOID HeapAlloc(
    _In_ HANDLE hHeap,
    _In_ DWORD dwFlags,
    _In_ SIZE_T dwBytes);
```

`HeapAlloc` accepts a handle to a heap, optional flags and the number of bytes of requested memory. The flags can be zero or a combination of the following values:

- `HEAP_ZERO_MEMORY` specifies that the returned memory block should be zeroed out by the function. Otherwise, whatever was in that block remains.
- `HEAP_NO_SERIALIZE` indicates the function should not take the heap's lock. This means the developer provides her own synchronization or guarantees that no concurrent access is made to the heap. This value should not be specified for the default process heap, since the default heap is created with synchronization and it's used with some APIs which do not expect unsynchronized access. For application-created heaps, this flag can be specified when a heap is created (`HeapCreate`, see later), so that specifying this value for every allocation is not required.
- `HEAP_GENERATE_EXCEPTIONS` indicates the function should report failure by raising an SEH exception (`STATUS_NO_MEMORY`), rather than returning `NULL`. If such behavior is desired for all heap operations, this flag can be specified when the heap is created with `HeapCreate`.

Here is an example of allocating some data from the default process heap:

```
struct MyData {
    //...
};

MyData* pData = (MyData*)::HeapAlloc(
    ::GetProcessHeap(), 0, sizeof(MyData));
if(pData == nullptr) {
    // handle failure
}
```

If an allocated block needs to be increased or decreased in size while preserving the existing data, `HeapReAlloc` can be used:

```
LPVOID HeapReAlloc(
    _Inout_ HANDLE hHeap,
    _In_ DWORD dwFlags,
    _Frees_ptr_opt_ LPVOID lpMem,
    _In_ SIZE_T dwBytes);
```

`lpMem` is the existing address, with `dwBytes` being the new requested size. The function returns the address of the new block, which may be the same as the original address if the new size is smaller than the original or the there is enough space to resize the block without moving it. If the new size is larger but does not fit in the existing heap-block, the memory is copied to a new location and the new address is returned from the function. If this copying is not desired, an additional flag is supported by `HeapReAlloc` - `HEAP_REALLOC_IN_PLACE_ONLY` - that if specified, fails the re-allocation if the new size cannot fit in the existing block.

Once an allocation is no longer needed, call `HeapFree` to return it to the heap:

```
BOOL HeapFree(
    _Inout_ HANDLE hHeap,
    _In_ DWORD dwFlags,
    _In_ LPVOID lpMem);
```

The only valid flag for `HeapFree` is `HEAP_NO_SERIALIZE`. After a successful call to `HeapFree`, the `lpMem` address should be considered invalid. In most cases, this address remains valid in the sense of it still pointing to committed memory, but new allocations may reuse this address. No access violation exception is likely to be thrown if that memory is accessed without a proper allocation. This is in contrast to `VirtualFree` that decommits memory so that any access to the same page will raise an access violation exception.

Private Heaps

A heap starts its life as a reserved chunk of memory, part of which may be committed. A heap can have a fixed maximum size or be growable. A growable heap can grow as far as the process address space allows.

The default process heap is growable - its initial reserved and committed sizes can be specified with linker settings. Figure 13-8 shows a project's properties dialog in Visual Studio for setting these values.

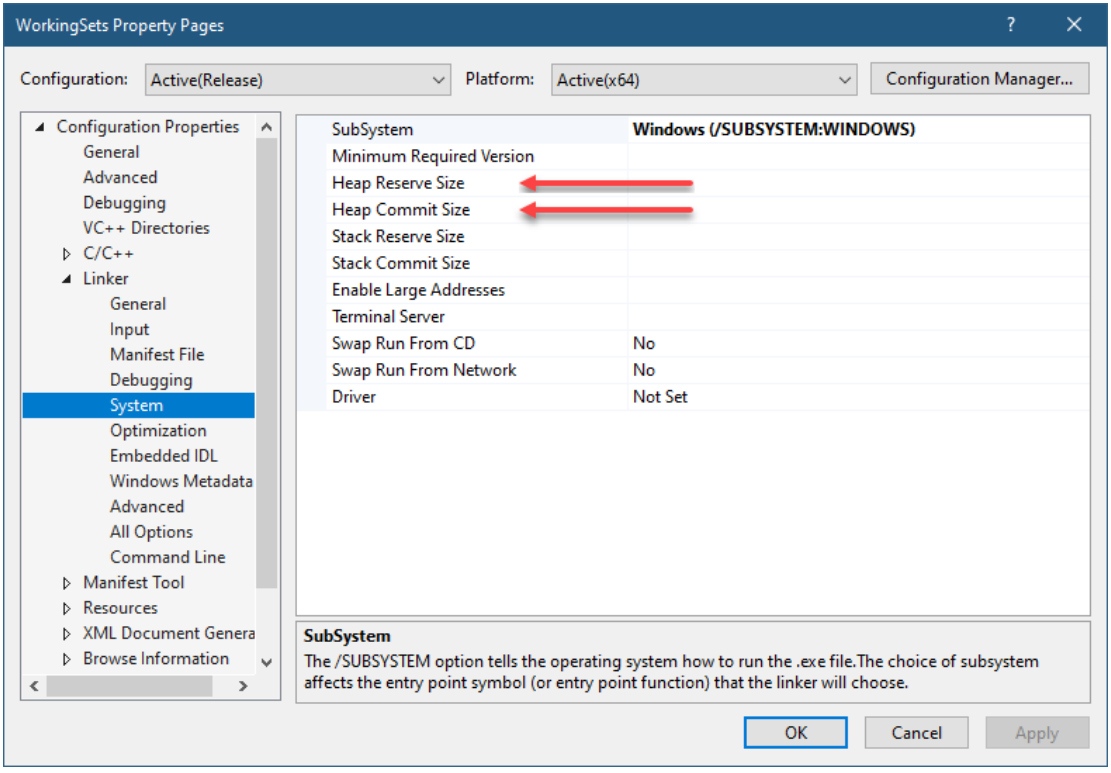


Figure 13-8: Heap sizes linker settings

The default sizes for the default process heap can be inspected with any PE viewer tool, such as *Dumpbin*. Here is an example for *Notepad*:

```
c:\>dumpbin /headers c:\Windows\System32\notepad.exe
Microsoft (R) COFF/PE Dumper Version 14.26.28805.0
Copyright (C) Microsoft Corporation.  All rights reserved.

Dump of file c:\Windows\System32\notepad.exe

PE signature found

File Type: EXECUTABLE IMAGE
...
OPTIONAL HEADER VALUES
             20B magic # (PE32+)
...
```

```
          100000 size of heap reserve
            1000 size of heap commit
...
```

The initial commit is a single page and the initial reserved size is 1 MB. This is the default (for the defaults), if no custom values are specified in the linker settings.

The default process heap always exists and cannot be destroyed. Since it's growable, why would you want to create additional heaps?

One reason is to avoid fragmentation. When various sizes are used for allocations on the same heap, fragmentation to one degree or another is inevitable. For example, suppose the application allocates several structures 16 bytes in size. Now suppose one of these structures is freed, and a new structure 24 bytes in size needs to be allocated. The gap of 16 bytes is not enough, and so the heap manager allocates the block at the end of the used heap area. If a 12-byte structure is needed, it can fit in that 16-byte space, but now the 4-byte space would probably never used again. This fragmentation leads to using more memory than is really needed. If the process lives for hours or even more, with lots of allocations and deallocations, the problem is compounded.

Every heap allocation carries with it some management overhead, in terms of CPU and memory. Thus, using `VirtualAlloc` is always a bit faster and does not have the same memory overhead.

One (partial) solution to the problem is using the *Low Fragmentation Heap* (LFH), described in the section *Heap Types*. The other is to create a separate (private) heap that holds blocks of a certain size. if one block is freed, there is exactly room for a new block of the same size. This scheme is useful if many objects of a certain size (typically a certain structure/class) are allocated and freed throughout the lifetime of the process.

Creating a new heap is done with `HeapCreate`:

```
HANDLE HeapCreate(
    _In_ DWORD flOptions,
    _In_ SIZE_T dwInitialSize,
    _In_ SIZE_T dwMaximumSize);
```

The first options parameter can be zero or a combination of three flags, two of which we met with `HeapAlloc`: `HEAP_GENERATE_EXCEPTIONS` and `HEAP_NO_SERIALIZE`. If `HEAP_NO_SERIALIZE` is specified, the heap manager does not hold any lock while performing heap operations. This means any concurrent access to the heap could result in a heap corruption. This option makes the heap manager slightly faster, but it's up to the developer to provide synchronized access to the heap if required. One potential downside of setting this flag, is that the heap cannot use the *Low Fragmentation Heap* (LFH) layer (described later).

The last flag, `HEAP_CREATE_ENABLE_EXECUTE`, tells the heap manager to allocate blocks with `PAGE_EXECUTE_READWRITE` access rather than just `PAGE_READWRITE`. This could be useful if code is to be written to the heap and executed, such as in *Just in time* (JIT) compilation scenarios.

The `dwInitialSize` parameter indicates the amount of memory that should be committed upfront. The value is rounded up to the nearest page. A value of zero equals a single page. `dwMaximumSize` specifies the maximum size the heap can grow to. The memory between the initial commit and the maximum size is kept reserved. If the maximum value specified is zero, then the heap is growable, otherwise this is the maximum size of the heap (fixed heap). Any attempt to allocate more than the heap's size, fails. If the heap is fixed then the largest chunk that can be allocated is slightly less than 512 KB in 32-bit systems and slightly less than 1 MB in 64-bit processes. You should not use the heap API anyway for such large sizes, but instead use the `VirtualAlloc` function.

The function's return value is a handle to the new heap, to be used with other heap functions such as `HeapAlloc` and `HeapFreee`. If the function fails, `NULL` is returned.

A private heap needs to be destroyed at some point, accomplished with `HeapDestroy`:

```
BOOL HeapDestroy(_In_ HANDLE hHeap);
```

`HeapDestroy` frees the entire heap in a single stroke, so there is no need to free every individual allocation if heap destruction is imminent. This could be yet another motivation to create a private heap.

One way to leverage a private heap with same-size structures is to use C++'s ability to override the `new` and `delete` operators, so that any dynamic allocation or deallocation of the structure ocurrs on the private heap, while the developer does not to concern herself with calling any specific APIs. Here is the header of such an example type:

```
class MyClass {
public:
    void* operator new(size_t);
    void operator delete(void*);

    void DoWork();
private:
    static HANDLE s_hHeap;
    static unsigned s_Count;
};
```

The class holds two static members (common to all instances of `MyClass`). These hold the heap handle and the count of instances, so that the heap can be destroyed when the last instance is freed. Here is the implementation:

```
HANDLE MyClass::s_hHeap = nullptr;
unsigned MyClass::s_Count = 0;

void* MyClass::operator new(size_t size) {
    if (InterlockedIncrement(&s_Count) == 1)
        s_hHeap = ::HeapCreate(0, 64 << 10, 16 << 20);
    return ::HeapAlloc(s_hHeap, 0, size);
}

void MyClass::operator delete(void* p) {
    ::HeapFree(s_hHeap, 0, p);
    if (::InterlockedDecrement(&s_Count) == 0)
        ::HeapDestroy(s_hHeap);
}
```

The private heap is created with 64 KB committed size and a maximum of 16 MB (these are just examples, of course). Any client to the `MyClass` type can use normal C++ operations:

```
auto obj = new MyClass;
obj->DoWork();
delete obj;
```

The client does not need to know that under the covers the objects are allocated from a private heap, that guarantees fragmentation-free usage.

Heap Types

We've seen that a heap can be created with a fixed maximum size or growable. To help mitigate heap fragmentation, Windows supports the *Low Fragmentation Heap* (LFH) for heaps that are not created without serialization (don't have the flag `HEAP_NO_SERIALIZE`). The LFH attempts to minimize fragmentation by using specific-sized blocks that have a better chance to fulfill allocations. For example, an 8-byte and a 12-byte allocation request will get 16 bytes. When such an allocation is freed, there is a better chance of putting a new allocation into the same block if its requested size is no more than 16 bytes. This means the LFH minimizes fragmentation by potentially using more memory than needed for each allocation.

The LFH is built as an optional front-layer for the standard heap. It's activated automatically under certain conditions, and cannot be turned off once activated. It's also not possible to force using the LFH.

Windows versions prior to Vista did allow turning the LFH on and off. It was difficult for developers to know when is a good time for such operations, so now the heap manager uses its own tuning logic.

You can query for the type of a heap with `HeapQueryInformation`:

```
BOOL HeapQueryInformation(
    _In_opt_ HANDLE HeapHandle,
    _In_ HEAP_INFORMATION_CLASS HeapInformationClass,
    _Out_ PVOID HeapInformation,
    _In_ SIZE_T HeapInformationLength,
    _Out_opt_ PSIZE_T ReturnLength);
```

The function provides a generic way to query some heap parameters, based on the `HEAP_INFORMATION_CLASS` enumeration, of which one value (`HeapCompatibilityInformation`) is officially documented. The output buffer is a 32-bit number, which can be 0 (no LFH) or 2 (LFH). The following example queries the type of the default process heap:

```
ULONG type;
::HeapQueryInformation(::GetProcessHeap(),
    HeapCompatibilityInformation, &type,
    sizeof(type), nullptr);
```

Windows 8 introduced another type of heap called the *Segment Heap*. This heap has better management of blocks and extra security measures to help prevent malicious code in the process from recognizing heap blocks just by having a pointer to somewhere on the heap. The segment heap is used by all UWP processes because it has a smaller memory footprint that is beneficial for UWP processes running on small devices such as phones or tablets. Some system processes use the segment heap as well, including *smss.exe*, *csrss.exe* and *svchost.exe*.

The segment heap is not the default heap for compatibility reasons. It is possible to enable it for a specific executable by creating a subkey with the executable name (with the EXE extension but no path) in the registry key *HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options* (this requires admin access). In that subkey, add a DWORD value named **FrontEndHeapDebugOptions** and set its value to 8. Figure 13-9 shows this value set for *Notepad.exe*. The next time this executable is launched, it will use the segment heap by default.

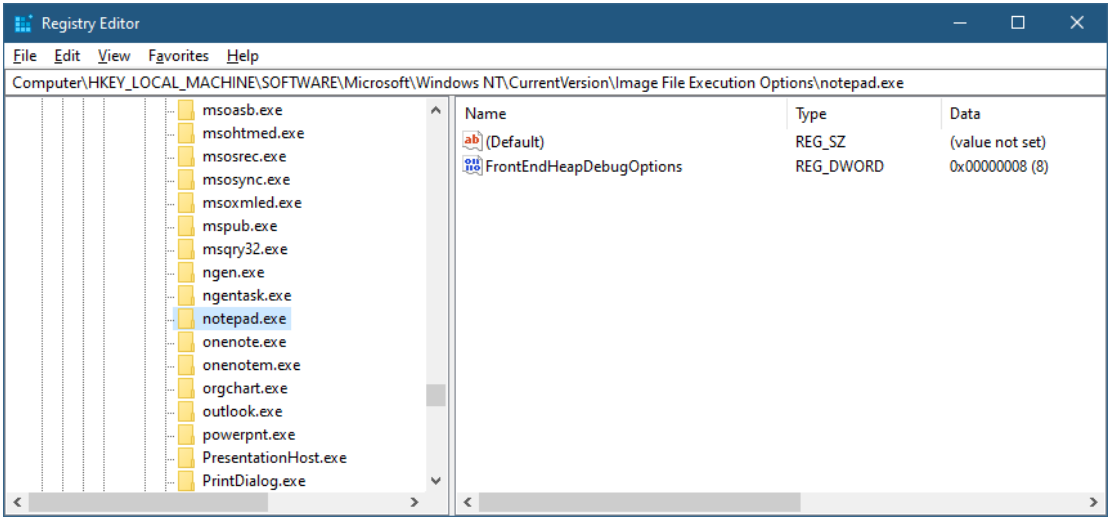


Figure 13-9: Enabling the segment heap for an executable

For more information on the segment heap, consult chapter 5 in the "Windows Internals 7th edition Part 1" book.

Heap Debugging Features

One too common occurrence when working with the heap APIs is heap corruption, typically due to accessing memory with dangling pointers, or writing beyond an allocation's size. These bugs are notoriously difficult to track down in all but the simplest applications. The main issue making such problems difficult to solve is the fact that the corruption itself is usually detected much later, when the offending code is no longer in the call stack. Other reasons for these occurrences is sometimes due to malicious code injection attempts on the heap. The heap manager provides some help in this regard.

You can call the `HeapValidate` function to tell the heap manager to scan all allocations from a given heap and validate their integrity:

```
BOOL HeapValidate(
    _In_ HANDLE hHeap,
    _In_ DWORD dwFlags,
    _In_opt_ LPCVOID lpMem);
```

The only valid flag to the function is `HEAP_NO_SERIALIZE`, discussed earlier. If `lpMem` is `NULL`, the entire heap is scanned, otherwise just the provided memory block is checked for integrity. If `lpMem` is not `NULL`, the pointer must be to a start of an allocation (returned by a previous call to `HeapAlloc` or `HeapReAlloc`).

`HeapValidate` returns `TRUE` if the heap/block is valid, and `FALSE` otherwise. If a debugger is connected to the process, an exception is triggered, breaking into the debugger by default. Validating a full heap can be time-consuming, so this option should generally be used for debugging purposes only.

If `lpMem` is `NULL`, `HeapValidate` always returns `TRUE` for the *segment heap*.

Some errors will go undetected by `HeapValidate`. If some code wrote to a block beyond its size, that happened to be free, or there was extra memory because the LFH is used, `HeapValidate` would miss the overflow.

Another option is to request the heap manager to terminate the process in case a heap corruption is detected (instead of just marching on), by calling `HeapSetInformation`:

```
BOOL HeapSetInformation(
    _In_opt_ HANDLE HeapHandle,
    _In_ HEAP_INFORMATION_CLASS HeapInformationClass,
    _In_ PVOID HeapInformation,
    _In_ SIZE_T HeapInformationLength);
```

The *HEAP_INFORMATION_CLASS* in question is `HeapEnableTerminationOnCorruption`. This option is process-wide, so the heap handle is ignored, and can be specified as `NULL`. The `HeapInformation` and `HeapInformationLength` should be set to `NULL` and zero, respectively. Once enabled, this feature cannot be disabled for the lifetime of the process.

There are some other heap debugging features that can be set by using certain flags set with the **NtGlobalFlags** value in the *Image File Execution Options* registry key used in a previous section. Normally, these values are set with a tool, such as *GFlags* (part of the Debugging Tools for Windows package, normally installed with the Windows SDK), or my own *GFlagsX* tool. Figure 13-10 shows *GFlags* used for *Notepad.exe*. There are several heap-related options, marked in figure 13-10. Consult the *GFlags* tool documentation for a description of these options.

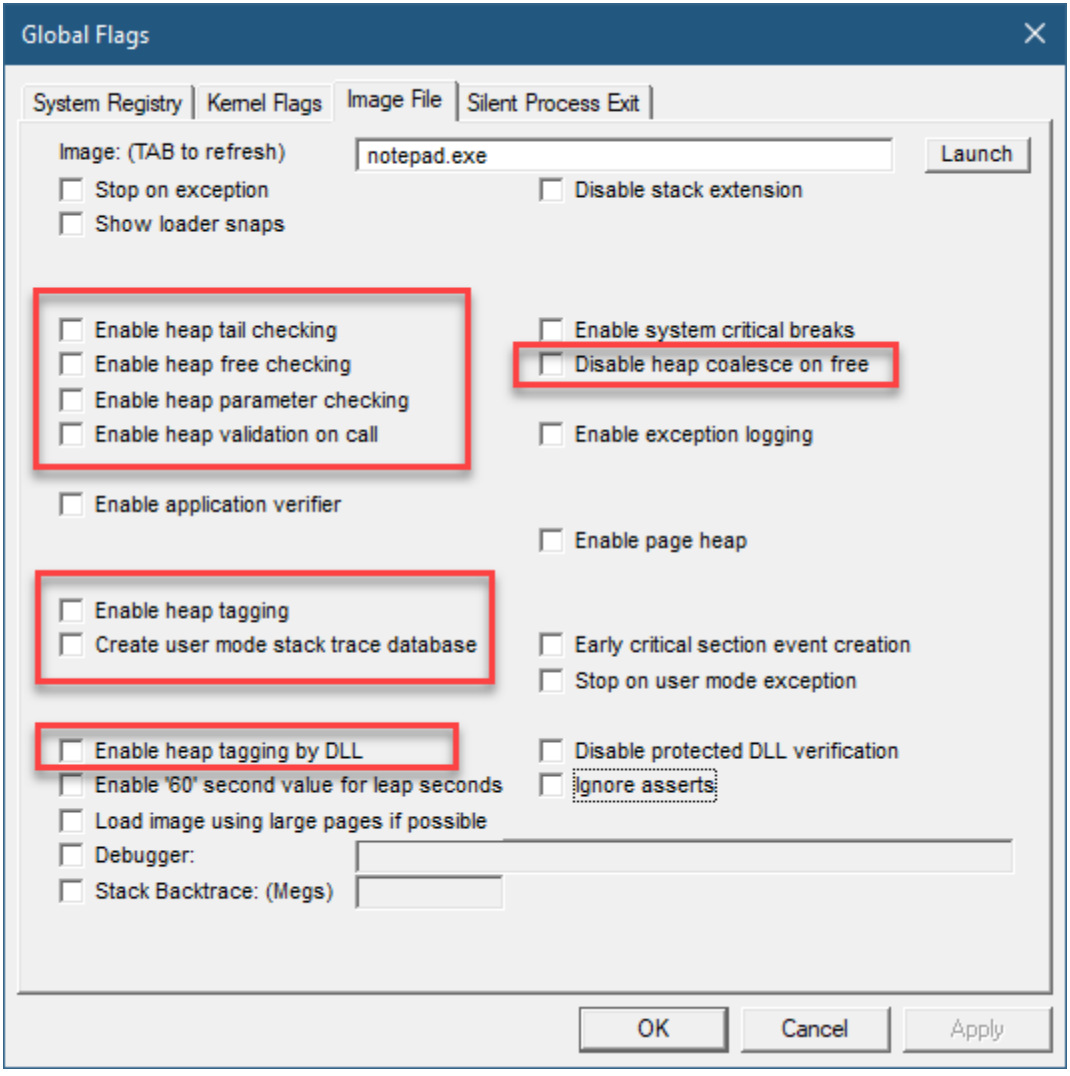


Figure 13-10: GFlags Heap related options

Using any of the debugging options (except "Disable heap coalesce on free") slows down all heap operations, so it's best to use these only while debugging or troubleshooting heap related issues.

The C/C++ Runtime

The implementation of the C/C++ memory management functions, such as `malloc`, `calloc`, `free`, the C++ `new` and `delete` operators, and so on, depends on the compiler-provided libraries. Since we do

our work mostly with Microsoft's compilers, then we can say something about the Visual C++ runtime implementation of these functions.

The C/C++ runtime uses the heap functions to manage their allocation. Current implementation use the default process heap. Since the CRT source code is distributed with Visual Studio, it's possible to simply look at the implementation. Here is the implementation of `malloc` with some macros and directives removed for clarity (in *malloc.cpp*):

```
extern "C" void* __cdecl malloc(size_t const size) {
    #ifdef _DEBUG
    return _malloc_dbg(size, _NORMAL_BLOCK, nullptr, 0);
    #else
    return _malloc_base(size);
    #endif
}
```

There are two implementations for `malloc` - one for debug build and another for release builds. Here is an excerpt from the release build version (in the file *malloc_base.cpp*):

```
extern "C" __declspec(noinline)
void* __cdecl _malloc_base(size_t const size) {
    // Ensure that the requested size is not too large:
    _VALIDATE_RETURN_NOEXC(_HEAP_MAXREQ >= size, ENOMEM, nullptr);

    // Ensure we request an allocation of at least one byte:
    size_t const actual_size = size == 0 ? 1 : size;

    for (;;) {
        void* const block = HeapAlloc(__acrt_heap, 0, actual_size);
        if (block)
            return block;
    //...code omitted...
}
```

The global `__acrt_heap` is initialized in this function (from *heap_handle.cpp*):

```
extern "C" bool __cdecl __acrt_initialize_heap() {
    __acrt_heap = GetProcessHeap();
    if (__acrt_heap == nullptr)
        return false;

    return true;
}
```

This implementation uses the default process heap.

The Local/Global APIs

The set APIs having the prefixes `Local` and `Global`, such as `LocalAlloc`, `GlobalAlloc`, `LocalFree`, `LocalLock` and others were created primarily for compatibility with 16-bit Windows. Their use is awkward, since an allocation returns a handle of sorts, that needs to be turned into a pointer with a `LocalLock` or `GlobalLock` function.

There are very specific scenarios that unfortunately require the use of some of these functions. The first relates to clipboard operations. Putting some data on the clipboard requires using an `HGLOBAL` handle returned from `GlobalAlloc`. Another scenario involves some APIs (especially security APIs), some of which allocate some data to be returned to the caller. The caller often has to free the data when it's no longer needed with the `LocalFree` function.

In short, these functions should only be used when some constraint requires it. For all other cases, use heap, C/C++, or `Virtual` APIs.

Other Heap Functions

There are a few more heap functions not covered by the previous sections. This section briefly describes the functions and their uses.

Getting summary information for a certain heap is available with `HeapSummary`:

```
typedef struct _HEAP_SUMMARY {
    DWORD cb;
    SIZE_T cbAllocated;
    SIZE_T cbCommitted;
    SIZE_T cbReserved;
    SIZE_T cbMaxReserve;
} HEAP_SUMMARY, *PHEAP_SUMMARY;

BOOL HeapSummary(
    _In_ HANDLE hHeap,
    _In_ DWORD dwFlags,
    _Out_ PHEAP_SUMMARY lpSummary);
```

Before calling `HeapSummary`, initialize the `cb` member to the size of the structure. The members of `HEAP_SUMMARY` are as follows:

- `cbAllocated` is the number of bytes currently allocated (actively used) on the heap.
- `cbCommitted` is the number of bytes currently committed on the heap.
- `cbReserved` is the reserved memory size to which the heap can grow.
- `cbMaxReserve` is the same as `cbReserved` in the current implementation.

The `HeapSize` function allows querying the size of an allocated block:

```
SIZE_T HeapSize(
    _In_ HANDLE hHeap,
    _In_ DWORD dwFlags,
    _In_ LPCVOID lpMem);
```

The only valid flag is `HEAP_NO_SERIALIZE`. The `lpMem` pointer must be one that was returned earlier from `HeapAlloc` or `HeapReAlloc`.

A heap that is created without `HEAP_NO_SERIALIZE` maintains a critical section (lock) to prevent heap corruption in case of concurrent access. The `HeapLock` and `HeapUnlock` allow acquiring and releasing the heap's critical section:

```
BOOL HeapLock(_In_ HANDLE hHeap);
BOOL HeapUnlock(_In_ HANDLE hHeap);
```

This could be used to speed up multiple operations that can be invoked without taking the lock. One such operation that can be used for debugging purposes is walking the heap blocks with `HeapWalk`:

```
BOOL HeapWalk(
    _In_ HANDLE hHeap,
    _Inout_ LPPROCESS_HEAP_ENTRY lpEntry);
```

A heap enumeration works by writing a loop that returns structures of type `PROCESS_HEAP_ENTRY`:

```
typedef struct _PROCESS_HEAP_ENTRY {
    PVOID lpData;
    DWORD cbData;
    BYTE cbOverhead;
    BYTE iRegionIndex;
    WORD wFlags;
    union {
        struct {
            HANDLE hMem;
            DWORD dwReserved[ 3 ];
        } Block;
        struct {
            DWORD dwCommittedSize;
            DWORD dwUnCommittedSize;
            LPVOID lpFirstBlock;
            LPVOID lpLastBlock;
        } Region;
    } DUMMYUNIONNAME;
} PROCESS_HEAP_ENTRY, *LPPROCESS_HEAP_ENTRY, *PPROCESS_HEAP_ENTRY;
```

A heap enumeration starts by allocating a `PROCESS_HEAP_ENTRY` instance and setting its `lpData` to `NULL`. Each call to `HeapWalk` gets back data for the next allocated block on the heap. The enumeration ends when `HeapWalk` returns `FALSE`. Check out the documentation for a detailed description of the various fields.

Write a heap enumeration function that accepts a heap handle and display the various blocks with their properties.

> Heap enumeration with `HeapWalk` is only available for the current process. If such heap walking is desired for other processes, the *ToolHelp* offer a flag for `CreateToolhelp32Snapshot` (`TH32CS_SNAPHEAPLIST`), which provides a way to enumerate heaps in a selected process (`Heap32ListFirst`, `Heap32ListNext`), and go further with enumerating blocks in each heap (`Heap32First`, `Heap32Next`).

When working with the heap, there may be two or more contiguous free memory blocks. Normally, the heap automatically coalesces these adjacent free blocks. One of the global flags shown in figure 13-10 allows disabling this feature (called "Disable Heap Coalesce on Free"), to save some processing time. In that case, you can call `HeapCompact` to force this coalescing to occur:

```
SIZE_T HeapCompact(
    _In_ HANDLE hHeap,
    _In_ DWORD dwFlags);
```

The only valid flag is `HEAP_NO_SERIALIZE`. The function coalesces free blocks if this was disabled, and returns the largest block that can be allocated on the heap.

Finally, getting handles to all heaps in the current process is possible with `GetProcessHeaps`:

```
DWORD GetProcessHeaps(
    _In_ DWORD NumberOfHeaps,
    _Out_ PHANDLE ProcessHeaps);
```

The function accepts a maximum number of heap handles to return and an array of handles. It returns the total number of heaps in the process. If the number is greater than `NumberOfHeaps`, then not all heap handles have been returned. A simple way to get the number of heaps in the current process is with this snippet:

```
DWORD heapCount = ::GetProcessHeaps(0, nullptr);
```

Other `Virtual` Functions

In this section, we'll look at other functions from the `Virtual` family of APIs, except for the `VirtualQuery` functions that are covered in chapter 12.

Memory Protection

Once a memory region is committed, the `VirtualProtect*` set of functions can be used to change the page protection for a region of pages that are part of the same initial reserved region:

```
BOOL VirtualProtect(
    _In_ LPVOID lpAddress,
    _In_ SIZE_T dwSize,
    _In_ DWORD flNewProtect,
    _Out_ PDWORD lpflOldProtect);
BOOL VirtualProtectEx(
    _In_ HANDLE hProcess,
    _In_ LPVOID lpAddress,
    _In_ SIZE_T dwSize,
    _In_ DWORD flNewProtect,
    _Out_ PDWORD lpflOldProtect);
BOOL VirtualProtectFromApp(
    _In_ PVOID Address,
    _In_ SIZE_T Size,
    _In_ ULONG NewProtection,
    _Out_ PULONG OldProtection);
```

As with `VirtualAllocEx`, `VirtualProtectEx` is able to perform the operation in a different process context, assuming the process handle has the `PROCESS_VM_OPERATION` access mask. `VirtualProtectFromApp` is the variant allowed to be called from a UWP process. As with `VirtualAlloc`, `VirtualProtect` is implemented inline if the macro `WINAPI_PARTITION_APP` is defined (indicating an AppContainer caller), by calling `VirtualProtectFromApp`.

The protection is changed for the address range spanning all pages between `lpAddress` and `lpAddress+dwSize-1`, setting the new protection to `flNewProtect` (see chapter 12 for a list of the possible protection attributes). The previous protection for the range of pages is returned via the `lpflOldProtect` parameter (which cannot be `NULL`). If the span of pages had more than a single protection value, the first one is returned.

Locking Memory

As we've seen already, committed memory that is part of a process' working set is not guaranteed to remain in the working set, but may be paged out. In some cases, a process might want to tell the memory manager that a certain memory buffer should not be paged out, even if it's not accessed for a long time. The `VirtualLock` function can be used for this purpose, with `VirtualUnlock` being the way to remove the locking:

```
BOOL VirtualLock(
    _In_ LPVOID lpAddress,
    _In_ SIZE_T dwSize);

BOOL VirtualUnlock(
    _In_ LPVOID lpAddress,
    _In_ SIZE_T dwSize);
```

The range of addresses for both functions is always rounded up to the nearest page boundaries, just like any `Virtual` API. The maximum size a process can lock is a bit less than its minimum working set size. If a larger block is to be locked, `SetProcessWorkingSetSize(Ex)` should be called to increase the minimum (and probably the maximum) working set sizes. Of course, a process should be careful not to lock too much memory because of the adverse effect it can have on other processes and the system as a whole.

Memory Block Information

Windows 10 version 1607 and Server 2016 introduced the `QueryVirtualMemoryInformation` function:

```
BOOL QueryVirtualMemoryInformation(
    _In_ HANDLE Process,
    _In_ const VOID* VirtualAddress,
    _In_ WIN32_MEMORY_INFORMATION_CLASS MemoryInformationClass,
    _Out_ PVOID MemoryInformation,
    _In_ SIZE_T MemoryInformationSize,
    _Out_opt_ PSIZE_T ReturnSize);
```

The only currently supported value for `MemoryInformationClass` is `MemoryRegionInfo` and the returned information is described by the `WIN32_MEMORY_REGION_INFORMATION` structure:

```
typedef struct WIN32_MEMORY_REGION_INFORMATION {
    PVOID AllocationBase;
    ULONG AllocationProtect;

    union {
        ULONG Flags;

        struct {
            ULONG Private : 1;
            ULONG MappedDataFile : 1;
            ULONG MappedImage : 1;
            ULONG MappedPageFile : 1;
            ULONG MappedPhysical : 1;
```

```
            ULONG DirectMapped : 1;
            ULONG Reserved : 26;
        } DUMMYSTRUCTNAME;
    } DUMMYUNIONNAME;

    SIZE_T RegionSize;
    SIZE_T CommitSize;
} WIN32_MEMORY_REGION_INFORMATION;
```

The function is somewhat similar to `VirtualQueryEx`, with the following differences:

- `QueryVirtualMemoryInformation` fails for an address pointing to a free page.
- It has some extra details for mapped memory (`MEM_MAPPED` in `VirtualQueryEx`):

`MappedDataFile`, `MappedPageFile`, `MappedPhysical` and `DirectMapped`.

Refer to the documentation for the full details.

Memory Hint Functions

This section describes functions that are never absolutely necessary, but can be used to improve performance in one respect or another, by giving the memory manager hints at the application's use of committed memory.

The `OfferVirtualMemory` function, introduced in Windows 8.1 (and Server 2012 R2), indicates to the memory manager that a range of committed memory is no longer of interest, so the system can discard the physical pages being used by that memory. The system should not bother writing the data to a page file.

```
typedef enum OFFER_PRIORITY {
    VmOfferPriorityVeryLow = 1,
    VmOfferPriorityLow,
    VmOfferPriorityBelowNormal,
    VmOfferPriorityNormal
} OFFER_PRIORITY;

DWORD OfferVirtualMemory(
    _Inout_ PVOID VirtualAddress,
    _In_ SIZE_T Size,
    _In_ OFFER_PRIORITY Priority);
```

The virtual address must be page-aligned, and the size must be a multiple of the page size.
At first it may seem that simply decommitting the memory (`VirtualFree`) has the same effect. From the point of view of releasing RAM, it's similar. But with `VirtualFree`, the application is giving up the

address range and the committed memory, so that a new allocation would be needed in the future, which may or may not succeed; it's also slower than just reusing an already existing committed memory block.

The priority parameter specifies the importance of the memory region. The lowest the priority, the more likely the physical memory will be discarded sooner rather than later. The function returns an error code directly (no point in calling `GetLastError`), where `ERROR_SUCCESS` (0) indicates success.

Once the application is ready to use the offered memory again, it can reclaim it back with `ReclaimVirtualMemory`:

```
DWORD ReclaimVirtualMemory(
    _In_ void const* VirtualAddress,
    _In_ SIZE_T Size);
```

The reclaimed memory may or may not have its previous contents. The application should assume the memory is needs to be repopulated with meaningful data.

Another variant similar to `OfferVirtualMemory` is `DiscardVirtualMemory`:

```
DWORD DiscardVirtualMemory(
    _Inout_ PVOID VirtualAddress,
    _In_ SIZE_T Size);
```

`DiscardVirtualMemory` is equivalent to calling `OfferVirtualMemory` with the `VmOfferPriorityVeryLow` priority.

The last function in this section is `PrefetchVirtualMemory` (available from Windows 8 and Server 2012):

```
typedef struct _WIN32_MEMORY_RANGE_ENTRY {
    PVOID VirtualAddress;
    SIZE_T NumberOfBytes;
} WIN32_MEMORY_RANGE_ENTRY, *PWIN32_MEMORY_RANGE_ENTRY;

BOOL PrefetchVirtualMemory(
    _In_ HANDLE hProcess,
    _In_ ULONG_PTR NumberOfEntries,
    _In_ PWIN32_MEMORY_RANGE_ENTRY VirtualAddresses,
    _In_ ULONG Flags);
```

The purpose of `PrefetchVirtualMemory` is to allow an application to optimize I/O usage for reading data from discontiguous blocks of memory that the process is fairly confident it's going to use. The caller provides the committed memory blocks (using an array of `WIN32_MEMORY_RANGE_ENTRY` structures), and the memory manager uses concurrent I/O with large buffers to fetch the data quicker than it would if the pages were to be accessed normally. This function is never *necessary*, it's merely an optimization.

Writing and Reading to/from Other Processes

Normally processes are protected from one another, but one process can read and/or write to another's process address space given strong enough handles. Here are the functions to use:

```
BOOL ReadProcessMemory(
    _In_ HANDLE hProcess,
    _In_ LPCVOID lpBaseAddress,
    _Out_ LPVOID lpBuffer,
    _In_ SIZE_T nSize,
    _Out_opt_ SIZE_T* lpNumberOfBytesRead);

BOOL WriteProcessMemory(
    _In_ HANDLE hProcess,
    _In_ LPVOID lpBaseAddress,
    _In_ LPCVOID lpBuffer,
    _In_ SIZE_T nSize,
    _Out_opt_ SIZE_T* lpNumberOfBytesWritten);
```

`ReadProcessMemory` requires a handle with `PROCESS_VM_READ` access mask, while `WriteProcessMemory` requires `PROCESS_VM_WRITE` access mask. `lpBaseAddress` is the address in the target process to read/write. `lpBuffer` is the local buffer to read to or write from. `nSize` is the size of the buffer to read/write. Finally, the last optional parameter returns the number of bytes actually read or written.

Even with the process access mask, these functions may fail because of incompatible page protection. For example, `WriteProcessMemory` fails to write to pages protected with `PAGE_READONLY`. Of course, the caller could try to change the protection with `VirtualProtectEx`.

The primary user of these functions is debuggers. A debugger must be able to read information from the debugged process, such as local variables, threads' stack, etc. Similarly, a debugger allows its user to make changes to data in the debugged process. There are other uses for these functions, however. We'll see one example for `WriteProcessMemory` in chapter 15 to help inject a DLL into a target process.

Large Pages

Windows supports two basic page sizes, small and large (with a third huge page size described in an upcoming aside). Table 12-1 shows the sizes of pages, where small pages are 4 KB on all architectures, and large pages are 2 MB on all but ARM architecture, where it's 4 MB. The `VirtualAlloc` family of functions support allocation using large pages with the `MEM_LARGE_PAGE` flag. What are the benefits of using large pages?

- Large pages perform better internally because the translation of virtual to physical addresses do not use page tables (only page directories, see the "Windows Internals" book). This also makes the *Translation Lookaside Buffer* (TLB) CPU cache more effective - a single entry maps 2 MB rather than just 4 KB.

- Large pages are always non-pageable (never paged out to disk).

Large pages come with a few downsides, however:

- Large pages cannot be shared between processes.
- Large page allocations must be an exact multiple of large page size.
- Large page allocations may fail if physical memory is too fragmented.

There is yet another important caveat - since large pages are always non-pageable, using large pages requires the **SeLockMemoryPrivilege** privilege, normally given to no user, including the Administrators group. Figure 13-11 shows the *Local Security Policy* tool showing the list of privileges, where the "Lock Pages in Memory" privilege is shown with no users or groups assigned.

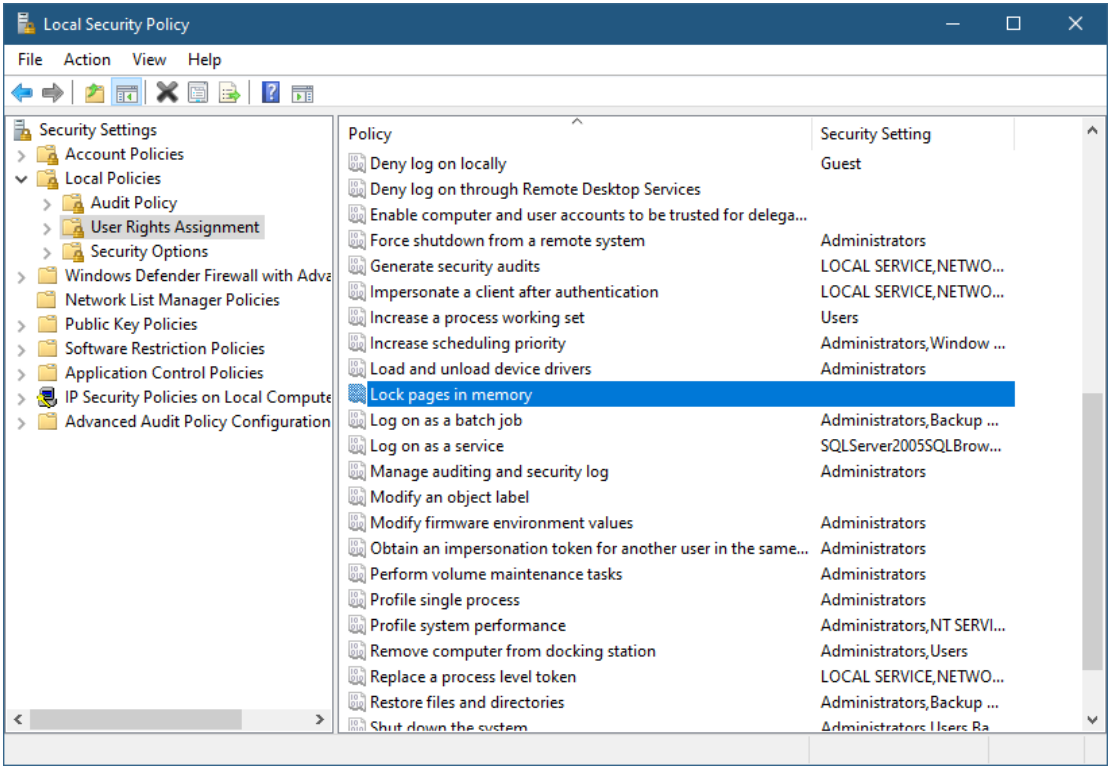


Figure 13-11: *Local Security Policy* privileges window

There are two ways to get the required privilege:

- An administrator can add users/group to have this privilege. The next time such a user logs off and logs on again, the privilege will be part of its access token.
- A service running under the Local System account can request any privilege it desires. (Services are described in chapter 19, along with this capability).

The large page size on a system can be queried with `GetLargePageMinimum`. This is important since large page allocations must be done in multiples of large page size:

```
SIZE_T GetLargePageMinimum();
```

Huge Pages

Modern processors support a third page size, huge page, 1 GB in size. The benefits of huge pages is essentially the same as for large pages, with even better use of the TLB cache. Windows 10 and Server 2016 (and later) support huge pages (if supported by the hardware). There is no flag to `VirtualAlloc` for using huge pages. Instead, when large page allocations occur, if the size is at least 1 GB, the system tries to locate huge pages first, and then use large pages for the remainder. If huge pages cannot be obtained (since that requires contigous 1 GB chunks in physical memory), large pages will be used as fallback.

Having the *SeLockMemoryPrivilege* privilege is not enough - it must be enabled as well. This is easy enough to do with a function very similar to the one used in chapter 3 to enable the Debug privilege (a detailed discussion of this code is saved for chapter 16):

```
bool EnableLockMemoryPrivilege() {
    HANDLE hToken;
    if (!::OpenProcessToken(::GetCurrentProcess(),
        TOKEN_ADJUST_PRIVILEGES, &hToken))
        return false;

    bool result = false;
    TOKEN_PRIVILEGES tp;
    tp.PrivilegeCount = 1;
    tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
    if (::LookupPrivilegeValue(nullptr, SE_LOCK_MEMORY_NAME,
        &tp.Privileges[0].Luid)) {
        if (::AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(tp),
            nullptr, nullptr))
            result = ::GetLastError() == ERROR_SUCCESS;
    }
    ::CloseHandle(hToken);
    return result;
}
```

Now using large pages is no different than using normal pages except for the `MEM_LARGE_PAGE` flag:

```
if (!EnableLockMemoryPrivilege()) {
    printf("Failed to enable privilege\n");
    return 1;
}
auto largePage = ::GetLargePageMinimum();
// allocate 5 large pages
auto p = ::VirtualAlloc(nullptr, 5 * largePage,
    MEM_RESERVE | MEM_COMMIT | MEM_LARGE_PAGES, PAGE_READWRITE);
if (p) {
    // success - use memory
}
else {
    printf("Error: %u\n", ::GetLastError());
}
// free memory
::VirtualFree(p, 0, MEM_RELEASE);
```

Address Windowing Extensions

In the early days of Windows NT, the system supported no more than 4 GB of physical memory, which was the maximum processors Windows ran on supported at the time. Starting with the Intel Pentium Pro, more than 4 GB of physical memory were supported on 32-bit systems (64 bit was not around at the time). Applications that wanted to utilize the extra physical memory beyond 4 GB had to use a special API called *Address Windowing Extensions* (AWE) - the usage of any physical memory above 4 GB was not "automatic".

The AWE allows a process to allocate directly physical pages, and then map them to their address space. Since the amount of physical memory allocated could be larger than can fit in a 32-bit address space, the application can map a "window" into this memory, use the memory, unmap the window, and map a new window at another offset.

> This mechanism is awkward and was used very little in practice. The only well-known application that used AWE to gain the advantages of large memory on 32-bit systems was SQL Server.

Since using AWE means the application is allocating physical pages, the *SeLockMemoryPrivilege* privilege is required, just like with large pages.

Here is an example that uses AWE to allocate and use physical pages:

```
EnableLockMemoryPrivilege();

ULONG_PTR pages = 1000;         // pages
ULONG_PTR parray[1000];         // opaque array (PFNs)
if (!::AllocateUserPhysicalPages(::GetCurrentProcess(), &pages, parray))
    return ::GetLastError();

if (pages < 1000)
    printf("Only allocated %zu pages\n", pages);

// access the first 200 pages at most
auto usePages = min(pages, 200);

// reserve memory region for mapping physical pages
void* pWindow = ::VirtualAlloc(nullptr, usePages << 12,
    MEM_RESERVE | MEM_PHYSICAL,
    PAGE_READWRITE);    // read/write is the only valid value
if (!pWindow)
    return ::GetLastError();

// map pages to the process address space
if (!::MapUserPhysicalPages(pWindow, usePages, parray))
    return ::GetLastError();

// use the memory...
::memset(pWindow, 0xff, usePages << 12);

// cleanup
::FreeUserPhysicalPages(::GetCurrentProcess(), &pages, parray);
::VirtualFree(pWindow, 0, MEM_RELEASE);
return 0;
```

AWE allocated pages are non pageable and must be protected with `PAGE_READWRITE` - no other value is supported.

32-bit processes running on 64-bit Windows (*WOW64*) cannot use AWE functions.

AWE is rarely used today, because on 64-bit systems (the norm), any amount of physical memory is accessible without any special API usage, although normal memory usage does not guarantee it will always be resident. The awkwardness of AWE and the fact it requires the *SeLockMemoryPrivilege* privilege makes it almost useless.

NUMA

Non-Uniform Memory Architecture (NUMA) systems involve a set of *nodes*, each one holding a set of processors and memory. Figure 13-12 shows an example topology of such a system.

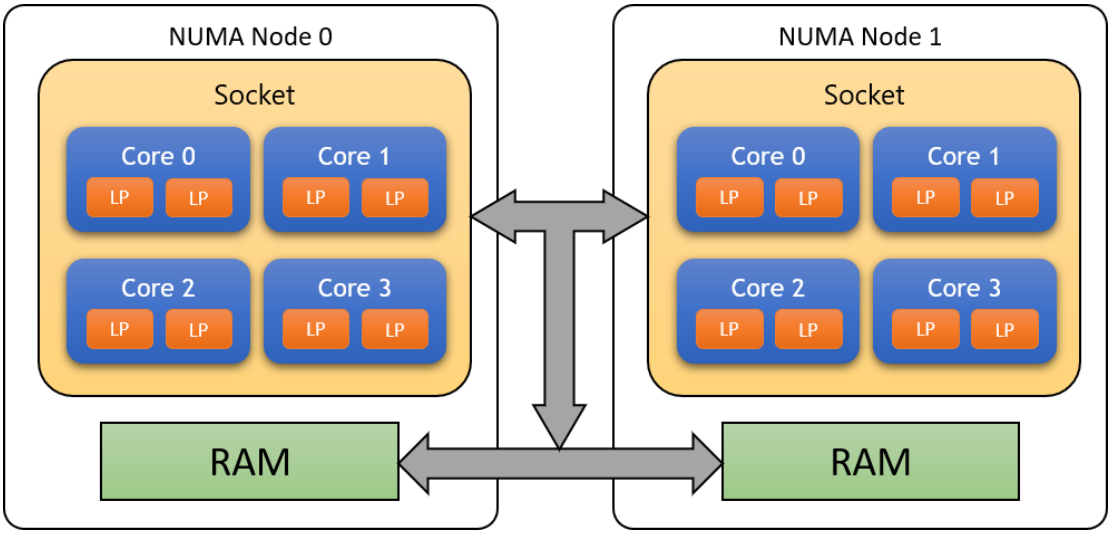


Figure 13-12: A NUMA system

Figure 13-12 shows an example of a system with two NUMA nodes. Each node holds a socket with 4 cores and 8 logical processors. NUMA systems are still symmetric in the sense that any CPU can run any code and access any memory in any node. However, accessing memory from a local node is considerably faster than accessing memory in another node.

Windows is aware of a NUMA system's topology. With thread scheduling discussed in chapter 6, the scheduler makes good use of this information, and tries to schedule threads on CPUs where the thread's stack is in that node's physical memory.

NUMA systems are common for server machines, where multiple sockets typically exist. Getting the NUMA topology information involves several API calls. The number of NUMA nodes on a system is available (somewhat indirectly) with `GetNumaHighestNodeNumber`:

```
BOOL GetNumaHighestNodeNumber(_Out_ PULONG HighestNodeNumber);
```

The function provides the highest NUMA node on the system, where 0 means it's not a NUMA system. On a two-node system `*HighestNodeNumber` is set to 1 on return.

For each node, the process affinity mask (the processors connected to a node) is available with `GetNumaNodeProcessorMaskEx`:

```
BOOL GetNumaNodeProcessorMaskEx(
    _In_ USHORT Node,
    _Out_ PGROUP_AFFINITY ProcessorMask);
```

The processors connected to a specific node can be retrieved with `GetNumaNodeProcessorMask` or `GetNumaNodeProcessorMaskEx`:

```
BOOL GetNumaNodeProcessorMask(
    _In_  UCHAR Node,
    _Out_ PULONGLONG ProcessorMask);
BOOL GetNumaNodeProcessorMaskEx(
    _In_ USHORT Node,
    _Out_ PGROUP_AFFINITY ProcessorMask
    );
```

`GetNumaNodeProcessorMask` is suitable for systems with less than 64 processors (it returns the process mask in the group this node is part of), but `GetNumaNodeProcessorMaskEx` can handle any number of processors by returning a `GROUP_AFFINITY` structure that combines a processor group and a bit mask of processors (see chapter 6 for more on process groups):

```
typedef struct _GROUP_AFFINITY {
    KAFFINITY Mask;
    WORD   Group;
    WORD   Reserved[3];
} GROUP_AFFINITY, *PGROUP_AFFINITY;
```

The amount of available physical memory in a node is available with `GetNumaAvailableMemoryNodeEx`. This can be used as a hint when allocating memory and targeting a specific node:

```
BOOL GetNumaAvailableMemoryNodeEx(
    _In_  USHORT Node,
    _Out_ PULONGLONG AvailableBytes);
```

The following function shows NUMA node information using the above functions:

```
void NumaInfo() {
    ULONG highestNode;
    ::GetNumaHighestNodeNumber(&highestNode);
    printf("NUMA nodes: %u\n", highestNode + 1);

    GROUP_AFFINITY group;
    for (USHORT node = 0; node <= (USHORT)highestNode; node++) {
        ::GetNumaNodeProcessorMaskEx(node, &group);
        printf("Node %d:\tProcessor Group: %2d, Affinity: 0x%08zX\n",
            (int)node, group.Group, group.Mask);
        ULONGLONG bytes;
        ::GetNumaAvailableMemoryNodeEx(node, &bytes);
        printf("\tAvailable memory: %llu KB\n", bytes >> 10);
    }
}
```

Here is an example run (2 nodes, 8 total processors):

```
NUMA nodes: 2
Node 0: Processor Group:  0, Affinity: 0x0000000F
        Available memory: 3567936 KB
Node 1: Processor Group:  0, Affinity: 0x000000F0
        Available memory: 3283832 KB
```

The `VirtualAlloc` function lets the system decide where the physical memory for committed memory should come from. If you want to select a preferred NUMA node, call `VirtualAllocExNuma`:

```
LPVOID VirtualAllocExNuma(
    _In_ HANDLE hProcess,
    _In_opt_ LPVOID lpAddress,
    _In_ SIZE_T dwSize,
    _In_ DWORD flAllocationType,
    _In_ DWORD flProtect,
    _In_ DWORD nndPreferred);
```

The function is identical to `VirtualAllocEx`, but adds a preferred NUMA node number as its last parameter. The provided NUMA node has effect only when the initial memory block is reserved or reserved and committed. Further manipulations of the same memory region disregard the NUMA node parameter, and it's much easier to continue working with `VirtualAlloc(Ex)`.

Here is an example that uses the above `NumaInfo` function twice, where some memory is committed (and forced into RAM) between calls:

```
NumaInfo();
auto p = ::VirtualAllocExNuma(::GetCurrentProcess(),
    nullptr, 1 << 30,  // 1 GB
    MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE, 1);   // node 1

// touch memory
::memset(p, 0xff, 1 << 30);

NumaInfo();
::VirtualFree(p, 0, MEM_RELEASE);
```

Here is an example output:

```
NUMA nodes: 2
Node 0: Processor Group:  0, Affinity: 0x0000000F
        Available memory: 3587332 KB
Node 1: Processor Group:  0, Affinity: 0x000000F0
        Available memory: 3287952 KB
NUMA nodes: 2
Node 0: Processor Group:  0, Affinity: 0x0000000F
        Available memory: 3584884 KB
Node 1: Processor Group:  0, Affinity: 0x000000F0
        Available memory: 2243764 KB
```

Notice the reduced amount of physical memory on node 1.

Testing NUMA-related code is problematic on non-NUMA systems, as there is just one node. One way to get around this is to use a virtualization technology, such as Hyper-V, to simulate NUMA nodes. For Hyper-V, a virtual machine's settings in the CPU node can be used to configure NUMA nodes (note that you must disable *Dynamic Memory* to make this work).

The `VirtualAlloc2` Function

The `VirtualAlloc2` function, introduced in Windows 10 version 1803 (RS4), as a possible replacement for the various other `VirtualAlloc` variants. It combines the power of all of them, so that a single call can use a different process than the current one, a preferred NUMA node, specific memory alignment, AWE and a selected memory partition (a semi-documented entity beyond the scope of this book). Its prototype is as follows:

```
PVOID VirtualAlloc2(
    _In_opt_ HANDLE Process,
    _In_opt_ PVOID BaseAddress,
    _In_ SIZE_T Size,
    _In_ ULONG AllocationType,
    _In_ ULONG PageProtection,
    _Inout_ MEM_EXTENDED_PARAMETER* ExtendedParameters,
    _In_ ULONG ParameterCount);
```

The first 5 parameters are identical to `VirtualAllocEx`. The last two parameters are an optional array of `MEM_EXTENDED_PARAMETER` structures, each one specifying some extra attribute related to the call. The number of such structures is provided by the last parameter. Here is what `MEM_EXTENDED_PARAMETER` looks like:

```
typedef struct DECLSPEC_ALIGN(8) MEM_EXTENDED_PARAMETER {
    struct {
        DWORD64 Type : MEM_EXTENDED_PARAMETER_TYPE_BITS;
        DWORD64 Reserved : 64 - MEM_EXTENDED_PARAMETER_TYPE_BITS;
    } DUMMYSTRUCTNAME;

    union {
        DWORD64 ULong64;
        PVOID Pointer;
        SIZE_T Size;
        HANDLE Handle;
        DWORD ULong;
    } DUMMYUNIONNAME;
} MEM_EXTENDED_PARAMETER, *PMEM_EXTENDED_PARAMETER;
```

This structure is really a union where just one member is valid, based on the `Type` member, which is an enumeration selecting the valid member inside the union:

```
typedef enum MEM_EXTENDED_PARAMETER_TYPE {
    MemExtendedParameterInvalidType = 0,
    MemExtendedParameterAddressRequirements,
    MemExtendedParameterNumaNode,
    MemExtendedParameterPartitionHandle,
    MemExtendedParameterUserPhysicalHandle,
    MemExtendedParameterAttributeFlags,
    MemExtendedParameterMax
} MEM_EXTENDED_PARAMETER_TYPE;
```

For example, setting a preferred NUMA node similar to the example given in the section “NUMA” can be accomplished with `VirtualAlloc2` like so:

```
MEM_EXTENDED_PARAMETER param = { 0 };
param.Type = MemExtendedParameterNumaNode;
param.ULong = 1;    // NUMA node

auto p = ::VirtualAlloc2(
    ::GetCurrentProcess(), // NULL also works for current process
    nullptr, 1 << 30,
    MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE,
    &param, 1);
```

Consult the documentation for some more examples.

Summary

In this chapter, we examined the various APIs provided by Windows for allocating and otherwise managing memory. Memory is one of the fundamental resources in a computer system, where almost everything else is mapped to memory usage in one way or another.

In the next chapter, we'll dive into Memory Mapped Files and their capabilities of mapping files to memory and sharing memory between processes.

Chapter 14: Memory Mapped Files

A *Memory Mapped File*, called *Section* in kernel terminology, is an object that provides the ability to seamlessly map a file's contents to memory. Additionally, it can be used to share memory efficiently between processes. These capabilities provides several benefits, which will be explored in this chapter.

In this chapter:

- **Introduction**
- **Mapping Files**
- **Sharing Memory**
- **The *Micro Excel 2* Application**
- **Other Memory Mapping Functions**
- **Data Coherence**

Introduction

File mapping objects are everywhere in Windows. When an image file (EXE or DLL) is loaded, it's mapped to memory using memory-mapped files. With this mapping, access to the underlying file is done indirectly, by accessing memory through standard pointers. When code needs to execute within the image, the initial access causes a page fault exception, which the memory manager takes care of by reading the data from the file and placing it in physical memory, before fixing the appropriate page tables used to map this memory, at which point the calling thread can access the code/data. This is all transparent to the application.

In chapter 11 we looked at various I/O related APIs to read and/or write data, such as `ReadFile` and `WriteFile`. Imagine that some code needs to search a file for some data, and that search requires going back and forth in the file. With I/O APIs, this is inconvenient at best, involving multiple calls to `ReadFile` (with a buffer allocated beforehand) and `SetFilePointer(Ex)`. On the other hand, if a "pointer" to the file was available, then moving and performing operations with the file is much easier: no buffers to allocate, no `ReadFile` calls, and any file pointer changes just translate to pointer arithmetic. All other common memory functions, such as `memcpy`, `memset`, etc. work just as well with memory-mapped files.

Mapping Files

The steps required to map an existing file is first to open it with a normal `CreateFile` call. The access mask to `CreateFile` must be appropriate to the access required to the file, such as read and/or write. Once such a file is open, calling `CreateFileMapping` creates the file mapping object where the file handle can be provided:

```
HANDLE CreateFileMapping(
    _In_ HANDLE hFile,
    _In_opt_ LPSECURITY_ATTRIBUTES lpFileMappingAttributes,
    _In_ DWORD flProtect,
    _In_ DWORD dwMaximumSizeHigh,
    _In_ DWORD dwMaximumSizeLow,
    _In_opt_ LPCTSTR lpName);
```

The first parameter to `CreateFileMapping` is a file handle. If this MMF object is to be map a file, then a valid file handle should be provided. If the file mapping object is going to be used for creating shared memory backed up by a page file, then `INVALID_HANDLE_VALUE` should be specified, indicating page file usage. We'll examine shared memory not backed up by a specific file later in this chapter.

Next, the standard `SECURITY_ATTRIBUTES` pointer can be set, typically to `NULL`. The `flProtect` parameter indicates what page protection should be used when physical storage is (later) used for this file mapping object. This should be the most permissive page protection that is needed later. Common examples are `PAGE_READWRITE` and `PAGE_READONLY`. Table 14-1 summarizes the valid values for `flProtect` and the corresponding valid values when creating/opening the file used with the file mapping object.

Table 14-1: Protection flags for `CreateFileMapping`

MMF protection flag	Minimum access flags for the file	Comments
`PAGE_READONLY`	`GENERIC_READ`	Writing to memory/file is not permitted
`PAGE_READWRITE`	`GENERIC_READ` and `GENERIC_WRITE`	
`PAGE_WRITECOPY`	`GENERIC_READ`	Equivalent to `PAGE_READONLY`
`PAGE_EXECUTE_READ`	`GENERIC_READ` and `GENERIC_EXECUTE`	
`PAGE_EXECUTE_READWRITE`	`GENERIC_READ`, `GENERIC_WRITE` and `GENERIC_EXECUTE`	
`PAGE_EXECUTE_WRITECOPY`	`GENERIC_READ` and `GENERIC_EXECUTE`	Equivalent to `PAGE_EXECUTE_READ`

One of the values in table 14-1 can be combined with some flags, described below (there are more flags, but the list is of the officially documented ones):

- `SEC_COMMIT` - for page-file backed MMF only (`hFile` is `INVALID_HANDLE_VALUE`), indicates that all mapped memory must be committed when a view is mapped into the process address. This flag is mutually exclusive with `SEC_RESERVE`, and is the default if neither is specified. In any case, it has no effect for MMF backed up by a specific file.
- `SEC_RESERVE` - the opposite of `SEC_COMMIT`. Any view is initially reserved, so that the actual committing must be performed explicitly with `VirtalAlloc` call(s).

- `SEC_IMAGE` - specifies that the file provided is a PE file. It should be combined with `PAGE_READONLY` protection, but the mapping is done according to the sections in the PE. This flag cannot be combined with any other flag.
- `SEC_IMAGE_NO_EXECUTE` - similar to `SEC_IMAGE`, but the PE is not intended for execution, just mapping.
- `SEC_LARGE_PAGES` - valid for page-file backed MMF only. Indicates the usage of large pages when mapped. This requires the `SeLockMemoryPrivilege` as described in chapter 13. It also requires any view into the MMF and the view size to be multiple of the large page size. This flag must be combined with `SEC_COMMIT`.
- `SEC_NOCACHE` and `SEC_WRITECOMBINE` - rarely used flags, typically because a device driver requires it for proper operation.

The next two parameters to `CreateFileMapping` specify the MMF size using two 32-bit values that should be treated as a 64-bit value. If the MMF is to map an existing file with read-only access, set both values to zero, which effectively set the size of the MMF to the size of the file.

If the file in question is to be written to, set the size to the maximum size of the file. Once set, the file cannot grow beyond this size, and in fact its size will immediately grow to the specified size. If the MMF is backed by a page file, then the size indicates the sized of the memory block, where the page files in the system must be able to accommodate at MMF creation time.

The final parameter to `CreateFileMapping` is the object's name. It can be `NULL`, or it can be named, just like other named object types (e.g. events, semaphores, mutexes). Given a name, it's easy to share the object with other processes. Finally, the function returns a handle to the memory-mapped file object, or `NULL` in case of failure.

The following example creates a memory-mapped file object based on a data file, for read access only (error handling omitted):

```
HANDLE hFile = ::CreateFile(L"c:\\mydata.dat", GENERIC_READ,
    FILE_SHARE_READ, nullptr, OPEN_EXISTING, 0, nullptr);
HANDLE hMemFile = ::CreateFileMapping(hFile, nullptr,
    PAGE_READONLY, 0, 0, nullptr);
::CloseHandle(hFile);
```

The last line may be alarming. Is it OK to close the file handle? Wouldn't that close the file, making it inaccessible by the file mapping object? As it turns out, the MMF cannot rely on the client to keep the file handle open long enough, and it duplicates it to make sure the file is not closed. This means closing the file handle is the right thing to do.

Once a memory-mapped file object is created, the process can use the returned handle to map all or part of the file's data into its address space, by calling `MapViewOfFile`:

```
LPVOID MapViewOfFile(
    _In_ HANDLE hFileMappingObject,
    _In_ DWORD dwDesiredAccess,
    _In_ DWORD dwFileOffsetHigh,
    _In_ DWORD dwFileOffsetLow,
    _In_ SIZE_T dwNumberOfBytesToMap);
```

`MapViewOfFile` takes the MMF handle and maps the file (or part of it) into the process address space. `dwDesiredAccess` can be a combination of one or more flags described in table 14-2.

Table 14-2: Mapping flags for `MapViewOfFile`

Desired access	Description
`FILE_MAP_READ`	map for read access
`FILE_MAP_WRITE`	map for write access
`FILE_MAP_EXECUTE`	map for execute access
`FILE_MAP_ALL_ACCESS`	equivalent to `FILE_MAP_WRITE` when used with `MapViewOfFile`
`FILE_MAP_COPY`	copy-on-write access. Any write gets a private copy, that is discarded when the view is unmapped
`FILE_MAP_LARGE_PAGES`	maps using large pages
`FILE_MAP_TARGETS_INVALID`	sets all locations in the view as invalid targets for *Control Flow Guard* (CFG). The default is that the view is a valid target for CFG (see chapter 16 for more on CFG)

The `dwFileOffsetHigh` and `dwFileOffsetLow` form the 64-bit offset from which to begin mapping. The offset must be a multiple of the allocation granularity (64 KB on all Windows versions and architectures). The last parameter, `dwNumberOfBytesToMap` specifies how many bytes to map starting from the offset. Setting this to zero maps to the end of the file mapping.

The function returns the virtual address of the mapped memory in the caller's address space. The caller can use the pointer with all standard memory operations (subject to the mapping constraints). Once the mapped view is no longer needed, it should be unmapped with `UnmapViewOfFile`:

```
BOOL UnmapViewOfFile(_In_ LPCVOID lpBaseAddress);
```

`lpBaseAddress` is the same value returned from `MapViewOfFile`. Once unmapped, the memory pointed by `lpBaseAddress` is no longer valid, and any access causes an access violation.

The *filehist* Application

The command-line application *filehist* (File Histogram) counts the number of occurrences of each byte (0 to 255) in a file, effectively building a histogram distribution of the byte values in the file. The application is built by using a memory-mapped file, so that views are mapped into the process address space and then

the values are accessed with normal pointers. The application can deal with files of any size, but mapping limited views into the process address space, processing the data, unmapping, and then mapping the next chunk in the file.

Running the application with no arguments shows the following:

```
C:\>filehist.exe
Usage:  filehist [view size in MB] <file path>
        Default view size is 10 MB
```

The view size is configurable, where the default is 10 MB (no special reason for this value). Here is an example with a large file and the default view size:

```
C:\>filehist.exe file1.dat
File size: 938857496 bytes
Using view size: 10 MB
Mapping offset: 0x0, size: 0xA00000 bytes
Mapping offset: 0xA00000, size: 0xA00000 bytes
Mapping offset: 0x1400000, size: 0xA00000 bytes
Mapping offset: 0x1E00000, size: 0xA00000 bytes
...
Mapping offset: 0x36600000, size: 0xA00000 bytes
Mapping offset: 0x37000000, size: 0xA00000 bytes
Mapping offset: 0x37A00000, size: 0x55D418 bytes
0xB3:     445612 ( 0.05 %)
0x9E:     460881 ( 0.05 %)
0x9F:     469939 ( 0.05 %)
0x9B:     496322 ( 0.05 %)
0x96:     546899 ( 0.06 %)
0xB5:     555019 ( 0.06 %)
...
0x0F:   11226199 ( 1.20 %)
0x7F:   11755158 ( 1.25 %)
0x01:   14336606 ( 1.53 %)
0x8B:   14824094 ( 1.58 %)
0x48:   20481378 ( 2.18 %)
0xFF:   72242071 ( 7.69 %)
0x00:  342452879 (36.48 %)
```

The value zero is clearly the dominant one. If we increase the view size to 400 MB, this is what we get:

```
C:\>filehist.exe 400 file1.dat
File size: 938857496 bytes
Using view size: 400 MB
Mapping offset: 0x0, size: 0x19000000 bytes
Mapping offset: 0x19000000, size: 0x19000000 bytes
Mapping offset: 0x32000000, size: 0x5F5D418 bytes
0xB3:     445612 ( 0.05 %)
0x9E:     460881 ( 0.05 %)
...
0x48:   20481378 ( 2.18 %)
0xFF:   72242071 ( 7.69 %)
0x00:  342452879 (36.48 %)
```

The first thing done in `main` is process some of the command line arguments:

```
int wmain(int argc, const wchar_t* argv[]) {
    if (argc < 2) {
        printf("Usage:\tfilehist [view size in MB] <file path>\n");
        printf("\tDefault view size is 10 MB\n");
        return 0;
    }

    DWORD viewSize = argc == 2 ? (10 << 20) : (_wtoi(argv[1]) << 20);
    if (viewSize == 0)
        viewSize = 10 << 20;
```

Next, we need an array where the values and counts are stored:

```
struct Data {
    BYTE Value;
    long long Count;
};

Data count[256] = { 0 };
for (int i = 0; i < 256; i++)
    count[i].Value = i;
```

Now we can open the file, get its size, and create a file mapping object pointing to that file:

```
HANDLE hFile = ::CreateFile(argv[argc - 1], GENERIC_READ,
    FILE_SHARE_READ, nullptr, OPEN_EXISTING, 0, nullptr);
if (hFile == INVALID_HANDLE_VALUE)
    return Error("Failed to open file");

LARGE_INTEGER fileSize;
if (!::GetFileSizeEx(hFile, &fileSize))
    return Error("Failed to get file size");

HANDLE hMapFile = ::CreateFileMapping(hFile, nullptr, PAGE_READONLY,
    0, 0, nullptr);
if (!hMapFile)
    return Error("Failed to create MMF");

::CloseHandle(hFile);
```

The file is opened for read-only access, since there is no intention of changing anything in the file. The MMF is opened with `PAGE_READONLY` access, compatible with the file's `GENERIC_READ` access. Next we need to loop a number of times, depending on the file size and selected view size and process the data:

```
auto total = fileSize.QuadPart;
printf("File size: %llu bytes\n", fileSize.QuadPart);
printf("Using view size: %u MB\n", (unsigned)(viewSize >> 20));

LARGE_INTEGER offset = { 0 };
while (fileSize.QuadPart > 0) {
    auto mapSize = (unsigned)min(viewSize, fileSize.QuadPart);
    printf("Mapping offset: 0x%llX, size: 0x%X bytes\n",
        offset.QuadPart, mapSize);

    auto p = (const BYTE*)::MapViewOfFile(hMapFile, FILE_MAP_READ,
        offset.HighPart, offset.LowPart, mapSize);
    if (!p)
        return Error("Failed in MapViewOfFile");

    // do the work
    for (DWORD i = 0; i < mapSize; i++)
        count[p[i]].Count++;
    ::UnmapViewOfFile(p);

    offset.QuadPart += mapSize;
    fileSize.QuadPart -= mapSize;
}
```

```
::CloseHandle(hMapFile);
```

As long as there are still bytes to process, `MapViewOfFile` is called to map a portion of the file from the current offset with the minimum of the view size and the bytes left to process. After processing the data, the view is unmapped, the offset incremented, the remaining bytes decremented, and the loop repeats.

The final act is displaying the results. The data array is first sorted by the count, and then everything is displayed in order:

```
// sort by ascending order
std::sort(std::begin(count), std::end(count),
    [](const auto& c1, const auto& c2) {
    return c2.Count > c1.Count;
    });

// display results
for (const auto& data : count) {
    printf("0x%02X: %10llu (%5.2f %%)\n", data.Value, data.Count,
        data.Count * 100.0 / total);
}
```

Static C++ arrays can be sorted with `std::sort` just like vectors. The global `std::begin` and `std::end` functions are needed to provide iterators for arrays because there are no methods in C++ arrays.

Sharing Memory

Processes are isolated from one another, so that each has its own address space, its own handle table, etc. Most of the time, this is what we want. However, there are cases where data needs to be shared in some way between processes. Windows provides many mechanisms for *Interprocess Communication* (IPC), including the *Component Object Model* (COM), windows messages, sockets, pipes, mailslots, *Remote Procedure Calls* (RPC), clipboard, *Dynamic Data Exchange* (DDE) and more. Each has its strengths and weaknesses, but the common theme with all the above is that data (memory) must be copied from one process to another.

Memory-mapped files are yet another mechanism for IPC, and it's the fastest of them all because there is no copying going around (in fact, some of the other IPC mechanisms use memory-mapped files under the covers when communicating between processes on the same machine). One process writes the data to the shared memory, and all other processes that have handles to the same file mapping object can see the memory immediately - there is no copying going around because each process maps the same memory into its own address space.

Sharing memory is based on multiple processes having access to the same file mapping object. The object can be shared in any of the three ways described in chapter 2. The simplest is to use a name for the file mapping object. The shared memory itself can be backed up by a specific file (valid file handle to `CreateFileMapping`), in which case the data is available even after the file mapping object is destroyed, or backed up by the paging file, in which case the data is discarded once the file mapping object is destroyed. Both options work essentially in the same way.

We'll start by using the *Basic Sharing* application from chapter 2. There we looked at sharing capabilities based on an object's name, but now we can dig into the details of sharing itself. Figure 14-1 shows two instances of the application running, where writing in one process and reading in another process shows the same data since these use the same file mapping object.

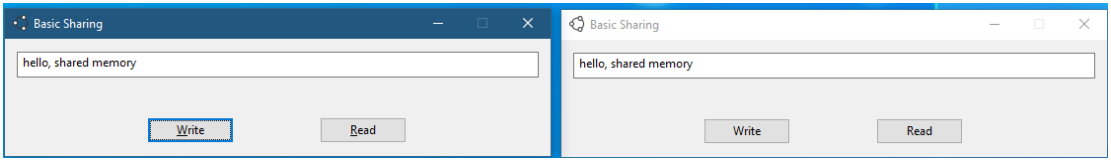


Figure 14-1: Multiple instances of *Basic Sharing*

In `CMainDlg::OnInitDialog` the file mapping object is created, backed up by a page file:

```
m_hSharedMem = ::CreateFileMapping(INVALID_HANDLE_VALUE, nullptr,
    PAGE_READWRITE, 0, 1 << 12, L"MySharedMemory");
```

The MMF is created for read/write access, and its size is 4 KB. Its name ("MySharedMemory") is going to be the method used for sharing the object with other processes. The first time `CreateFileMapping` is called with that object name, the object is created. Any subsequent calls using the same name just get a handle to the already existing file mapping object. This means the other parameters are not really used. For example, a second caller cannot specify a different size for the memory - the initial creator determines the size.

Alternatively, a process may want to open a handle to an existing file mapping object and fail if it does not exist. This is the role of `OpenFileMapping`:

```
HANDLE OpenFileMapping(
    _In_ DWORD dwDesiredAccess,
    _In_ BOOL bInheritHandle,
    _In_ LPCTSTR lpName);
```

The desired access parameter is a combination of access masks from table 14-2.
`bInheritHandle` specifies whether the returned handle is inheritable or not (see chapter 2 for more on handle inheritance). Finally, `lpName` is the named MMF to locate. The function fails if the there is no file mapping object with the given name (returning `NULL`).

In most cases, using `CreateFileMapping` is more convenient, especially if sharing is intended by multiple processes based off the same executable image - the first process creates the object and all subsequent processes just get handles to the existing object - no need to synchronize creation vs. opening.

With a file mapping object handle in place, writing to the memory is done by the following function:

```
LRESULT CMainDlg::OnWrite(WORD, WORD, HWND, BOOL &) {
    void* buffer = ::MapViewOfFile(m_hSharedMem, FILE_MAP_WRITE,
        0, 0, 0);
    if (!buffer) {
       AtlMessageBox(m_hWnd, L"Failed to map memory", IDR_MAINFRAME);
       return 0;
    }

    CString text;
    GetDlgItemText(IDC_TEXT, text);
    ::wcscpy_s((PWSTR)buffer, text.GetLength() + 1, text);

    ::UnmapViewOfFile(buffer);

    return 0;
}
```

The call to `MapViewOfFile` is not much different than the call from the *filehist* application. `FILE_MAP_WRITE` is used to gain write access to the mapped memory. The offset is zero and the mapping size is specified as zero as well, which means mapping all the way to the end. Since the shared memory is only 4 KB in size, that's not an issue, and in any case everything is rounded up to the nearest page boundary. After the data is written, `UnmapViewOfFile` is called to unmap the view from the process address space.

Reading data is very similar, just using different flags for access:

```
LRESULT CMainDlg::OnRead(WORD, WORD, HWND, BOOL &) {
    void* buffer = ::MapViewOfFile(m_hSharedMem, FILE_MAP_READ,
        0, 0, 0);
    if (!buffer) {
        AtlMessageBox(m_hWnd, L"Failed to map memory", IDR_MAINFRAME);
        return 0;
    }

    SetDlgItemText(IDC_TEXT, (PCWSTR)buffer);
    ::UnmapViewOfFile(buffer);

    return 0;
}
```

We can create a new application, that can use the shared memory just the same. The *memview* application monitors changes to the data in the shared memory and displays any new data that appears.

First, the file mapping object must be open. In this case, I decided to use `OpenFileMapping`, because this is a monitoring application, and should not be able to determine the shared memory size or backup file:

```
int main() {
    HANDLE hMemMap = ::OpenFileMapping(FILE_MAP_READ, FALSE,
        L"MySharedMemory");
    if (!hMemMap) {
        printf("File mapping object is not available\n");
        return 1;
    }
```

Next, we need to map the memory into the process address space:

```
WCHAR text[1024] = { 0 };
auto data = (const WCHAR*)::MapViewOfFile(hMemMap, FILE_MAP_READ,
    0, 0, 0);
if (!data) {
    printf("Failed to map shared memory\n");
    return 1;
}
```

The "monitoring" is based on reading the data every certain period of time (1 second in the following code). The `text` local variable stores the current text from the shared memory. It's compared to the new data and updated if needed:

```
for (;;) {
    if (::_wcsicmp(text, data) != 0) {
        // text changed, update and display
        ::wcscpy_s(text, data);
        printf("%ws\n", text);
    }
    ::Sleep(1000);
}
```

The loop is infinite in this simple example, but it's easy to come up with an appropriate exit condition. You can try it out and watch the text updated whenever any of the running *Basic Sharing* instances writes a new string to the shared memory.

If you open *Process Explorer* and look for one of the handles to the file mapping object, you'll find the total handles to the MMF object reflect the total number of processes using the shared memory. If you have two instances of *Basic Sharing* and one instance of *memview*, then three handles are expected (figure 14-2).

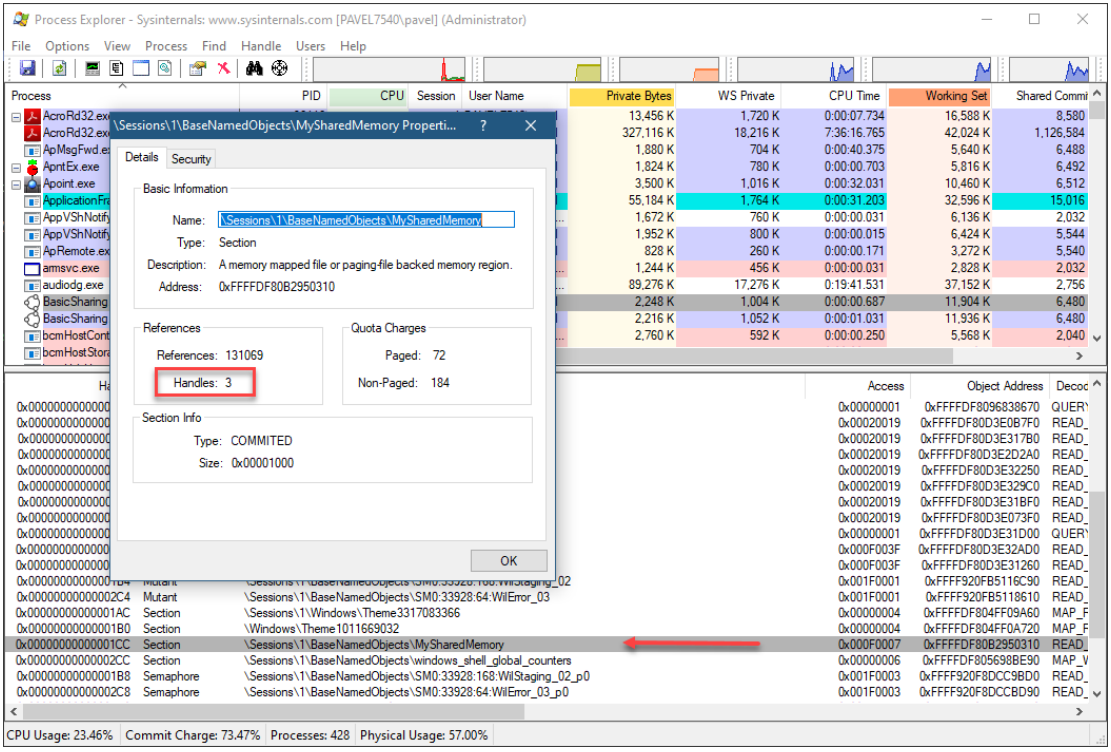


Figure 14-2: Object shared in *Process Explorer*

Sharing Memory with File Backing

The *Basic Sharing+* application demonstrates the usage of shared memory possibly backed up by a file other than the paging file. The application is based on the *Basic Sharing* application. Figure 14-3 shows the application's window at startup.

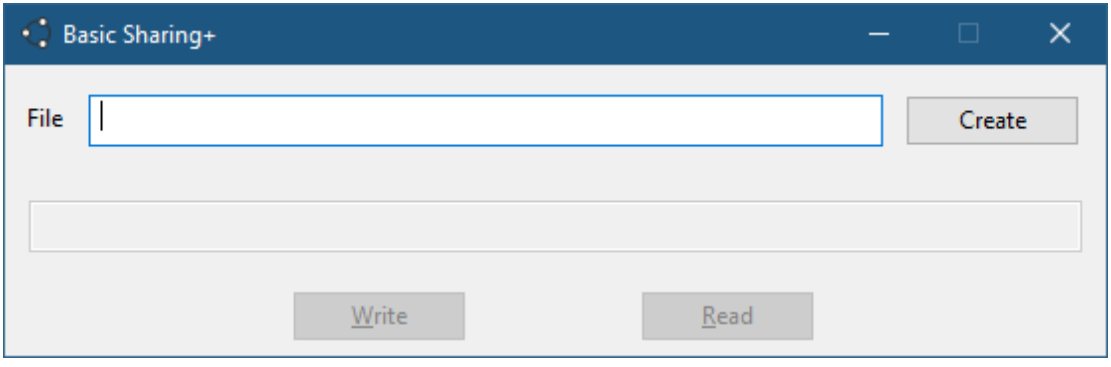


Figure 14-3: The *Basic Application+* window

You can specify a file to use, or leave the edit box empty, in which case the page file will be used as backup (equivalent to the *Basic Sharing* application). Clicking the *Create* button creates the file mapping object.

If the file is specified and exists, its size determines the size of the file mapping object. If the file does not exist, it's created with the size specified in `CreateFileMapping` (4 KB, just like in *Basic Sharing*). The file size itself becomes 4 KB immediately.

Once the file mapping object is created, the UI focus changes to the data edit box and the read and write buttons, just like in *Basic Sharing*. If you now launch another instance of *Basic Sharing+*, it will automatically go to the editing mode, disabling the *Create* button. This is done by calling `OpenFileMapping` when the process is launched. If the file mapping object exists, there is no point in allowing the user to select a file, since that has no effect.

The `CMainDlg::OnInitDialog` attempts to open the file mapping object if it exists:

```
m_hSharedMem = ::OpenFileMapping(FILE_MAP_READ | FILE_MAP_WRITE,
    FALSE, L"MySharedMemory");
if (m_hSharedMem)
    EnableUI();
```

If this succeeds, `EnableUI` is called to disable the file name edit box and *Create* button and enable the data edit box and the *Read* and *Write* buttons. Clicking the *Create* button (if enabled), creates the file mapping object as requested:

```
LRESULT CMainDlg::OnCreate(WORD, WORD, HWND, BOOL&) {
    CString filename;
    GetDlgItemText(IDC_FILENAME, filename);
    HANDLE hFile = INVALID_HANDLE_VALUE;
    if (!filename.IsEmpty()) {
        hFile = ::CreateFile(filename, GENERIC_READ | GENERIC_WRITE, 0,
            nullptr, OPEN_ALWAYS, 0, nullptr);
        if (hFile == INVALID_HANDLE_VALUE) {
            AtlMessageBox(*this, L"Failed to create/open file",
                IDR_MAINFRAME, MB_ICONERROR);
            return 0;
        }
    }

    m_hSharedMem = ::CreateFileMapping(hFile, nullptr, PAGE_READWRITE,
        0, 1 << 12, L"MySharedMemory");
    if (!m_hSharedMem) {
        AtlMessageBox(m_hWnd, L"Failed to create shared memory",
            IDR_MAINFRAME, MB_ICONERROR);
        EndDialog(IDCANCEL);
    }

    if (hFile != INVALID_HANDLE_VALUE)
        ::CloseHandle(hFile);
```

```
    EnableUI();

    return 0;
}
```

If a file name is specified, `CreateFile` is called to open or create the file. It uses the `OPEN_ALWAYS` flag that means "create the file if it does not exist, or open otherwise". The file handle is passed to `CreateFileMapping` to create the file mapping object. Finally, the file handle is closed (if previously opened) and `EnableUI` is called to put the application in the data editing mode.

The *Micro Excel 2* Application

The *Micro Excel* application from chapter 13 demonstrated how to reserve a large region of memory and then only commit those pages that are being actively used by the application. We can combine this approach with a memory-mapped file, so that the memory can also be shared efficiently with other processes. Figure 14-3 shows the application in action.

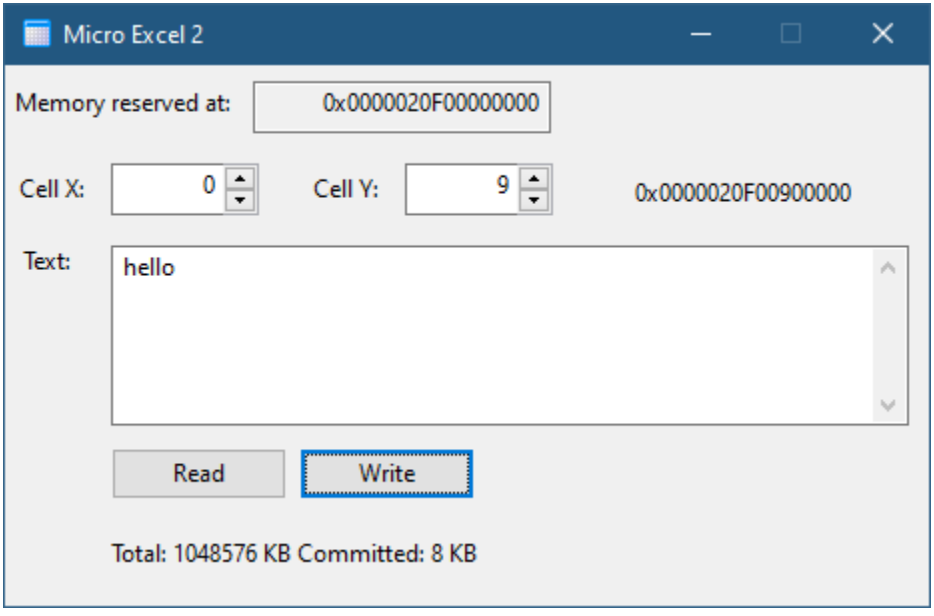


Figure 14-4: The *Micro Excel 2* application

The secret to mapping a large region of memory without committing when `MapViewOfFile` is called, is by using the `SEC_RESERVE` flag with `CreateFileMapping`. This causes the mapped region to be reserved only, meaning direct access causes an access violation. In order to commit pages, the `VirtualAlloc` function needs to be called.

Let's examine the changes we need to make to *Micro Excel* to support this functionality with file mapping. First, the file mapping object creation:

```
bool CMainDlg::AllocateRegion() {
    m_hSharedMem = ::CreateFileMapping(INVALID_HANDLE_VALUE, nullptr,
        PAGE_READWRITE | SEC_RESERVE, TotalSize >> 32, (DWORD)TotalSize,
        L"MicroExcelMem");

    if (!m_hSharedMem) {
        AtlMessageBox(nullptr, L"Failed to create shared memory",
            IDR_MAINFRAME, MB_ICONERROR);
        EndDialog(IDCANCEL);
        return false;
    }

    m_Address = ::MapViewOfFile(m_hSharedMem,
        FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, TotalSize);
    CString addr;
    addr.Format(L"0x%p", m_Address);
    SetDlgItemText(IDC_ADDRESS, addr);
    SetDlgItemText(IDC_CELLADDR, addr);

    return true;
}
```

`AllocateRegion` is called when the dialog is initialized. The call to `CreateFileMapping` uses the page file as a backup (this is the only scenario supported with `SEC_RESERVE`), and requests the `SEC_RESERVE` flag along with `PAGE_READWRITE`. The file mapping object is given a name for easy sharing with other processes.

Next, `MapViewOfFile` is called to map the entire shared memory (`TotalSize`=1 GB). It would of course been possible to map just part of this memory, and this is in fact a very good idea for 32-bit processes, where the address space range is somewhat limited. Because of the `SEC_RESERVE` flag, the entire region is reserved, rather than committed.

Writing and reading data from any cell is done in exactly the same way as with the original *Micro Excel*: An initial write attempt causes an access violation exception, which is caught, where `VirtualAlloc` is called to explicitly commit the page where the particular cell falls, and then returns `EXCEPTION_CONTINUE_EXECUTION` to tell the processor to try the access again. The code for writing and handling the exception is repeated here for convenience:

```
LRESULT CMainDlg::OnWrite(WORD, WORD, HWND, BOOL&) {
    int x, y;
    auto p = GetCell(x, y);
    if(!p)
        return 0;

    WCHAR text[512];
    GetDlgItemText(IDC_TEXT, text, _countof(text));

    __try {
        ::wcscpy_s((WCHAR*)p, CellSize / sizeof(WCHAR), text);
    }
    __except (FixMemory(p, GetExceptionCode())) {
        // nothing to do: this code is never reached
    }

    return 0;
}

int CMainDlg::FixMemory(void* address, DWORD exceptionCode) {
    if (exceptionCode == EXCEPTION_ACCESS_VIOLATION) {
        ::VirtualAlloc(address, CellSize, MEM_COMMIT,
            PAGE_READWRITE);
        return EXCEPTION_CONTINUE_EXECUTION;
    }

    return EXCEPTION_CONTINUE_SEARCH;
}
```

If you run a second *Micro Excel 2* application, you'll find that the same information is visible in the other processes, because it's the same mapped memory. However, notice that the address to which the 1 GB region is mapped in each process is unlikely to be the same. This is completely fine, and does not deter from the fact that both processes see the exact same memory (figure 14-5).

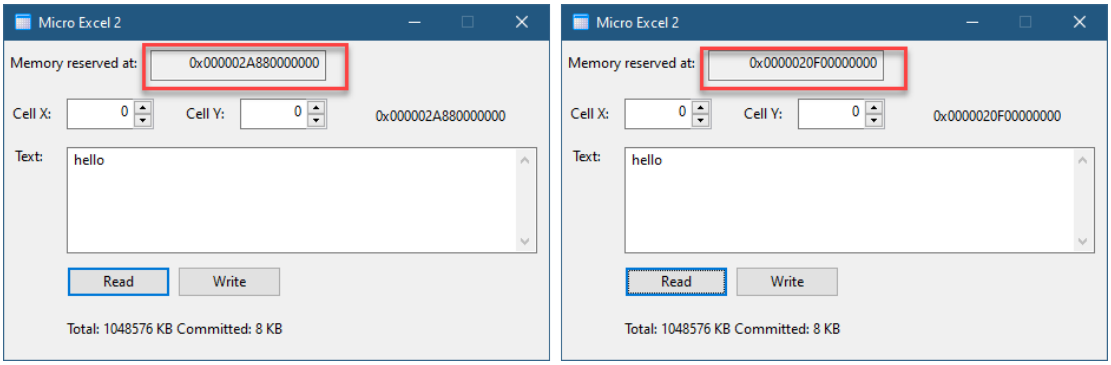


Figure 14-5: Two *Micro Excel 2* instances sharing memory

If you want to view this memory arrangement in *VMMap*, the correct memory "type" to look at is *Shareable* (figure 14-6).

Address	Type	Size	Committed	Private	Total WS
⊟ 0000020F00000000	Shareable	1,048,576 K	8 K		8 K
0000020F00000000	Shareable	4 K	4 K		4 K
0000020F00001000	Shareable	9,212 K			
0000020F00900000	Shareable	4 K	4 K		4 K
0000020F00901000	Shareable	1,039,356 K			
⊞ 0000020F443E0000	Shareable	32 K	16 K		12 K
⊞ 0000020F443F0000	Shareable	116 K	116 K		112 K

Figure 14-6: Page file-backed shared memory in *VMMap*

Other Memory Mapping Functions

An extended version of `MapViewOfFile` allows selecting the address to which mapping occurs:

```
LPVOID MapViewOfFileEx(
    _In_ HANDLE hFileMappingObject,
    _In_ DWORD dwDesiredAccess,
    _In_ DWORD dwFileOffsetHigh,
    _In_ DWORD dwFileOffsetLow,
    _In_ SIZE_T dwNumberOfBytesToMap,
    _In_opt_ LPVOID lpBaseAddress);
```

All parameters are identical to `MapViewOfFile`, except the last one. This is the requested address for the mapping. The address must be a multiple of the system's allocation granularity (64 KB). The function may fail if the specified address with the requesting mapping size is already occupied in the process address space. This is why it's almost always better to let the system locate an empty region by specifying `NULL` as the value, which makes the function identical to `MapViewOfFile`.

Why would you want to set up a specific address? One common case (for the system, at least) is in the case a PE file must be mapped by multiple processes (`SEC_IMAGE` flag). This is used for PE images, because

code can contain pointers (addresses) that reference another location in the PE image range. If the mapping is done to a different address, then some of the code needs to change. This normally happens in cases DLLs need to be relocated (discussed in the next chapter).

For data, it's also possible to store pointers that point to other locations in the mapped region, but that would not be a good idea, because `MapViewOfFileEx` may fail. It's better to store offsets in the data, as these are address-independent.

Another variation on `MapViewOfFile` is about selecting a preferred NUMA node for the physical memory used by the mapping:

```
LPVOID MapViewOfFileExNuma(
    _In_     HANDLE hFileMappingObject,
    _In_     DWORD dwDesiredAccess,
    _In_     DWORD dwFileOffsetHigh,
    _In_     DWORD dwFileOffsetLow,
    _In_     SIZE_T dwNumberOfBytesToMap,
    _In_opt_ LPVOID lpBaseAddress,
    _In_     DWORD nndPreferred);
```

`MapViewOfFileExNuma` extends `MapViewOfFileEx` with a preferred NUMA node (refer to chapter 13 for more on NUMA).

Windows 10 version 1703 (RS2) introduced `MapViewOfFile2`:

```
PVOID MapViewOfFile2(
    _In_ HANDLE FileMappingHandle,
    _In_ HANDLE ProcessHandle,
    _In_ ULONG64 Offset,
    _In_opt_ PVOID BaseAddress,
    _In_ SIZE_T ViewSize,
    _In_ ULONG AllocationType,
    _In_ ULONG PageProtection);
```

This function is implemented inline by calling the more expanded function `MapViewOfFileNuma2` by passing `NUMA_NO_PREFERRED_NODE` (-1) as the preferred NUMA node:

```
PVOID MapViewOfFileNuma2(
    _In_ HANDLE FileMappingHandle,
    _In_ HANDLE ProcessHandle,
    _In_ ULONG64 Offset,
    _In_opt_ PVOID BaseAddress,
    _In_ SIZE_T ViewSize,
    _In_ ULONG AllocationType,
    _In_ ULONG PageProtection,
    _In_ ULONG PreferredNode);
```

These functions and others that follow in this section require the import library *mincore.lib*. Currently the documentation specifies *kernel32.lib* incorrectly.

These functions add a second parameter (`hProcess`) identifying the process into which to map the view (the original functions always work on the current process). Of course, using `GetCurrentProcess` is perfectly legal. If the process in question is different, the handle must have the `PROCESS_VM_OPERATION` access mask. A nice bonus of these functions is the offset that can be specified as a single 64-bit number instead of the two 32-bit values in the original functions.

The `AllocationType` parameter can be 0 (for normal committed view), or `MEM_RESERVE` for reserving the view without committing. Also, `MEM_LARGE_PAGES` can be specified if large pages are to be used for mapping. In that case, the file mapping object would have to be created with the `SEC_LARGE_PAGES` flag and the caller must have the *SeLockMemoryPrivilege* privilege.

The rest of the parameters are the same as for `MapViewOfFileExNuma` (albeit in a different order). The returned address is valid in the target process address space only. This function can be useful when some memory is required to share with another process without that process having any knowledge about the file mapping object that is needed to open, the region to map. etc. This means the file mapping object can be created without a name, which makes it harder to interfere with. The only piece of information that needs to be passed to the target process is the resulting address, which can even be predefined if `BaseAddress` is not `NULL`. Passing a single pointer value to another process is much easier to do than more complex information. For example, a window message can be used, or even a shared variable from a DLL, as demonstrated in chapter 12.

Unmapping the mapped view can be done from within the target process normally with `UnmapViewOfFile` or from the mapping process with `UnmapViewOfFile2`:

```
BOOL UnmapViewOfFile2(
    _In_ HANDLE Process,
    _In_ PVOID BaseAddress,
    _In_ ULONG UnmapFlags
    );
```

`UnmapFlags` is typically zero, but can have two more values. Consult the documentation for the details. Another variant is `UnmapViewOfFileEx` that works like `UnmapViewOfFile2`, but always uses the calling process.

UWP processes that need to use `MapViewOfFile2` have their own version, `MapViewOfFile2FromApp`. As with similar functions in the `Virtual` family, if compiled in a UWP app, `MapViewOfFile2` is implemented inline to call `MapViewOfFile2FromApp`. Check the documentation for the details.

There is yet another `MapViewOfFile` variant, introduced in Windows 10 version 1803 (RS4):

```
PVOID MapViewOfFile3(
    _In_ HANDLE FileMapping,
    _In_opt_ HANDLE Process,
    _In_opt_ PVOID BaseAddress,
    _In_ ULONG64 Offset,
    _In_ SIZE_T ViewSize,
    _In_ ULONG AllocationType,
    _In_ ULONG PageProtection,
    _Inout_ MEM_EXTENDED_PARAMETER* ExtendedParameters,
    _In_ ULONG ParameterCount);
```

This is a "super function" combining the capabilities of the other variants, where the properties are given as an array of `MEM_EXTENDED_PARAMETER` structures. Refer to the discussion of `VirtualAlloc2` in chapter 13, as this is the same structure used there.

As perhaps expected, there is another variant for UWP processes - `MapViewOfFile3FromApp`, implemented similarly as previously described.

Finally, Windows 10 version 1809 (RS5) added a variant for `CreateFileMapping`:

```
HANDLE CreateFileMapping2(
    _In_ HANDLE File,
    _In_opt_ SECURITY_ATTRIBUTES* SecurityAttributes,
    _In_ ULONG DesiredAccess,
    _In_ ULONG PageProtection,
    _In_ ULONG AllocationAttributes,
    _In_ ULONG64 MaximumSize,
    _In_opt_ PCWSTR Name,
    _Inout_updates_opt_(ParameterCount)
        MEM_EXTENDED_PARAMETER* ExtendedParameters,
    _In_ ULONG ParameterCount);
```

This function seems to be currently undocumented, but it uses the same `MEM_EXTENDED_PARAMETER` structures. For example, specifying a preferred NUMA node for all mappings from this file mapping object can be done like so:

```
HANDLE hFile = ...;
MEM_EXTENDED_PARAMETER param = { 0 };
param.Type = MemExtendedParameterNumaNode;
param.ULong = 1;        // NUMA node 1
HANDLE hMemMap = ::CreateFileMapping2(hFile, nullptr, FILE_MAP_READ,
    PAGE_READONLY, 0, 0, nullptr, &param, 1);
```

Data Coherence

File mapping objects provide several guarantees in terms of data coherence.

- Multiple views of the same data/file, even from multiple processes, are guaranteed to be synchronized, since the various views are mapped to the same physical memory. The only exception is when mapping a remote file on the network. In that case, views from different machine may not be synchronized at all times. Views from the same machine continue to be synchronized.
- Multiple file mapping objects that map the same file are not guaranteed to be synchronized. Generally, it's a bad idea to map the same file with two or more file mapping objects. It's best to open the file in question for exclusive access so that no other access is possible on the file (at least if writing is intended).
- If a file is mapped by a file mapping object, and at the same time opened for normal I/O (`ReadFile`, `WriteFile`, etc.), the changes from I/O operations will not generally be immediately reflected in views mapped to the same locations in the file. This situation should be avoided.

Summary

Memory-mapped file objects are flexible and fast, providing shared memory capabilities, whether they map a specific file or just sharing memory backed up by the page file. They are very efficient, and I consider them one of my favorite features in Windows.

In the next chapter, we'll turn our attention to *Dynamic Link Libraries* (DLL), which are a crucial part of Windows.

Chapter 15: Dynamic Link Libraries

Dynamic Link Libraries (DLLs) were a fundamental part of Windows NT since its inception. The major motivation behind the existence of DLLs is the fact they can be easily shared between processes so that a single copy of a DLL is in RAM and all processes that need it can share the DLL's code. In those early days, RAM was much smaller than it is today, which made that memory saving very important. Even today these memory savings are significant, as a typical process uses dozens of DLLs.

DLLs have many uses today, many of which we'll examine in this chapter.

In this chapter:

Introduction

DLLs are *Portable Executable* (PE) files that can contain one or more of the following: code, data and resources. Every user-mode process uses subsystem DLLs, such as *kernel32.dll*, *user32.dll*, *gdi32.dll*, *advapi32.dll*, implementing the documented Windows API. And naturally, *Ntdll.Dll* is mandatory in every user-mode process, including native applications.

DLLs are libraries that can contain functions, global variables, and resources, such as menus, bitmaps, icons. Some functions (and types) can be exported by a DLL, so that they can be used directly by another DLL or executable that loads the DLL. A DLL can be loaded into a process implicitly, when the process starts up, or explicitly when the application calls the `LoadLibrary` or `LoadLibraryEx` function.

Building a DLL

We'll start by looking at how to build a DLL and export symbols. With Visual Studio, a new DLL project can be created by selecting the appropriate project template (figure 15-1).

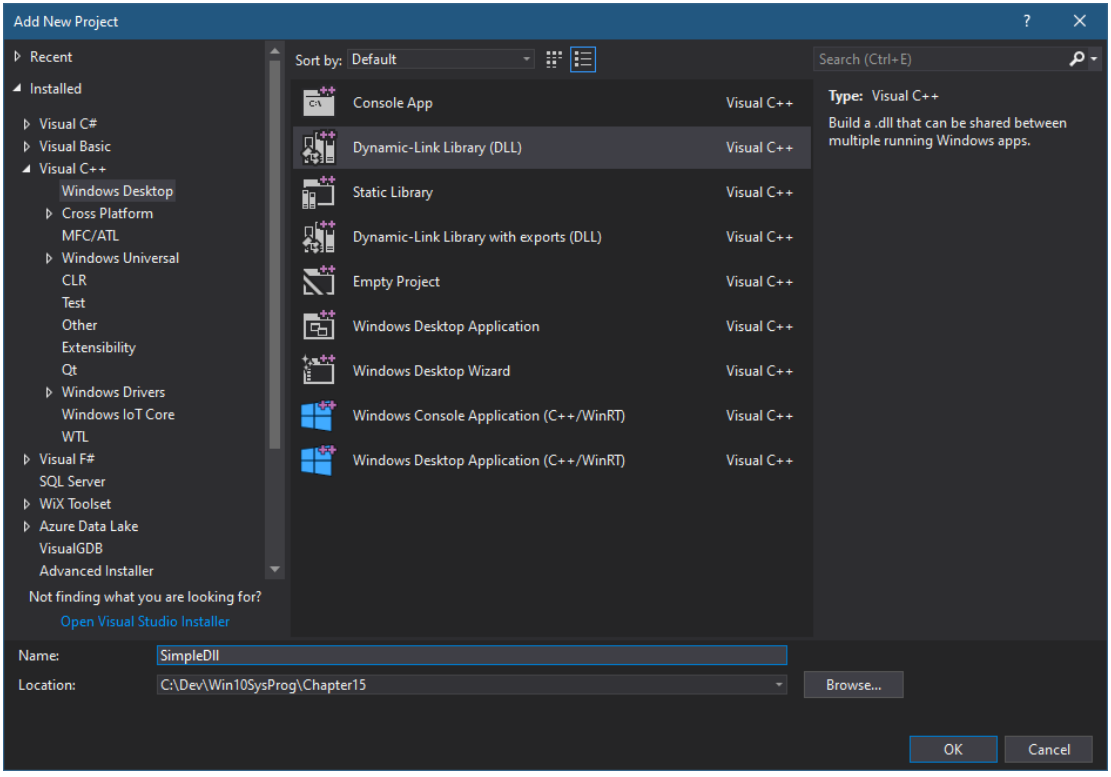


Figure 15-1: New DLL Project in Visual Studio

One of the templates indicates the DLL *exports symbols*, but any DLL template will do. The only fundamental change for a DLL project compared to an EXE project is the *Configuration Type* (figure 15-2) in the project's properties.

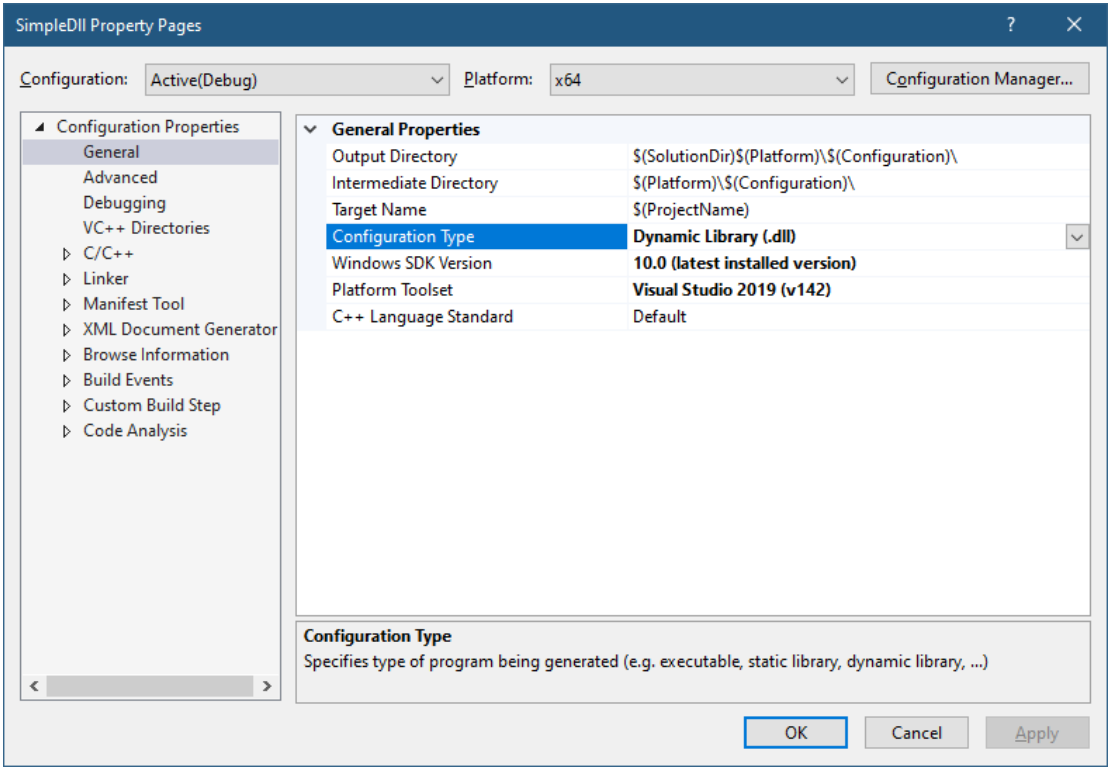


Figure 15-2: DLL Project properties in Visual Studio

A typical project created by Visual Studio would have the following files:

- *pch.h* and *pch.cpp* - precompiled header and implementation.
- *framework.h* - included by *pch.h* and should contain all "standard" Windows headers such as *Windows.h*. I typically delete this file and just put all Windows headers in *pch.h*.
- *dllmain.cpp* - includes the `DllMain` function (discussed later in this chapter).

At this point we can build the project successfully. However, the DLL is fairly useless. Most DLLs export some functionality to be called by other modules (other DLLs or an EXE). Let's add one function to the DLL, called `IsPrime`. First in a header file that can be included by users of the DLL:

```
// Simple.h

bool IsPrime(int n);
```

Then the implementation in a different file, since this should not be visible by users of the DLL:

```
// Simple.cpp

#include "pch.h"
#include "Simple.h"
#include <cmath>

bool IsPrime(int n) {
    int limit = (int)::sqrt(n);
    for (int i = 2; i <= limit; i++)
        if (n % i == 0)
            return false;
    return true;
}
```

The implementation is not important for the purposes of this section. The point is, we have some functionality in our DLL and we want to be able to use it. Let's add a console application project to the same solution in Visual Studio named *SimplePrimes*.

To gain access to the DLL's functionality, we add an include to *Simple.h* before our `main` function:

```
// SimplePrimes.cpp

#include "..\SimpleDll\Simple.h"
// other includes...
```

Let's add a simple test by calling `IsPrime`:

```
int main() {
    bool test = IsPrime(17);
    printf("%d\n", (int)test);

    return 0;
}
```

If we compile this, it compiles fine, but it fails to link with the dreaded "unresolved external" error: *SimplePrimes.obj : error LNK2019: unresolved external symbol "bool __cdecl IsPrime(int)" (?IsPrime@@YA_NH@Z) referenced in function _main*

The compiler finds the declaration of the function in *Simple.h*, so it's relatively happy. It also looks for an implementation, but cannot find one. Instead of complaining, it signals the linker that the implementation of `IsPrime` is missing, so perhaps the linker can resolve it.

How can the linker do so? The linker has a "global" view of the project and is aware of libraries that may be provided as binary pieces of compiled code. The linker, however, does not find anything in the list of libraries it knows about, and eventually gives up with an "unresolved external" error.

There are two pieces missing here: one is some reference to where to find the implementation. We'll add that by right-clicking the *References* node in the *SimplePrimes* project and selecting *Add Reference...* from the menu. The *Add Reference* dialog opens (figure 15-3).

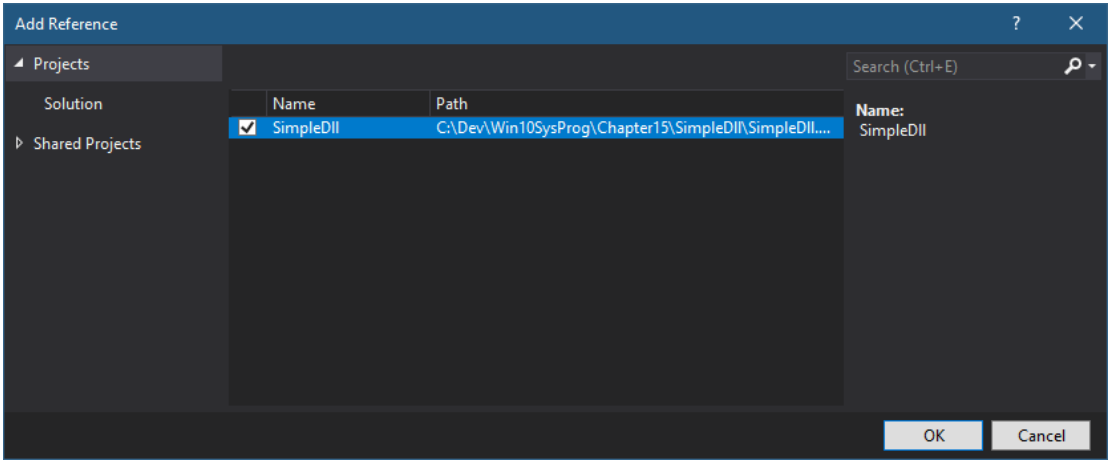


Figure 15-3: The *Add Reference* dialog box

You have to check the box for *Simple.Dll* and click OK. A node named *Simple.Dll* appears under the *References* node. Building the project now produces the same "unresolved external" error. This is where the second missing piece comes in: the *IsPrime* function must be exported. Here is one way to do it in *Simple.h* by extending the declaration of `IsPrime`:

```
__declspec(dllexport) bool IsPrime(int n);
```

In general, there are several `__declspec` types supported by Microsoft's compiler, as there is no standard way of exporting symbols from modules.

The C++ feature called *Modules* that is part of the C++ 20 standard attempts to address this issue. However, it's not necessarily geared towards DLLs. It's not fully implemented by the Visual C++ compiler at the time of this writing.

Now we can build the project and should build successfully. We can run *SimplePrimes* and get the expected result. Adding the `dllexport` specifier added the `IsPrime` function to the list of exported symbols. This still does not explain exactly why the linker was satisfied, and how the DLL was found at runtime. We'll get to these details in the next section.

You can now open any PE viewer tool and look at *Simple.Dll* and *SimplePrimes.exe*. For *Simple.Dll*, the `IsPrime` function should be listed as exported (figure 15-4 using my own *PE Explorer V2*). You can also get this information with the *Dumpbin.exe* command-line tool like so:

```
C:\>dumpbin /exports SimpleDll.dll
Microsoft (R) COFF/PE Dumper Version 14.26.28805.0
Copyright (C) Microsoft Corporation.  All rights reserved.

Dump of file c:\dev\Win10SysProg\Chapter15\Debug\SimpleDll.dll

File Type: DLL

  Section contains the following exports for SimpleDll.dll

    00000000 characteristics
    FFFFFFFF time date stamp
        0.00 version
           1 ordinal base
           1 number of functions
           1 number of names

    ordinal hint RVA      name

          1    0 000111F9 ?IsPrime@@YA_NH@Z = @ILT+500(?IsPrime@@YA_NH@Z)

  Summary
        1000 .00cfg
        1000 .data
        1000 .idata
        1000 .msvcjmc
        2000 .rdata
        1000 .reloc
        1000 .rsrc
        7000 .text
       10000 .textbss
```

Notice the `IsPrime`-ish symbol. It has some weird decorations, explained in the next section.

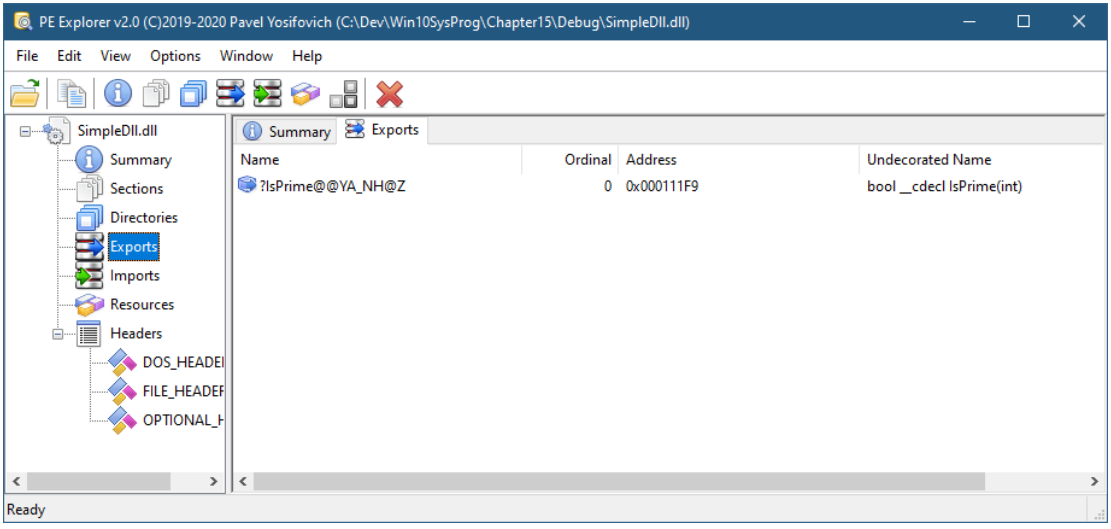


Figure 15-4: Exports in *Simple.dll*

Figure 15-5 shows the imports for *SimplePrimes.exe*. One of them is from *Simple.Dll* - the `IsPrime` function.

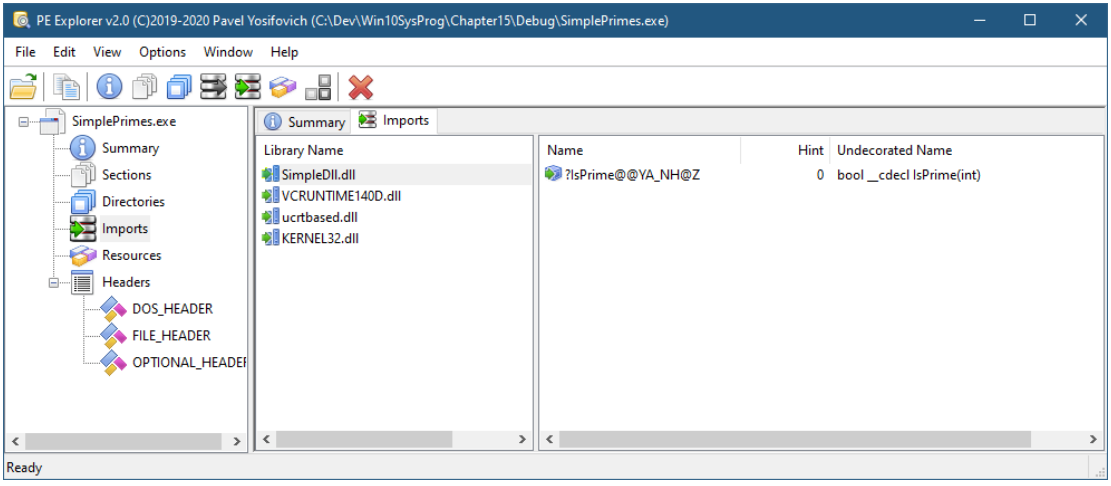


Figure 15-5: Imports in *SimplePrimes.exe*

Implicit and Explicit Linking

There are two fundamental ways to link against a DLL (so that its functionality can be used). The first and simplest is *implicit linking* (sometimes called static linking to DLLs), used in the previous section. The second is *explicit linking*, which is more involved but provides more control over the timing of loading and unloading of DLL.

Implicit Linking

When a DLL is generated, an accompanying file called an *import library* is generated by default as well. This file has the LIB extension and contains two pieces of information:

- The file name of the DLL (with no path)
- List of exported symbols (functions and variables)

When adding a reference to a DLL project in Visual Studio, as was done in the previous section, the import library generated by the DLL project is added as a dependency to the EXE project (or another DLL project that wants to use the DLL). Instead of adding a reference with Visual Studio (an option that did not exist in early versions of Visual Studio), the LIB file can be added as a dependency using the project's properties (figure 15-6).

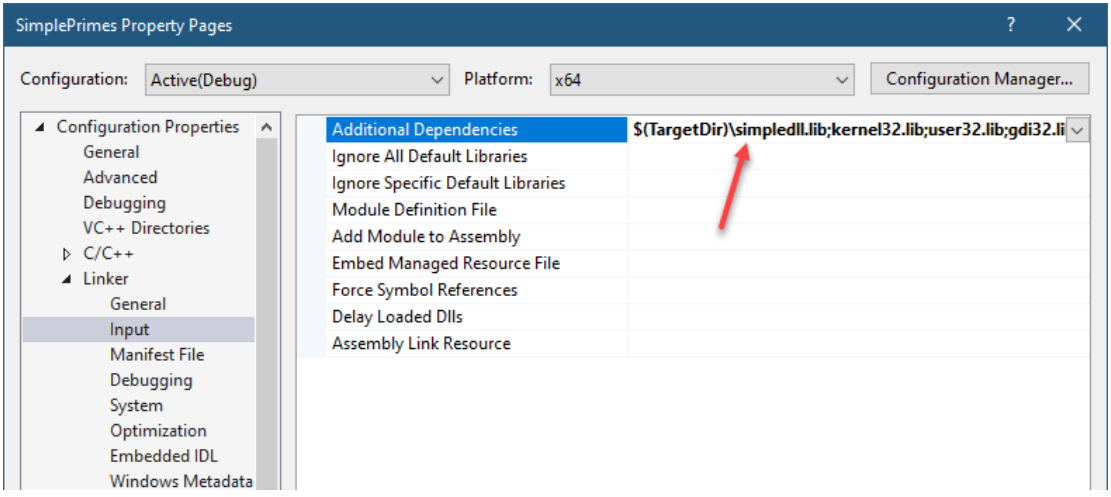


Figure 15-6: Import libraries in Project Properties

Figure 15-6 uses the **$(TargetDir)** variable used by Visual Studio to locate the LIB file properly, but more generally, the LIB file can be copied to anywhere and referenced. You can see all the "standard" subsystem DLLs the application links against using import libraries provided by the Visual Studio installation.

An alternative way to link against an import library (or a static library, for that matter), is add the dependency in code:

```
#ifdef _WIN64
    #pragma comment(lib, "../x64/Debug/SimpleDll.lib")
#else
    #pragma comment(lib, "../Debug/SimpleDll.lib")
#endif
```

The gymnastics around locating the LIB file is not elegant, but this can be improved by more configuration options to place the resulting LIB file in a more convenient directory, or by configuring other default search directories (all possible to do in a project's properties).

Using the `#pragma` option is somewhat easier to see, as the dependency on the *vcxproj* file is reduced.

With the import library in place, building the dependent project (the one using the DLL) take the following steps:

- The compiler sees a call to a function (`IsPrime` in *SimpleDll*) that it cannot find an implementation for in any source file.
- The compiler places instructions for the linker to locate such an implementation.
- The linker attempts to locate an implementation in static library files (that also typically have a LIB extension), but fails.
- The linker sees the imported lib where the function `IsPrime` is said to be implemented in the *SimpleDll.Dll*. The linker adds the appropriate data to the resulting PE that instructs the loader to locate the DLL at runtime.

At runtime, the loader (in *NtDll.Dll*), reads the information in the PE, and realizes it needs to locate the *SimpleDll.Dll* file. The search path the loader uses is the same described in chapter 3 for locating required DLLs by a newly created process - this is implicit linking in action. The search list (in order) is repeated here for convenience:

1. If the DLL name is one of the *KnownDLLs* (specified in the registry), the existing mapped file is used without searching.
2. The directory of the executable
3. The current directory of the process (determined by the parent process).
4. The System directory returned by `GetSystemDirectory` (e.g. *c:\windows\system32*)
5. The Windows directory returned by `GetWindowsDirectory` (e.g. *c:\Windows*)
6. The directories listed in the *PATH* environment variable

If the DLL is not found in any of these directories, the process shows the error message box shown in figure 15-7 and the process terminates.

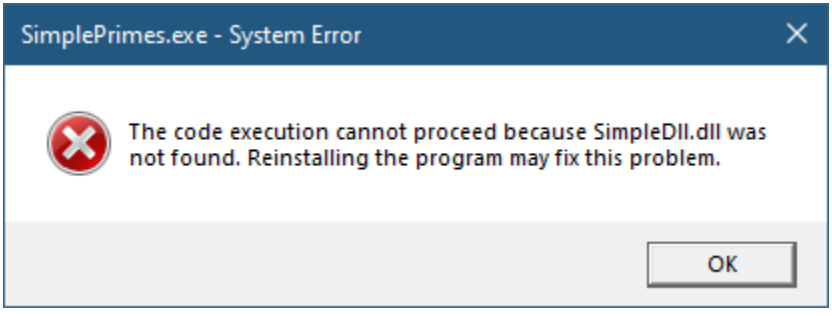


Figure 15-7: Failure to locate a DLL with implicit linking

The *KnownDLLs* registry key specifies DLLs that should be searched for in the system directory before trying other locations. This is used to prevent a hijacking of one of these DLLs. For example, a malicious

application may place its own copy of (say) *kernel32.dll* in the application's directory, causing the process to load the malicious version. This cannot happen, though, because *kernel32.dll* is in the list of the known DLLs. Figure 15-8 shows the *KnownDLLs* registry key at *HKLM\System\CurrentControlSet\Control\Session Manager\KnownDLLs*.

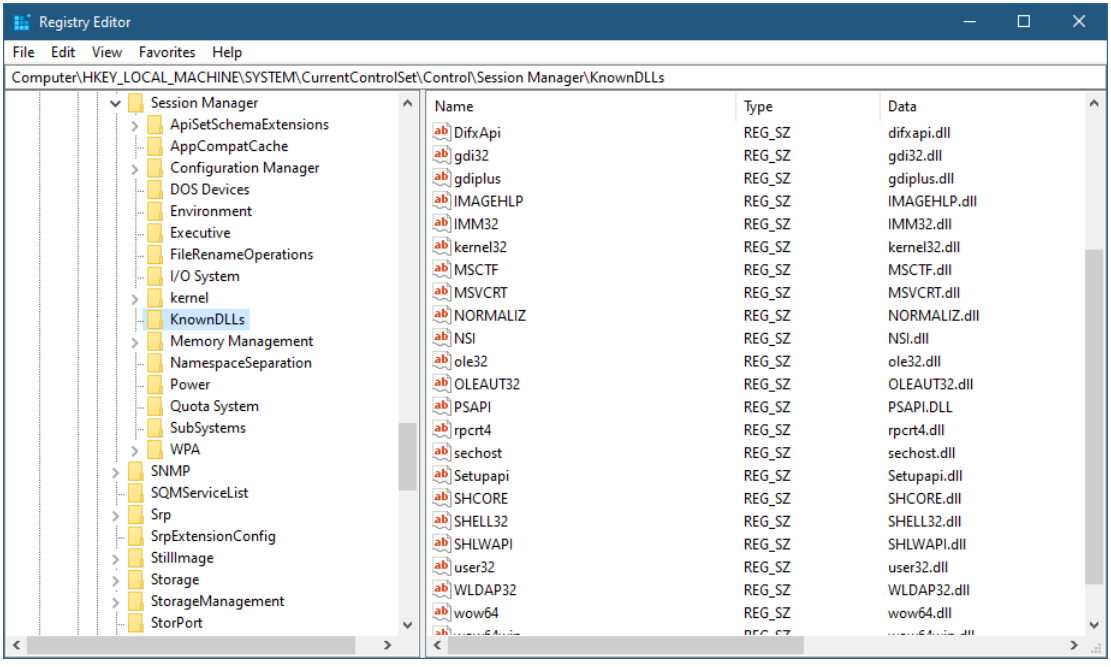


Figure 15-8: *KnownDLLs* in the Registry

The known DLLs are mapped when the system is initialized (with memory-mapped file objects), so that loading these DLLs into processes is faster, since the memory-mapped files are ready even before the DLLs are needed by the process. These file mapping (section) objects can be seen in *WinObj* (figure 15-9) from *Sysinternals* or my own *System Explorer*.

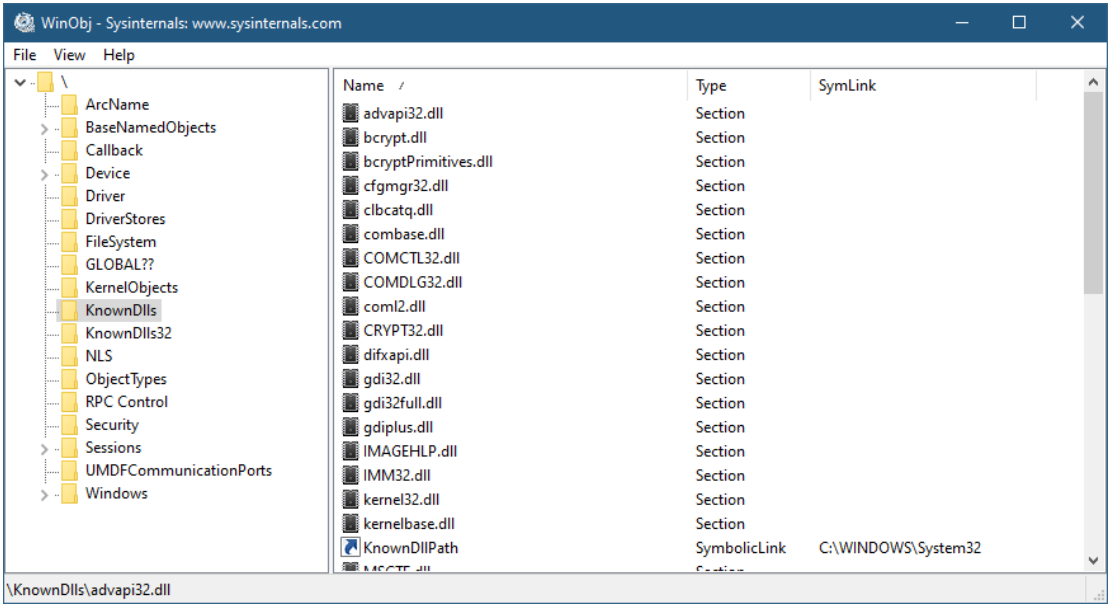


Figure 15-9: *KnownDLLs* sections in *WinObj*

If a DLL has dependencies on other DLLs (implicit linking again), these are searched in exactly the same way, recursively. All these DLLs must be located successfully, otherwise process terminates with the error message box in figure 15-7.

Once an implicitly loaded DLL is found, its `DllMain` function (if provided) is executed with a reason parameter of `DLL_PROCESS_ATTACH`, indicating to the DLL that it has now been loaded into a process (see the section "The `DllMain` Function" for more on `DllMain`). If `DllMain` returns `FALSE`, it indicates to the loader that the DLL was not initialized successfully, and the process terminates with a similar message box to figure 15-7.

All implicitly loaded DLLs load at process startup and unload when the process exits/terminates. Attempts to use the `FreeLibrary` function (discussed later) to unload such DLLs appear to succeed, but do nothing in practice.

Implicitly linking to DLLs is easy from the developer's perspective. Here is a summary of the steps needed by an application developer that wants to link implicitly with a DLL:

- Add the relevant `#include`(s) where the exported functions/variables are declared.
- Add the import library provided with the DLL to its set of imports (in one of the ways described earlier).
- Invoke exported functions or access exported variables.

The "exported functions" are not necessarily global functions. These can be C++ member functions in classes as well. The `__declspec(dllexport)` directive can be applied to a class like so:

```
class __declspec(dllexport) PrimeCalculator {
public:
    bool IsPrime(int n) const;
    std::vector<int> CalcRange(int from, int to);
};
```

And using this class is done normally, for example:

```
PrimeCalculator calc;
printf("123 prime? %s\n", calc.IsPrime(123) ? "Yes" : "No");
```

Explicit Linking

Explicit linking to a DLL provides more control over when the DLL is loaded and unloaded. Also, if the DLL fails to load the process does not crash, so the application can handle the error and carry on. One common use of explicit DLL linking is to load language-related resources. For example, an application might try to load a DLL with resources in the current system locale, and if not found, can load a default resource DLL that is always provided as part of the application's installation.

With explicit linking, no import library is used so that the loader does not attempt to load the DLL (as it may or may not exist). This also means that you cannot use `#include` to get the exported symbols declarations, because the linker will fail with "unresolved external" errors. How can we use such a DLL?

The first step is to load it at runtime, typically close to where it's needed. This is the job of `LoadLibrary`:

```
HMODULE LoadLibrary(_In_ LPCTSTR lpLibFileName);
```

`LoadLibrary` accepts a file name only or a full path. If a file name only is specified, the search for the DLL is done in the same order as for implicitly loaded DLLs as described in the previous section. If a full path is specified, only that file is attempted loading.

Before the actual search begins, the loader checks if there is a module with the same name that is loaded into the process address space already. If so, no search is performed, and the existing DLL's handle is returned. For example, if *SimpleDll.Dll* is loaded already (from whatever path), and `LoadLibrary` is called to load a file named *SimpleDll.Dll* (in whatever path or without apath), no additional DLL is loaded.

If the DLL is located successfully, it's mapped into the process address space, and `LoadLibrary`'s return value is the virtual address into which it's mapped in the process. The type itself is `HMODULE` and sometimes `HINSTANCE` - these are interchangeable; these are not "handles" in any sense. These type names are relics from 16-bit Windows. In any case, this return value represents the DLL uniquely in the process address space, and is the parameter to use with other functions that access information in the DLL, as we shall soon see.

As with all DLL loads, `DllMain` is called on the loaded DLL. If `TRUE` is returned, the DLL is considered loaded successfully, returning control to the caller. Otherwise, the DLL unloads and the function fails.

If the function fails (because the DLL failed to locate or its `DllMain` returned `FALSE`), `NULL` is returned to the caller.

Now that the DLL is loaded, how can we access exported functions from the DLL? The function in question is `GetProcAddress`:

```
FARPROC GetProcAddress(
    _In_ HMODULE hModule,
    _In_ LPCSTR lpProcName);
```

The function returns the address of an exported symbol from the DLL. Its first parameter is the DLL handle returned from `LoadLibrary`. The second parameter is the name of the symbol. Notice the name must be ASCII - there is no Unicode variant. The return value is a generic `FARPROC`, yet another type from the old 16-bit days, where "far" and "near" meant different things. The actual definition of `FARPROC` is unimportant; the caller will cast the return value to the appropriate type based on some foreknowledge, such as a header file (that cannot be included) or old-fashioned documentation. If the symbol does not exist (or is not exported, which is the same thing), `GetProcAddress` returns `NULL`.

Let's go back to the `IsPrime` function exported form *SimpleDll.Dll* like so:

```
__declspec(dllexport) bool IsPrime(int n);
```

It looks innocent enough. Here is a first attempt at loading *SimpleDll.Dll* dynamically, and then locating `IsPrime`:

```
auto hPrimesLib = ::LoadLibrary(L"SimpleDll.dll");
if (hPrimesLib) {
    // DLL found
    using PIsPrime = bool (*)(int);
    auto IsPrime = (PIsPrime)::GetProcAddress(hPrimesLib, "IsPrime");
    if (IsPrime) {
        bool test = IsPrime(17);
        printf("%d\n", (int)test);
    }
}
```

The `using` statement in the preceding code is the preferred way in C++ 11 and higher to create type definition, effectively replacing `typedef`. If you're using C or an old compiler, replace this `using` line with `typedef bool (*PIsPrime)(int);` Function type definitions are never pretty, but `using` makes them bearable.

The code looks relatively straightforward - the DLL is loaded (it's located in the executable's directory when building a DLL and an EXE as part of the same solution in Visual Studio), so it's located just fine. Unfortunately, the call to `GetProcAddress` fails, with `GetLastError` returning 127 ("The specified procedure cannot be found"). Clearly, `GetProcAddress` cannot locate the exported function, even though it was exported. Why?

The reason has to do with the name of the function. If we go back to the information dumped by *Dumpbin* about *SimpleDll.Dll*, this is what we find (see earlier in this chapter):

```
1    0 000111F9 ?IsPrime@@YA_NH@Z = @ILT+500(?IsPrime@@YA_NH@Z)
```

The linker "mangled" the name of the function to be `?IsPrime@@YA_NH@Z`. The reason has to do with the fact that the name "IsPrime" is not unique enough in C++. An `IsPrime` function can be in class `A` and in class `B` as well as globally. And it could be part of some namespace `C`. If this is not enough, there can be multiple functions named `IsPrime` in the same scope, due to C++ function overloading. So the linker gives the function a weird-looking name that contains these unique attributes. We can try substituting this mangled name in the preceding code example like so:

```
auto IsPrime = (PIsPrime)::GetProcAddress(
    hPrimesLib, "?IsPrime@@YA_NH@Z");
```

And this works! However, it's not fun, and we have to look up the mangled name with a tool to get it right. The common practice is to turn all exported functions into C-style functions. Because C does not support function overloading or classes, the linker does not have to do complex mangling. Here is one way to export a function as C:

```
extern "C" __declspec(dllexport) bool IsPrime(int n);
```

If you're compiling C files, this would be the default.

With this change, getting the pointer to the `IsPrime` function is simplified:

```
auto IsPrime = (PIsPrime)::GetProcAddress(hPrimesLib, "IsPrime");
```

This scheme of turning function into C-style, cannot be done for member functions in classes. This is why it's not practical to use `GetProcAddress` to access C++ functions. This is why most DLLs that are intended to be used with `LoadLibrary` / `GetProcAddress` expose C-style functions only.

Once a DLL is no longer needed, call `FreeLibrary` to unload it from the process:

```
BOOL FreeLibrary(_In_ HMODULE hLibModule);
```

The system maintains a per-process counter for each loaded DLL. If multiple calls to `LoadLibrary` are made for the same DLL, the same number of `FreeLibrary` calls are needed to truly unload the DLL from the process address space.

If a handle to a loaded DLL is needed, `GetModuleHandle` can be used to retrieve it:

```
HMODULE GetModuleHandle(_In_opt_ LPCTSTR lpModuleName);
```

The module name does not need a full path, just a DLL name. If no extension is provided, ".dll" is appended by default. The function does **not** increment the load count for the DLL. If the module name is `NULL`, the handle to the executable is returned. The executable is mapped to the process address space just like a DLL - the returned "handle" is in fact the virtual address to which the executable image was loaded.

Calling Conventions

The term *Calling Convention* indicates (among other things) how function parameters are passed to functions, and who is responsible for cleaning up the parameters, if passed on the stack. For x64, there is only one calling convention. For x86, there are several. The most common are the *standard calling convention* (`stdcall`) and the *C calling convention* (`cdecl`). Both `stdcall` and `cdecl` use the stack to pass arguments, pushed from right to left. The main difference between them is that with `stdcall` the callee (the function body itself) is responsible for cleaning up the stack, whereas with `cdecl` the caller is responsible for that.

`stdcall` has the advantage of being smaller, since the stack cleanup code appears only one (as part of the fucntion's body). With `cdecl`, every call to the function must be followed by an instruction to clean up the arguments from the stack. The advantage of `cdecl` functions is the fact they can accept a variable number of parameters (specified by the ellipsis `...` in C/C++), because only the caller knows how many arguments were passed in.

> The discussion on calling conventions in this section is far from exhaustive. Check online resources to get all the details.

The default calling convention used in user mode projects in Visual C++ is `cdecl`. Specifying the calling convention is done by placing the proper keyword between the return type and the function name. The Microsoft compiler recognizes the `__cdecl` and `__stdcall` keywords for this purpose. The keyword used must be specified in the implementation as well. Here is an example of turning `IsPrime` to use `stdcall`:

```
extern "C" __declspec(dllexport) bool __stdcall IsPrime(int n);
```

This also means that when defining the function pointer for use with `GetProcAddress`, the correct calling convention must be specified as well, otherwise we'll get runtime errors or stack corruption:

```
using PIsPrime = bool (__stdcall *)(int);
// or
typedef bool(__stdcall* PIsPrime)(int);
```

`__stdcall` is the calling convention used for most Windows APIs. This is usually conveyed using one of the following macros, which mean the exact same thing (`WINAPI`, `APIENTRY`, `PASCAL`, `CALLBACK`). This is why one of these macros is used in the Windows headers. Here is the exact declaration of the `Sleep` function:

```
VOID WINAPI Sleep(_In_ DWORD dwMilliseconds);
```

I've omitted these macros when showing function declarations to simplify them and focus on the important stuff.

There is an additional wrinkle with `stdcall` functions. The linker mangles them differently than `cdecl`, by prefixing their names with an underscore and appending the `@` sign and the number of bytes passed in as arguments. So for `IsPrime`, the actual exported name is `_IsPrime@4`. This also means the name passed to `GetProcAddress` should be this name for x86, but just `IsPrime` for x64 (x64 does not mangle names since it has a single calling convention).

The solution for `stdcall` functions is to use a *Module Definition* (DEF) file. This file can be added to a DLL project to specify various options. Its main usage is to list exported symbols, which means using `__declspec(dllexport)` is no longer needed. The exported functions can be looked up by their simple name regardless of the calling convention.

You can add a DEF file by using Visual Studio's "Add New Item..." menu just like any other file. You can search for "def" or just specify the file's extension explicitly. The DEF file name must be the same as the projec's name so it's processed without any extra configuration. For *SimpleDll* the file name is *SimpleDll.def*. Here is the contents for the DEF file to export `IsPrime` in a consistent manner:

```
LIBRARY
EXPORTS
    IsPrime
```

If more exports are needed, add each one in a separate line. With this file in place, the `IsPrime` function can be looked up by its simple name with `GetProcAddress`.

DLL Search and Redirection

When calling `LoadLibrary` with a file name only, a certain search path is used. It's possible to add a custom path to search for DLLs with `SetDllDirectory`:

```
BOOL SetDllDirectory(_In_opt_ LPCTSTR lpPathName);
```

The specified path is looked up after the executable's directory. If `lpPathName` is `NULL`, any directory set earlier by `SetDllDirectory` is removed, restoring the default search order. If `lpPathName` is an empty string, then the current directory of the process is removed from the search list. Each call to `SetDllDirectory` replaces any previous call.

If multiple search directories are desired, call `AddDllDirectory`:

```
DLL_DIRECTORY_COOKIE AddDllDirectory(_In_ PCTSTR NewDirectory);
```

The function adds the specified directory to the search path, and returns an opaque pointer that represents this "registration". However, directories added with `AddDllDirectory` are not used automatically. An additional call must be made to `SetDefaultDllDirectories` to enable these extra directories:

```
BOOL SetDefaultDllDirectories(_In_ DWORD DirectoryFlags);
```

The flags can be a combination of the values listed in table 15-1.

Table 15-1: Flags for `SetDefaultDllDirectories`

Value (`LOAD_LIBRARY_SEARCH_` prefix)	Description
`APPLICATION_DIR`	Executable directory is included in the search
`USER_DIRS`	Adds directories added with `AddDllDirectory` to the search
`SYSTEM32`	Adds the *System32* directory to the search
`DEFAULT_DIRS`	Combines all the previous values
`DLL_LOAD_DIR`	The loaded DLL's directory is added temporarily to the search for dependent DLLs

To allow `AddDllDirectory` directories to have an effect on future `LoadLibrary` calls, use the following call:

```
::SetDefaultDllDirectories(LOAD_LIBRARY_SEARCH_USER_DIRS);
```

An added directory should be removed at some point with `RemoveDllDirectory`:

```
BOOL RemoveDllDirectory(_In_ DLL_DIRECTORY_COOKIE Cookie);
```

> There is no explicit function to return to the default search paths. The best way to handle this is to call `RemoveDllDirectory` for each `AddDllDirectory` and call `SetDllDirectory` with `NULL`. Alternatively, the `LoadLibraryEx` function can be used for a "one time" search path alteration (see later in this chapter).

The `DllMain` Function

A DLL can have an entry point, traditionally called `DllMain` that must have the following prototype:

```
BOOL WINAPI DllMain(HINSTANCE hInsdDll, DWROD reason, PVOID reserved);
```

The `hInstance` parameter is the virtual address the DLL is loaded into the process. It's the same value returned from `LoadLibrary` if the DLL is loaded explicitly. The `reason` parameter indicates why `DllMain` was called. It can have the values listed in table 15-2.

Table 15-2: Reason values for `DllMain`

Reason value	Description
`DLL_PROCESS_ATTACH`	Called when the DLL is attached to a process
`DLL_PROCESS_DETACH`	Called before the DLL is unloaded from a process
`DLL_THREAD_ATTACH`	Called when a new thread is created in the process
`DLL_THREAD_DETACH`	Called before a thread exits in the process

When a DLL is loaded into a process, `DllMain` is called with reason `DLL_PROCESS_ATTACH`. If the same DLL is loaded multiple times into the same process (calling `LoadLibrary` multiple times), the internal reference counter for the DLL is incremented, but `DllMain` is not called again. With `DLL_PROCESS_ATTACH`, `DllMain` must return `TRUE` to indicate the DLL initialized properly, or `FALSE` otherwise. If `FALSE` is returned, the DLL is unloaded.

The opposite of `DLL_PROCESS_ATTACH` is `DLL_PROCESS_DETACH`, called before the DLL is unloaded. It could be because the entire process is shutting down, or because `FreeLibrary` was called to unload this DLL. Remember that forceful process termination with `TerminateProcess` does not invoke `DllMain` (see chapter 3 for more details).

The remaining two values cause `DllMain` to be called when a new thread is created (`DLL_THREAD_ATTACH`) and before a thread exits (`DLL_THREAD_DETACH`). Many DLLs don't care about threads created or destroyed in their hosting process. A useful optimization in such a case is to call `DisableThreadlibraryCalls`:

```
BOOL DisableThreadLibraryCalls(_In_ HMODULE hLibModule);
```

This call with the DLL's module handle tells the system not to call `DllMain` for thread-related events. This call is typically invoked when `DLL_PROCESS_ATTACH` reason is sent.

> The `DLL_THREAD_ATTACH` reason is not invoked for the first thread in the process - the DLL should use `DLL_PROCESS_ATTACH` for that.

If the DLL uses *Thread Local Storage* (TLS, discussed in chapter 10), then it might want to allocate some structure for each thread in the process. The reasons `DLL_THREAD_ATTACH` and `DLL_THREAD_DETACH` are useful for such allocations and deallocations.

There is a fifth value supported for reason, called `DLL_PROCESS_VERIFIER` (equal to 4), that can be used to write *Application Verifier* DLLs, although it's not officially documented. I'll say more about Application Verifier in chapter 20.

The last parameter to `DllMain` is called "reserved", but it indicates whether the DLL is implicitly loaded (`lpReserved` is non-`NULL`) or explicitly loaded (`lpReserved` is `NULL`).

Creating a DLL project with Visual Studio provides a bare-bones `DllMain` that just returns `TRUE` for all notifications. Here is a simple `DllMain` that calls `DisableThreadlibraryCalls` when the DLL is loaded into a process:

```
BOOL WINAPI DllMain(HMODULE hModule, DWORD reason, LPVOID) {
    switch (reason) {
        case DLL_PROCESS_ATTACH:
            ::DisableThreadLibraryCalls(hModule);
            break;

        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
            break;
    }
    return TRUE;
}
```

The `DllMain` function is called with the *Loader Lock* held. You can think of the Loader Lock as a critical section. What this means is that calling some functions is not allowed or dangerous from `DllMain` as they can lead to deadlocks. For example, if `DllMain` code (say in `DLL_THREAD_ATTACH`) waits to acquire a mutex managed by the application while another thread holds the mutex. The other thread, before releasing the mutex, calls some function that causes an attempt to acquire the Loader Lock, such as calling `LoadLibrary` or `GetModuleHandle`. This causes a deadlock.

The recommendation is simple: do as little as possible in `DllMain`, and defer other initialization to an explicit function that could be called right after `DllMain` returns. Functions that create/destroy processes and DLLs (`CreateProcess`, `CreateThread`, etc.) should be avoided. Using heap or virtual APIs, I/O functions, TLS and most other functions from *kernel32.dll* are safe to use.

DLL Injection

In some cases, it's desirable to inject a DLL into another process. By "injecting a DLL" I mean forcing in some way another process to load a specific DLL. This allows that DLL to execute code in the context of the target process. There are many uses for such an ability, but all of them essentially boil down to some form of customization or interception of operations within the target process. Here are some concrete examples:

- Anti-malware solutions and other applications may want to hook API functions in the target process. Hooking is described in the next major section.
- The ability to customize windows by *subclassing* windows or controls, allowing behavioral changes to the UI.
- Being part of a target process gives unlimited access to anything in that process. Some can be used for good, like DLLs that monitor an application's behavior to locate bugs, and some for bad.

In this section, we'll look at some common techniques for DLL injections. These are by no means exhaustive, as the cyber-security community always manages to come up with an ingenious way to inject code into a target process. This section focuses on the more "traditional" or "standard" techniques to make the fundamentals understandable.

Injection with Remote Thread

Injecting a DLL by creating a thread in the target process that loads the required DLL is probably the most well-known and straightforward technique (relatively speaking). The idea is to create a thread in a target process that calls the `LoadLibrary` function with the DLL path to be injected. The problem is, how do you get the code to execute into the target process?

The *Injector* project demonstrates this technique. First, we need to check command-line arguments:

```
int main(int argc, const char* argv[]) {
    if (argc < 3) {
        printf("Usage: injector <pid> <dllpath>\n");
        return 0;
    }
```

The injector requires the target's process ID and the DLL to inject. Next, we open a handle to the target process:

```
HANDLE hProcess = ::OpenProcess(
    PROCESS_VM_WRITE | PROCESS_VM_OPERATION | PROCESS_CREATE_THREAD,
    FALSE, atoi(argv[1]));
if (!hProcess)
    return Error("Failed to open process");
```

As we'll see very soon, we need quite a few access mask bits to gain sufficient power for this injection technique. This means that some processes would not be accessible.

The trick with this injection method is the fact that from a binary standpoint, the `LoadLibrary` function and a thread's function are essentially the same:

```
HMODULE WINAPI LoadLibrary(PCTSTR);
DWORD WINAPI ThreadFunction(PVOID);
```

Both prototypes accept a pointer, and here lies the trick: we can create a thread that runs the function `LoadLibrary`! This is nice because the code to `LoadLibrary` is already in the target's process (as it's part of *kernel32.dll* that must be loaded into every process which is part of the Windows subsystem).

The next task is to prepare the DLL path to load. The path string itself must be placed in the target process since that's where `LoadLibrary` would execute. We can use the `VirtualAllocEx` function for this purpose:

```
void* buffer = ::VirtualAllocEx(hProcess, nullptr, 1 << 12,
    MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
if (!buffer)
    return Error("Failed to allocate buffer in target process");
```

Using `VirtualAllocEx` requires `PROCESS_VM_OPERATION` access mask, which we requested at attractive`OpenProcess` time. We allocate a 4 KB buffer, which is overkill, but even if we specify less than that, it would be rounded up to 4 KB anyway. Note the returned pointer has no meaning in the caller's process - it's the allocated address in the target process.

Next we need to copy the DLL path to the allocated buffer with `WriteProcessMemory`:

```
if (!::WriteProcessMemory(hProcess, buffer,
    argv[2], ::strlen(argv[2]) + 1, nullptr))
    return Error("Failed to write to target process");
```

Using `WriteProcessMemory` requires the process handle to have the `PROCESS_VM_WRITE` access mask, which it does. The code writes the DLL's path retrieved from the command line using ASCII, which may seem unusual. We'll see why in a moment. Everything is ready = it's time to create the remote thread:

```
DWORD tid;
HANDLE hThread = ::CreateRemoteThread(hProcess, nullptr, 0,
    (LPTHREAD_START_ROUTINE)::GetProcAddress(
        ::GetModuleHandle(L"kernel32"), "LoadLibraryA"),
    buffer, 0, &tid);
if (!hThread)
    return Error("Failed to create remote thread");
```

`CreateRemoteThread` accepts the target process handle (must have `PROCESS_CREATE_THREAD` access mask, which it does), a `NULL` security descriptor, default stack size, and the thread's start routine. This is where we take advantage of the binary equivalence of `LoadLibrary` and a thread's function.

First, `LoadLibrary` is not a function at all - it's a macro. We must select `LoadLibraryA` or `LoadLibraryW` - I selected `LoadLibraryA`, just because it's a bit more convenient to use. This is why the copied string above was ASCII. We could have used `LoadLibraryW` and copied a Unicode string to the target process and essentially get the same result.

The `GetProcAddress` call is used to dynamically locate the address of `LoadLibraryA`, taking advantage of the fact that it's the same address in the current process. This is the key to this technique - no need to copy code into the target process. The parameter to the thread is `buffer` - the address in the target process where we copied the DLL path.

And that's it. One important thing to note is that the injected DLL must be the same "bitness" as the target process, as Windows does not allow a 32-bit process to load a 64-bit DLL or vice versa.

All that's left is to do some cleanup:

```
printf("Thread %u created successfully!\n", tid);
if (WAIT_OBJECT_0 == ::WaitForSingleObject(hThread, 5000))
    printf("Thread exited.\n");
else
    printf("Thread still hanging around...\n");

// be nice
::VirtualFreeEx(hProcess, buffer, 0, MEM_RELEASE);

::CloseHandle(hThread);
::CloseHandle(hProcess);
```

Waiting for the thread to terminate is not mandatory, but we need to give it some time before calling `VirtualFreeEx` to remove the allocation done with `VirtualAllocEx`. This is polite, but not strictly necessary. We could just as well leave that 4 KB committed in the target process.

The solution for this chapter has a DLL project named *Injected* you can use to test this technique. Here is a command-line example for testing:

```
C:\>Injector.exe 44532 C:\Temp\Injected.dll
```

You might get a notification from your anti-virus software if you have any, as the above combination of APIs is typically monitored for and considered malicious. *Windows Defender* on my machine labeled *Injector.exe* as malware, threatening to delete it.

You must specify a full path to the DLL, as the loading rules are from the target process' perspective, rather than the caller's. The *Injected* DLL's `DllMain` shows a simple message box:

```
BOOL WINAPI DllMain(HMODULE hModule, DWORD reason, PVOID) {
    switch (reason) {
        case DLL_PROCESS_ATTACH:
            wchar_t text[128];
            ::StringCchPrintf(text, _countof(text),
                L"Injected into process %u",
                ::GetCurrentProcessId());
            ::MessageBox(nullptr, text, L"Injected.Dll", MB_OK);
            break;
    }
    return TRUE;
}
```

Windows Hooks

The term *Windows Hooks* used in this section refers to a set of user interface-related hooks available with the `SetWindowsHookEx` API:

```
HHOOK SetWindowsHookEx(
    _In_ int idHook,
    _In_ HOOKPROC lpfn,
    _In_opt_ HINSTANCE hmod,
    _In_ DWORD dwThreadId);
```

The first parameter is the hook type. There are several types of hooks, each with its own semantics. You can find the full list and details in the official documentation. These hooks can be installed for a specific thread (provided by the `dwThreadId` parameter), or globally, for all processes in the caller's desktop (`dwThreadId` set to zero). Some hook types can only be installed globally (`WH_JOURNALRECORD`, `WH_JOURNALPLAYBACK`, `WH_MOUSE_LL`, `WH_KEYBOARD_LL`, `WH_SYSMSGFILTER`), while the others may be installed globally or for a particular thread.

The hook function provided by `lpfn` has the following prototype:

```
LRESULT CALLBACK HookProc(int code, WPARAM wParam, LPARAM lParam);
```

The meaning of the parameters is described for each individual type of hook. The function must be valid in the context of the hooked process. If the hook is used globally or on a thread of a different process, the callback function must be part of a DLL, that is injected into the target process or processes. In that case, the `hmod` parameter is the DLL's handle provided by the caller. If the hooked thread is within the calling process, the module handle can be `NULL`, and the hook callback can be part of the caller's process.

> There are other details specifically for global hooks. Since a 32-bit DLL cannot be loaded by a 64-bit process and vice versa, how do you hook both 32-bit and 64-bit processes? One option is to have two hooking applications, 32-bit and 64-bit, each providing its own DLL with the correct bitness. Another option is to use just one installing application, but that causes the other bitness processes to make a remote call back to the hooking application, which must pump messages. This works, but is slower, and the callback is not running in the context of a target process. Check out the documentation for more details.

The return value of `SetWindowsHookEx` is a handle to the hook, or `NULL` if the function fails. The nice thing about `SetWindowsHookEx` is that the DLL (if provided) is automatically injected by *Win32k.sys* behind the scenes. This is considerably less visible than using something like `CreateRemoteThread`.

`SetWindowsHookEx` is not perfect. Here are a few of its shortcomings:

- It can only be used on processes that load *user32.dll*. Processes that don't have a GUI typically don't load *user32.dll*.
- The global hooks are "global" to all threads that use the caller's desktop. Thus, it cannot hook processes in other sessions, even if they have a GUI.

DLL Injecting and Hooking with `SetWindowsHookEx`

The following example uses `SetWindowsHookEx` with the `WH_GETMESSAGE` hook type to inject a DLL into the first *Notepad* process found, and monitor all keys typed in. These keys are sent to the monitoring application, which effectively sees every key stroke made by the user in *Notepad*.

There are two projects involved in this system. The injecting executable (*HookInject*) and the DLL to be injected indirectly with `SetWindowsHookEx` (*HookDll*). Lets' start with the injecting application.

To test these, run *Notepad* first, and then run *HookInject*. Now start typing in *Notepad*. You'll see the same text echoed in the console window of *HookInject* (figure 15-10).

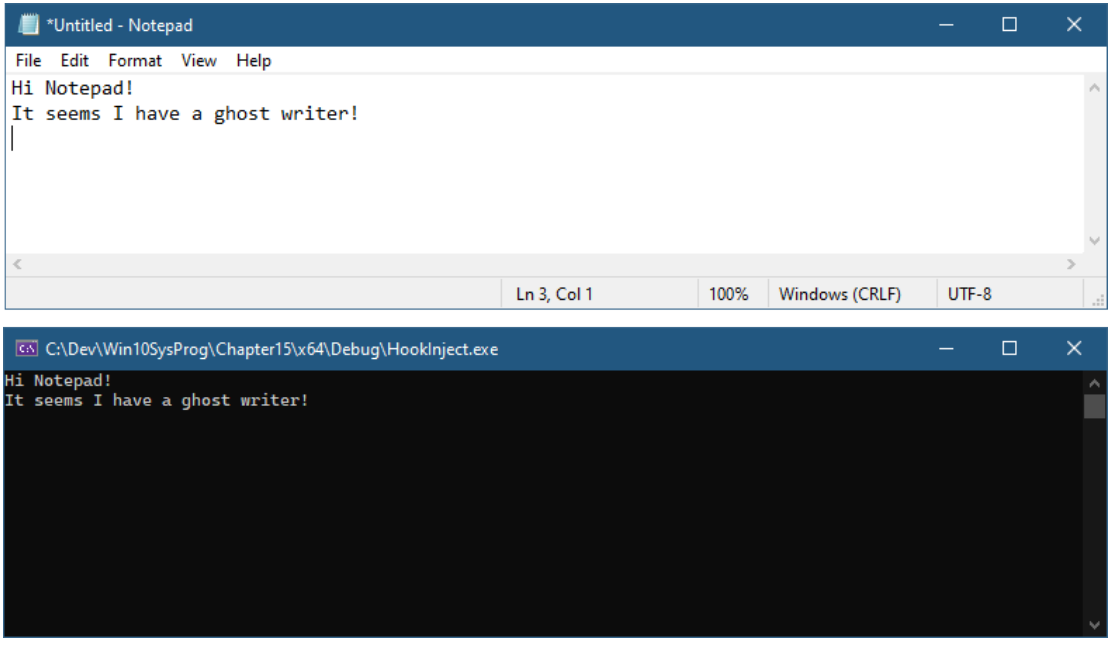


Figure 15-10: *Notepad* hooked

You can make sure the DLL is in fact injected by looking it up *Notepad* in *Process Explorer* and examine the loaded modules (figure 15-11).

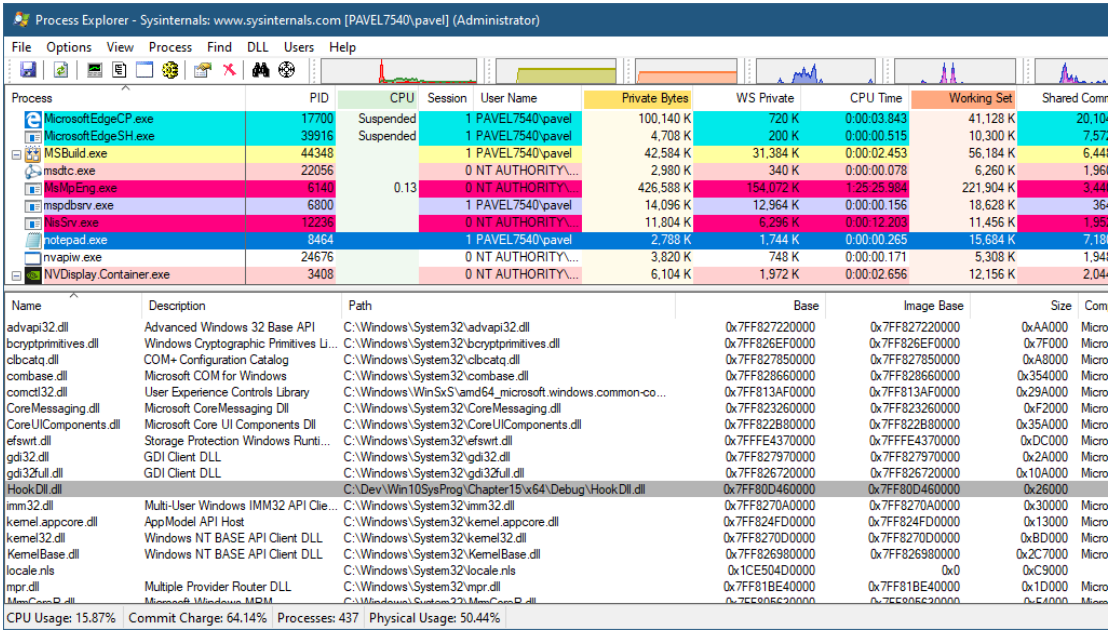


Figure 15-11: Injected DLL in *Notepad*'s process

The first order of business is to locate the first thread of the first *Notepad* instance, since we need to get information on messages processes by *Notepad*'s UI, handled by *Notepad*'s first thread. To this end, we can write a thread enumeration function using the *Toolhelp* API and locate that thread:

```
DWORD FindMainNotepadThread() {
    auto hSnapshot = ::CreateToolhelp32Snapshot(
        TH32CS_SNAPTHREAD, 0);
    if (hSnapshot == INVALID_HANDLE_VALUE)
        return 0;

    DWORD tid = 0;
    THREADENTRY32 th32;
    th32.dwSize = sizeof(th32);

    ::Thread32First(hSnapshot, &th32);
    do {
        auto hProcess = ::OpenProcess(
            PROCESS_QUERY_LIMITED_INFORMATION,
            FALSE, th32.th32OwnerProcessID);
        if (hProcess) {
            WCHAR name[MAX_PATH];
            if (::GetProcessImageFileName(hProcess, name,
                MAX_PATH)) {
```

```
                auto bs = ::wcsrchr(name, L'\\');
                if (bs && ::_wcsicmp(bs, L"\\notepad.exe") == 0) {
                    tid = th32.th32ThreadID;
                }
            }
            ::CloseHandle(hProcess);
        }
    } while (tid == 0 && ::Thread32Next(hSnapshot, &th32));
    ::CloseHandle(hSnapshot);

    return tid;
}
```

`CreateToolhelp32Snapshot` is used with `TH32CS_SNAPTHREAD` to enumerate all threads in the system (the API does not support enumerating threads in a specific process). For each thread, a handle to the parent process is opened. If successful, the image path of the executable is looked up with `GetProcessImageFileName`. If this ends in *\Notepad.exe*, then the thread ID is returned, since it would be the first thread.

Now the `main` function can get to work. It starts by calling `FindMainNotepadThread`:

```
int main() {
    DWORD tid = FindMainNotepadThread();
    if (tid == 0)
        return Error("Failed to locate Notepad");
```

Next, the DLL to be injected is loaded, and two exported functions are extracted:

```
auto hDll = ::LoadLibrary(L"HookDll");
if (!hDll)
    return Error("Failed to locate Dll\n");

using PSetNotify = void (WINAPI*)(DWORD, HHOOK);
auto setNotify = (PSetNotify)::GetProcAddress(hDll,
    "SetNotificationThread");
if (!setNotify)
    return Error("Failed to locate SetNotificationThread function");

auto hookFunc = (HOOKPROC)::GetProcAddress(hDll, "HookFunction");
if (!hookFunc)
    return Error("Failed to locate HookFunction function");
```

`SetNotificationThread` is a function exported from the DLL that will be used later to communicate information from the *Notepad* process to the injector/monitoring process. `HookFunction` is the hook function itself that must be passed to `SetWindowsHookEx`.

It's time to install the hook:

```
auto hHook = ::SetWindowsHookEx(WH_GETMESSAGE, hookFunc, hDll, tid);
if (!hHook)
    return Error("Failed to install hook");
```

The hook type is `WH_GETMESSAGE`, which is useful for intercepting messages destined to windows created by the hooked thread. An added bonus of this hook is the ability of the hook function to change the message if desired before it reached its destination.

At this point the hook DLL is injected automatically into *Notepad*'s process when the next message is sent to *Notepad*'s window, and the hook function will be called for each message. What can the hook function do with the message information? The simplest option would be to simply write it to some file. However, to make it a bit more interesting, the hook function notifies the injecting process of all keystrokes by using thread messages. The hook function (inside *Notepad*'s process) needs to know to which thread to send the messages. This is the role of `SetNotificationThread`, which is now invoked:

```
setNotify(::GetCurrentThreadId(), hHook);
::PostThreadMessage(tid, WM_NULL, 0, 0);
```

`SetNotificationThread` is passed two pieces of information: the caller's thread ID to receive messages, and the hook's handle itself that will be needed by the hook function as we'll soon see. If you're paying attention, the call does not make perfect sense, since it calls `SetNotificationThread` in the local process - the DLL was loaded into this process, but the information conveyed by these two parameters should be available in the context of the *Notepad* process. What gives? We'll solve this conundrum soon.

The call to `PostThreadMessage` is a trick to wake up *Notepad* with a dummy message (`WM_NULL`) that will force it to load out hook DLL, if it wasn't loaded already.

> `PostThreadMessage` is function that allows sending a window message to a thread. Normal messages are targeted at a window (using the window handle). With `PostThreadMessage`, the window handle is effectively `NULL`. The rest of the parameters are the same as for other window message sending functions, such as `SendMessage` and `PostMessage`. For thread messages, there is no "SendThreadMessage", meaning `PostThreadMessage` is asynchronous in nature - it puts the message in the target thread's queue and returns immediately. With messages targeted at a window handle, there is more flexibility - `SendMessage` is synchronous while `PostMessage` is asynchronous.

All that's left to do now is wait for incoming messages from the hooked *Notepad* and handle them in some way:

```
    MSG msg;
    while (::GetMessage(&msg, nullptr, 0, 0)) {
        if (msg.message == WM_APP) {
            printf("%c", (int)msg.wParam);
            if (msg.wParam == 13)
                printf("\n");
        }
    }
    ::UnhookWindowsHookEx(hHook);
    ::FreeLibrary(hDll);

    return 0;
}
```

`GetMessage` examines the message queue of the current thread and only returns if a message is there. Since the current thread has no windows, any message is destined for the thread, coming from the *Notepad* process. `GetMessage` returns if a message is available other than `WM_QUIT`. We would want `WM_QUIT` to be posted by the hook function if the *Notepad* process is terminating.

The expected message is `WM_APP` (0x8000), which is guaranteed to be unused by standard window message constants, and it's the one that is posted from the hook function (as we shall soon see). The `wParam` member of the `MSG` structure holds the key (again, provided by the hook function), which the application just echos to the console.

Finally, when `WM_QUIT` message is received, the process cleans up by unhooking the hook and unloading the DLL.

Now let's turn our attention to the hook DLL. The conundrum we had earlier is solved by having some global shared variables that are part of the DLL:

```
#pragma data_seg(".shared")
DWORD g_ThreadId = 0;
HHOOK g_hHook = nullptr;
#pragma data_seg()
#pragma comment(linker, "/section:.shared,RWS")
```

This technique of sharing variables in a DLL (or EXE) was described in chapter 12. The injecting application called `SetNotificationThread` in the context of its own process, but the function writes the information to the shared variables, so these are available to any process using the same DLL:

```
extern "C"
void WINAPI SetNotificationThread(DWORD threadId, HHOOK hHook) {
    g_ThreadId = threadId;
    g_hHook = hHook;
}
```

This arrangement is depicted in figure 15-12.

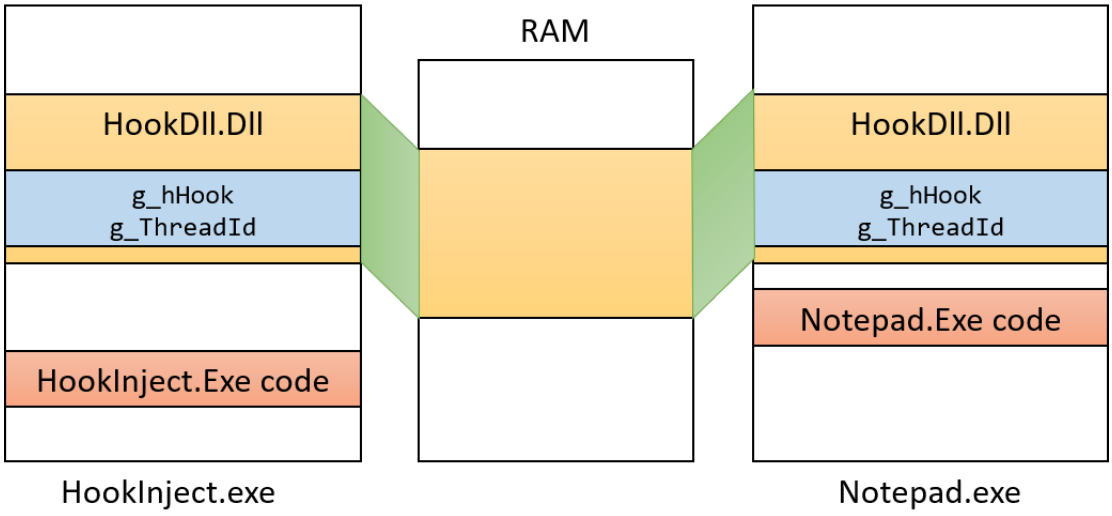


Figure 15-12: Injeting and injected processes

The `DllMain` function is implemented like so:

```
BOOL WINAPI DllMain(HMODULE hModule, DWORD reason, PVOID resevred) {
    switch (reason) {
        case DLL_PROCESS_ATTACH:
            ::DisableThreadLibraryCalls(hModule);
            break;
        case DLL_PROCESS_DETACH:
            ::PostThreadMessage(g_ThreadId, WM_QUIT, 0, 0);
            break;

    }
    return TRUE;
}
```

First, it calls `DisableThreadLibraryCalls` when attached to the process, indicating the DLL does not care about thread creation/destruction. When unloaded, probably because *Notepad* is exiting, a `WM_QUIT` message is sent to the injector's thread, that causes its `GetMessage` call to return `FALSE`.

The interesting work is done by the hook function:

```
extern "C" LRESULT CALLBACK
HookFunction(int code, WPARAM wParam, LPARAM lParam) {
    if (code == HC_ACTION) {
        auto msg = (MSG*)lParam;
        if (msg->message == WM_CHAR) {
            ::PostThreadMessage(g_ThreadId,
                WM_APP, msg->wParam, msg->lParam);
            // prevent 'A' characters from getting to the app
            //if (msg->wParam == 'A' || msg->wParam == 'a')
            //    msg->wParam = 0;
        }
    }
    return ::CallNextHookEx(g_hHook, code, wParam, lParam);
}
```

Every hook function used with `SetWindowsHookEx` has the same prototype, but the rules are not the same. You should always read carefully the documentation for the particular hook callback. In our case (`WH_GETMESSAGE`), if `code` is `HC_ACTION`, the hook should process the notification. The message is packed into the `lParam` value. If the message is `WM_CHAR`, indicating a "printable" character, the callback posts a message to the injector's thread with the message information (`wParam` holds the key itself).

The commented code shows how simple it is to change the message so that the key never reaches *Notepad*'s thread. Finally, the call to `CallNextHookEx` is recommended to allow other hook functions that may be in the chain of hooks to get a chance to do their work. However, this is not mandatory.

API Hooking

The term *API Hooking* refers to the act of intercepting Windows APIs (or, more generally, any external function), so that its arguments can be inspected and its behavior possibly changed. This is an extremely powerful technique, employed first and foremost by anti-malware solutions, that typically inject their own DLL into every process (or most processes) and hook certain functions that they care about, such as `VirtualAllocEx` and `CreateRemoteThread`, redirecting them to an alternate implementation provided by their DLL. In that implementation, they can check parameters and do whatever they need before returning a failure code to the caller or forwarding the call to the original function.

In this section, we'll take a look at two common techniques to hook functions.

IAT Hooking

Import Address Table (IAT) hooking is probably the simplest approach to function hooking. It's relatively simple to set up and does not require any platform-specific code.

Every PE image has an import table that lists the DLLs it depends on, and the functions it uses from DLLs. You can view these imports by examining the PE file with *Dumpbin* or a graphical tool. Here is an excerpt from *Notepad.exe*'s modules:

```
dumpbin /imports c:\Windows\System32\notepad.exe
Microsoft (R) COFF/PE Dumper Version 14.26.28805.0
Copyright (C) Microsoft Corporation.  All rights reserved.

Dump of file c:\Windows\System32\notepad.exe

File Type: EXECUTABLE IMAGE

  Section contains the following imports:

    KERNEL32.dll
             1400268B0 Import Address Table
             14002D560 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference

                         2B7 GetProcAddress
                          DB CreateMutexExW
                           1 AcquireSRWLockShared
                         113 DeleteCriticalSection
                         220 GetCurrentProcessId
                         2BD GetProcessHeap
...
    GDI32.dll
             1400267F8 Import Address Table
             14002D4A8 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference

                          34 CreateDCW
                         39F StartPage
                         39D StartDocW
                         366 SetAbortProc
...
    USER32.dll
             140026B50 Import Address Table
             14002D800 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference

                         157 GetFocus
                         2AF PostMessageW
                         177 GetMenu
```

```
                         43 CheckMenuItem
...
```

The output above shows the functions used by *Notepad* from each module *Notepad* depends on. Each module, in turn, has its own import table. Here is the example for *User32.dll*:

```
dumpbin /imports c:\Windows\System32\User32.dll
Microsoft (R) COFF/PE Dumper Version 14.26.28805.0
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file c:\Windows\System32\user32.dll

File Type: DLL

  Section contains the following imports:

    win32u.dll
             180092AD0 Import Address Table
             1800AA0B0 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference

                         297 NtMITSetInputDelegationMode
                         29B NtMITSetLastInputRecipient
                         363 NtUserEnableScrollBar
                         4FA NtUserTestForInteractiveUser
                         501 NtUserTransformRect
                         384 NtUserGetClassName
...
    ntdll.dll
             180092700 Import Address Table
             1800A9CE0 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference

                         8C5 __chkstk
                         95F toupper
                         936 memcmp
                         96A wcscmp
                         937 memcpy
                         5A1 RtlSetLastWin32Error
                          BB NlsAnsiCodePage
```

```
...
    GDI32.dll
             180091C58 Import Address Table
             1800A9238 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference

                         309 PatBlt
                         36C SetBkMode
                         364 SelectObject
                         2E3 IntersectClipRect
...
```

The way these imported functions are called is through the *Import Address Table*, which contains the final addresses of these functions once the loader (*NtDll.Dll*) has mapped them at runtime. These addresses are not known in advance, since the DLLs may not load at their preferred address (see the section "DLL Base Address", later in this chapter).

IAT hooking exploits the fact that all calls are indirect, and just replaces the function address in the table at runtime to point to an alternate function, while saving the original address so that implementation can be invoked if desired. This hooking can be done on the current process or combined with DLL injection can be performed in another process' context.

The functions to hook must be searched in all process modules because each module has its own IAT. For example, the function `CreateFileW` can be called by the *Notepad.exe* module itself, but it can also be called by *ComCtl32.dll* when the Open File dialog box is invoked. If only *Notepad*'s invocations are of interest, its IAT is the only one that needed to be hooked. Otherwise, all loaded modules must be searched and their IAT entry for `CreateFileW` must be replaced.

To demonstrate this technique, I've copied the *Working Sets* application from chapter 13 to this chapter's Visual Studio solution. We will hook the `GetSysColor` API from *User32.Dll* for demonstration purposes and change a couple of colors in the application without touching the application's UI code.

In the `WinMain` function, we call a helper function presented later to do the hooking. First, we need a variable to save the original function pointer:

```
decltype(::GetSysColor)* GetSysColorOrg;
```

The `decltype` keyword (C++ 11+) saves typing and errors by obtaining the correct type of the expression in parenthesis, in this case the type of `GetSysColor`. Now we can start by getting the original function:

```
void HookFunctions() {
    auto hUser32 = ::GetModuleHandle(L"user32");
    // save original functions
    GetSysColorOrg = (decltype(GetSysColorOrg))::GetProcAddress(
        hUser32, "GetSysColor");
```

And the hooking itself is simple because of some helpers functions we'll soon see:

```
    auto count = IATHelper::HookAllModules("user32.dll",
        GetSysColorOrg, GetSysColorHooked);
    ATLTRACE(L"Hooked %d calls to GetSysColor\n");
}
```

`GetSysColorHooked` is our hook replacement function for `GetSysColor`. It must have the same prototype as the original. Here is our custom implementation:

```
COLORREF WINAPI GetSysColorHooked(int index) {
    switch (index) {
        case COLOR_BTNTEXT:
            return RGB(0, 128, 0);

        case COLOR_WINDOWTEXT:
            return RGB(0, 0, 255);
    }

    return GetSysColorOrg(index);
}
```

The hooked function returns different colors for a couple of indices, and invokes the original function for all other inputs.

The secret, of course, is in `IATHelper::HookAllModules`. This function and another helper is part of a static library named *IATHelper*, also part of the same solution and linked to the *WorkingSets* project. Here is the class declaration:

```
// IATHelper.h

struct IATHelper final abstract {
    static int HookFunction(PCWSTR callerModule, PCSTR moduleName,
        PVOID originalProc, PVOID hookProc);
    static int HookAllModules(PCSTR moduleName, PVOID originalProc,
        PVOID hookProc);
};
```

`HookFunction`'s task is to hook a single function called by a single module.
`HookAllModules` iterates over all currently loaded modules in the process and invokes `HookFunction`. `HookAllModules` accepts the module name in which the function to hook is exported from (*user32.dll* in our case). Notice it's passed as an ASCII string rather than Unicode, because the module names are stored in ASCII in the import table. The next parameters are the original function (so it can be located in the import tables), and the new function to replace the old one.

```
int IATHelper::HookAllModules(PCSTR moduleName, PVOID originalProc,
    PVOID hookProc) {
    HMODULE hMod[1024];    // should be enough (famous last words)
    DWORD needed;
    if (!::EnumProcessModules(::GetCurrentProcess(), hMod,
        sizeof(hMod), &needed))
        return 0;

    assert(needed <= sizeof(hMod));

    WCHAR name[256];
    int count = 0;
    for (DWORD i = 0; i < needed / sizeof(HMODULE); i++) {
        if (::GetModuleBaseName(::GetCurrentProcess(),
            hMod[i], name, _countof(name))) {
            count += HookFunction(
                name, moduleName, originalProc, hookProc);
        }
    }

    return count;
}
```

The function is fairly simple. It enumerates the modules in the current process with the PSAPI function `EnumProcessModules` declared like so:

```
BOOL EnumProcessModules(
    _In_ HANDLE hProcess,
    _Out_ HMODULE* lphModule,
    _In_ DWORD cb,
    _Out_ LPDWORD lpcbNeeded);
```

`EnumProcessModules` fills the provided `lphModule` array up to the size set by `cb`, and returns the required size in `lpcbNeeded`. If the returned size is greater than `cb`, then some modules were not returned, and the caller should re-allocate and enumerate again. In my code, I've assumed no more than 1024 modules for simplicity, but in production-level code this could miss modules. The process handle must have the `PROCESS_QUERY_INFORMATION` access mask, which is never an issue for the current process.

`GetModuleBaseName` is another function from PSAPI that returns the base name (excludes any path) of a module:

```
DWORD GetModuleBaseName(
    _In_ HANDLE hProcess,
    _In_opt_ HMODULE hModule,
    _Out_ LPTSTR lpBaseName,
    _In_ DWORD nSize);
```

The `HookFunction` does all the hard work. It first gets the caller's module handle to be used as the basis for accessing its import table:

```
int IATHelper::HookFunction(PCWSTR callerModule, PCSTR moduleName,
    PVOID originalProc, PVOID hookProc) {
    HMODULE hMod = ::GetModuleHandle(callerModule);
    if (!hMod)
        return 0;
```

Now comes the tricky bits. The import table must be located by parsing the PE file. Fortunately, some of this parsing logic is available by some APIs from *dbghelp* and *imagehlp*. In this case, a *dbghelp* function is used to get to the import table quickly:

```
ULONG size;
auto desc = (PIMAGE_IMPORT_DESCRIPTOR)::ImageDirectoryEntryToData(
    hMod, TRUE, IMAGE_DIRECTORY_ENTRY_IMPORT, &size);
if (!desc)    // no import table
    return 0;
```

> Discussion of the PE file format is outside the scope of this chapter. More information can be found in Appendix A. The full specification is documented at https://docs.microsoft.com/en-us/windows/win32/debug/pe-format. Quite a few articles and blogs cover the format well.

`ImageDirectoryEntryToData` returns one of the so-called *data directories*, that are part of an image PE. Here is its declaration:

```
PVOID ImageDirectoryEntryToData (
    _In_ PVOID Base,
    _In_ BOOLEAN MappedAsImage,
    _In_ USHORT DirectoryEntry,
    _Out_ PULONG Size);
```

`Base` is the base address of the module and as we know that's the module's "handle". `MappedAsImage` should be set to `TRUE`, because the image is mapped to the address space as a true image, rather than loaded as a data file. `DirectoryEntry` is the index of the data directory to retrieve, and `Size` returns the data directory's size. The return value from the function is the virtual address where the data directory is located.

The import table for a specific module may contain many imported libraries this module depends on. We need to locate just the one where the function is. It's *user32.dll* in the *WorkingSet* example. The `IMAGE_IMPORT_DESCRIPTOR` is the structure the heads the import library, from which our search begins:

```
int count = 0;
for (; desc->Name; desc++) {
    auto modName = (PSTR)hMod + desc->Name;
    if (::_stricmp(moduleName, modName) == 0) {
```

The code loops through all the modules and compares their names to the module in question. The module name is stored as ASCII, which is why we passed it to `HookFunction` as such. The `Name` member of `IMAGE_IMPORT_DESCRIPTOR` is the offset where the module name is stored from the module's beginning.

Now that the module is found, we need to iterate over all imported functions, looking for our original function pointer:

```
auto thunk = (PIMAGE_THUNK_DATA)((PBYTE)hMod + desc->FirstThunk);
for (; thunk->u1.Function; thunk++) {
    auto addr = &thunk->u1.Function;
    if (*(PVOID*)addr == originalProc) {
        // found it
```

The code is not very pretty and is based on the data structures that make up information in the PE - `IMAGE_THUNK_DATA` in this case. Each one of these "thunks" stores the address of a function, and so we compare it with the original function we received. If they are equal, we have our match:

```
            DWORD old;
            if (::VirtualProtect(addr, sizeof(void*),
                PAGE_WRITECOPY, &old)) {
                *(void**)addr = (void*)hookProc;
                count++;
            }
        }
    }
    break;
}
```

We need to replace the existing value with the new value. However, the pages of the import table are protected with `PAGE_READONLY`, so we must replace that with `PAGE_WRITECOPY` to get our own writable copy of the page we have to access.

> The structures and layout of members is based on the PE file format documentation.

Combine API hooking with DLL injection from the previous section to hook `GetSysColor` in a *Notepad* process, rather than the current process. Try hooking other functions!

What's the downside of IAT based hooking? First, if a new module is loaded later, it must be hooked as well. Paradoxically, this can be done by hooking `LoadLibraryW`, `LoadLibraryExW` and `LdrLoadDll` (undocumented from *NtDll.dll*, but possibly used).
Second, It's easy to circumvent by avoiding the IAT - by calling the API directly with the function pointer returned from `GetProcAddress`. This means calling the original `GetSysColor` function could have been done with this code:

```
return ((decltype(::GetSysColor)*)::GetProcAddress(
    GetModuleHandle(L"user32"), "GetSysColor"))(index);
```

If the hooking is done for security purposes, then IAT hooking is probably unacceptable because it's easily circumvented. If the need is for something else, where normal code uses functions without calling `GetProcAddress`, IAT hooking is convenient and reliable.

"Detours" Style Hooking

The other common way of hooking a function is by following these steps:

- Locate the original function's address and save it.

- Replace the first few bytes of the code with a JMP assembly instruction, saving the old code.
- The JMP instruction calls the hooked function.
- If the original code is to be invoked, do so with the saved address from the first step.
- When unhooking, restore the modified bytes.

This scheme is more powerful that IAT hooking because the real function code is modified, whether it's called through the IAT or not. There are two drawbacks to this method:

- The replaced code is platform-specific. The code for x86, x64, ARM and ARM64 is different, making it more difficult to get right.
- The steps above must be done atomically. There could be some other thread in the process that invokes the hooked function just as assembly bytes are being replaced. This is likely to cause a crash.

Implementing this hooking is difficult, and requires intricate knowledge of CPU instructions and calling conventions, not to mention the synchronization problem above.

There are several open-source and free libraries that provide this functionality. One of them is called "Detours" from Microsoft (hence the section name), but there are others like *MinHook* and *EasyHook* which you can find online. If you need this kind of hooking, consider using an existing library rather than rolling your own.

To demonstrate hooking with the *Detours* library, we'll use the *Basic Sharing* application from chapter 14. We'll hook two functions: `GetWindowTextLengthW` and `GetWindowTextW`. With the hooked functions, the code will return a custom string for edit controls only.

The first step is to add support for *Detours*. Fortunately, this is easy through *Nuget* - just search for "detours" and you shall receive (figure 15-13).

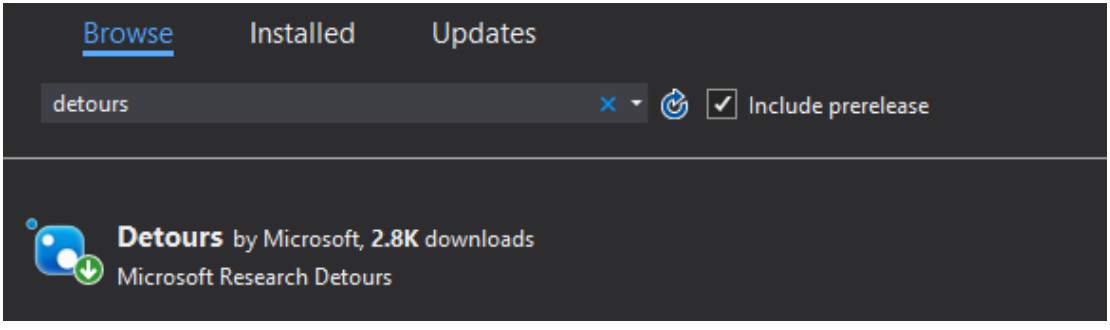


Figure 15-13: *Detours* library in Nuget package manager

Setting up the hooks is fairly straightforward. Here is the helper function that does it:

```
#include <detours.h>

bool HookFunctions() {
    DetourTransactionBegin();
    DetourUpdateThread(GetCurrentThread());
    DetourAttach((PVOID*)&GetWindowTextOrg, GetWindowTextHooked);
    DetourAttach((PVOID*)&GetWindowTextLengthOrg,
        GetWindowTextLengthHooked);
    auto error = DetourTransactionCommit();
    return error == ERROR_SUCCESS;
}
```

Detours works with the concept of *transactions* - a set of operations that are committed to be executed atomically. We need to save the original functions. This can be done with `GetProcAddress` prior to hooking or right with the pointers definitions:

```
decltype(::GetWindowTextW)* GetWindowTextOrg = ::GetWindowTextW;
decltype(::GetWindowTextLengthW)* GetWindowTextLengthOrg =
    ::GetWindowTextLengthW;
```

The hooked functions provide some implementation. Here is what this demo does:

```
static WCHAR extra[] = L" (Hooked!)";

bool IsEditControl(HWND hWnd) {
    WCHAR name[32];
    return ::GetClassName(hWnd, name, _countof(name)) &&
        ::_wcsicmp(name, L"EDIT") == 0;
}

int WINAPI GetWindowTextHooked(
    _In_  HWND   hWnd,
    _Out_ LPWSTR lpString,
    _In_  int    nMaxCount) {

    auto count = GetWindowTextOrg(hWnd, lpString, nMaxCount);

    if (IsEditControl(hWnd)) {
        if (count + _countof(extra) <= nMaxCount) {
            ::StringCchCatW(lpString, nMaxCount, extra);
            count += _countof(extra);
        }
    }
```

```
    return count;
}

int WINAPI GetWindowTextLengthHooked(HWND hWnd) {
    auto len = GetWindowTextLengthOrg(hWnd);
    if(IsEditControl(hWnd))
        len += (int)wcslen(extra);
    return len;
}
```

The hooked `GetWindowTextW` adds the extra string to edit controls only. If you run *Basic Sharing* now, type "hello" in the edit box, click *Write* and click *Read*, you'll get what figure 15-14 shows.

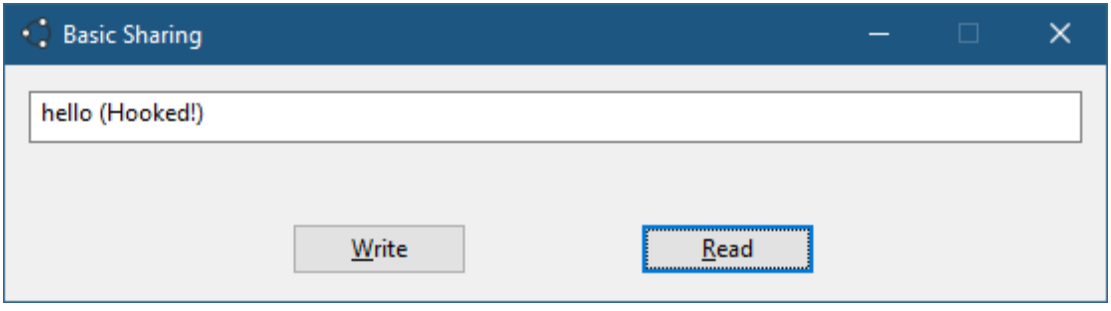


Figure 15-14: A hooked *Basic Sharing*

The *Write* button click handler calls `GetDlgItemText` which calls `GetWindowText`, which invokes the hooked function.

Use *Detours* to hook *Notepad* similarly to *Basic Sharing*.

DLL Base Address

Every DLL has a preferred load (base) address, that is part of the PE header. It can even be specified using the project's properties in Visual Studio (figure 15-15).

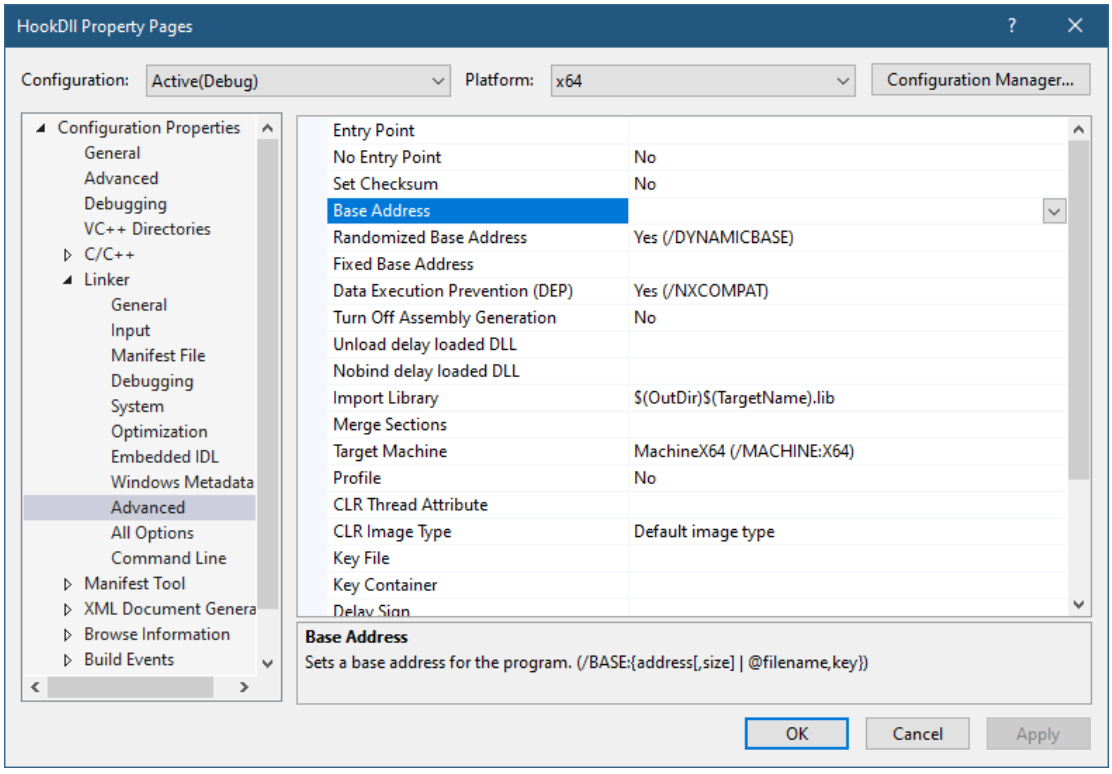


Figure 15-15: Setting a DLL's base address in Visual Studio

There is nothing there by default, which makes Visual Studio use some default values. These are `0x10000000` for 32-bit DLLs and `0x180000000` for 64-bit DLLs. You can verify these by dumping header information from the PE:

```
dumpbin /headers c:\dev\Win10SysProg\Chapter15\x64\Debug\HookDll.dll
...
OPTIONAL HEADER VALUES
        20B magic # (PE32+)
...
      112FD entry point (00000001800112FD) @ILT+760(_DllMainCRTStartup)
       1000 base of code
  180000000 image base (0000000180000000 to 0000000180025FFF)
...

dumpbin /headers c:\dev\Win10SysProg\Chapter15\Debug\HookDll.dll
...
OPTIONAL HEADER VALUES
        10B magic # (PE32)
...
```

```
      111B8 entry point (100111B8) @ILT+435(__DllMainCRTStartup@12)
       1000 base of code
       1000 base of data
   10000000 image base (10000000 to 1001FFFF)
...
```

In the old days (pre-Vista), when *Address Space Load Randomization* (ASLR) didn't exist, DLLs insisted on loading to their preferred address. If that address was already occupied by some other DLL or data, the DLL went through *relocation*. The loader had to find a new location for the DLL in the process address space. Furthermore, it would have to perform code fixups (stored in the PE by the linker), because some code had to change. For example, a string that was expected to be in address `x` now has moved to some other address `y`, that the loader had to fix. These fixes take time, and extra memory because the code in that page can no longer be shared.

In *Process Explorer*, relocated DLLs are easy to spot. First, there is a yellowish color you can enable called "Relocated DLLs" and appears in the Modules (DLLs) view for every relocated DLLs. Second, the sure way to recognize relocated DLLs is where the *Image Base* column is different than *Base* (figure 15-16).

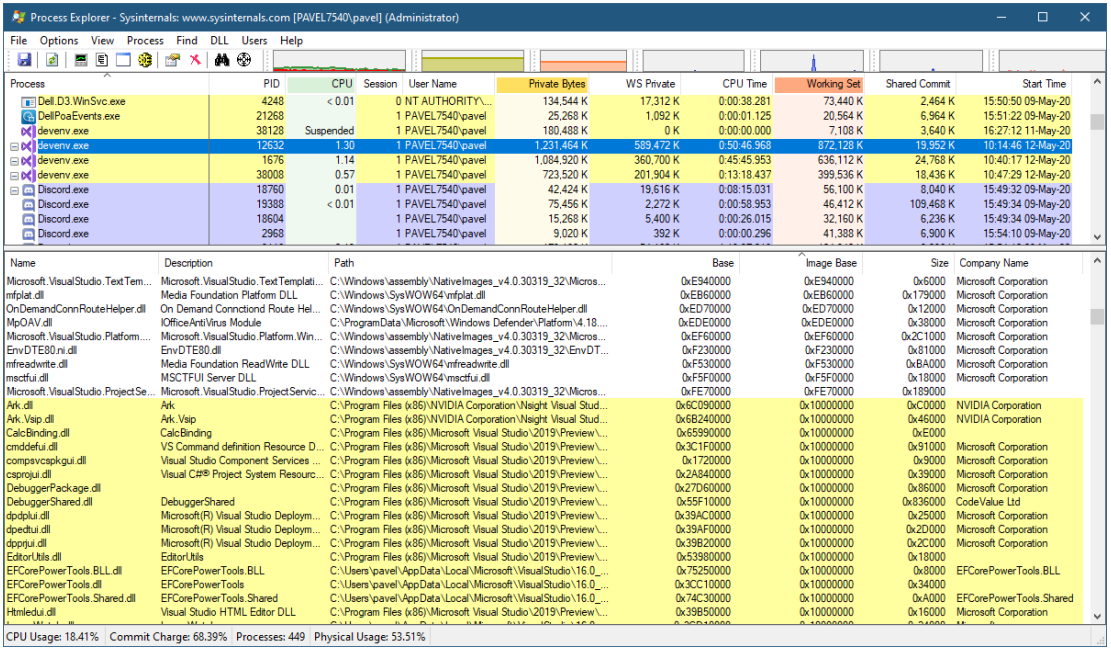


Figure 15-16: Relocated DLLs in *Process Explorer*

I've chosen Visual Studio's process (*devenv.exe*) in figure 15-16, which shows lots of relocated DLLs. However, this is not as common as it used to be in the old days. Most processes have very few relocated DLLs.

The solution to the relocation problem was selecting different addresses for different DLLs, to minimize the chance of collisions. This was sometimes done with the *rebase.exe* tool, part of the Windows SDK, that can perform the operation on multiple DLLs at the same time. The tool does not require source code,

since it manipulates the PE. The functionality of the *rebase.exe* tool is available programmatically with the *ReBaseImage64* function from the debug help API.

Most DLLs have the *Dynamic Base* characteristic flag that indicates the DLL is ok with relocation. With ASLR, the loader selects an address that does not conflict with previously selected DLLs, so the likelihood of collisions is very low, since the addresses used start for a high address and move down for each loaded DLL. A DLL can specify it wants a fixed address so that relocation is forbidden, but that may cause the DLL to fail to load if its preferred address range is taken. There should be a very good reason for a DLL to insist on a fixed base address.

Delay-Load DLLs

We have examined the two primary ways to link to a DLL: either implicit linking with a LIB file (easiest and most convenient), and dynamic linking (loading the DLL explicitly and locating functions to use). It turns out there is a third way, sort of a "middle ground" between static and dynamic linking - delay-load DLLs.

With delay-loading, we get the benefits of both options: the convenience of static linking with dynamic loading of the DLL only in case it's needed.

To use delay-load DLLs, some changes are required to the modules that use these DLLs, whether that's an executable or another DLL. The DLLs that should be delay-loaded are added to the linker's options in the *Input* tab (figure 15-17).

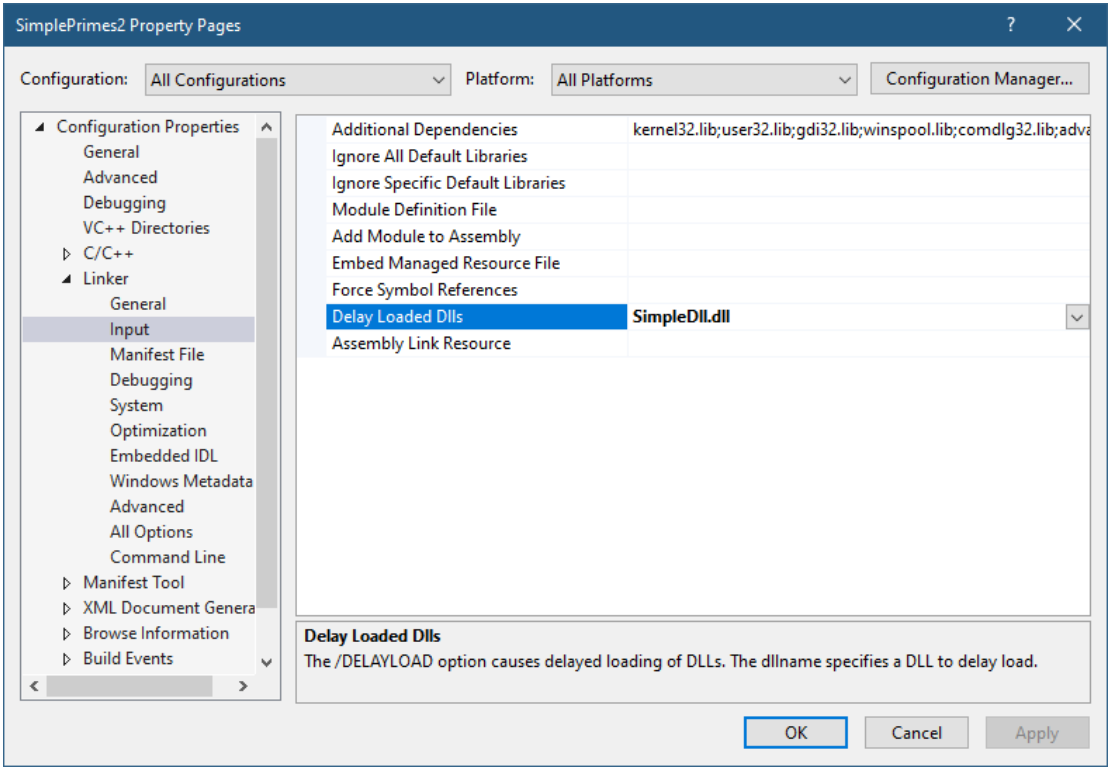


Figure 15-17: Specifying delay-load DLLs in project's properties

If you want to support dynamic unloading of delay-load DLLs, add that option in the *Advanced* linker tab ("Unload delay loaded DLL").

All that's left is to link with the import libraries (LIB) files of the DLLs and use the exported functionality just like you would an implicitly linked DLL.

Here is an example from the project *SimplePrimes2* that delay-load links with *SimpleDll.Dll*:

```
#include "..\SimpleDll\Simple.h"
#include <delayimp.h>

bool IsLoaded() {
    auto hModule = ::GetModuleHandle(L"simpledll");
    printf("SimpleDll loaded: %s\n", hModule ? "Yes" : "No");
    return hModule != nullptr;
}

int main() {
    IsLoaded();
```

```
    bool prime = IsPrime(17);
    IsLoaded();

    printf("17 is prime? %s\n", prime ? "Yes" : "No");
    __FUnloadDelayLoadedDLL2("SimpleDll.dll");

    IsLoaded();
    prime = IsPrime(1234567);
    IsLoaded();

    return 0;
}
```

The strange-looking function `__FUnloadDelayLoadedDLL2` (from *delayimp.h*) is the one to use to unload a delay-load DLL. If you call `FreeLibrary`, the DLL will unload; however it will not load again if needed just by invoking an exported function, and instead will throw an access violation exception.

Running the above program shows:

```
SimpleDll loaded: No
SimpleDll loaded: Yes
17 is prime? Yes
SimpleDll loaded: No
SimpleDll loaded: Yes
```

When you call an exported function from a delay-load DLL, there is a different function that is being called (provided by the delay-load infrastructure), that knows to call `LoadLibrary` and `GetProcAddress` and then invoke the function, fixing the import table so that future calls to the same function go directly to its implementation. This also explains why unloading a delay-loaded DLL must be done with a special function that can restore the initial behavior.

The `LoadLibraryEx` Function

`LoadLibraryEx` is an extended `LoadLibrary` function defined like so:

```
HMODULE LoadLibraryEx(
    _In_ LPCTSTR lpLibFileName,
    _Reserved_ HANDLE hFile,    // must be NULL
    _In_ DWORD dwFlags);
```

`LoadLibraryEx` has the same purpose as `LoadLibrary`: to load a DLL explicitly. As you can see, `LoadLibraryEx` supports a set of flags that affect the way the DLL in question is searched for and/or loaded. Some of the acceptable flags are from table 15-1, where the search path and order can be modified to some extent. Here are some other possibly useful flags (check the documentation for a complete list):

- `LOAD_LIBRARY_AS_DATAFILE` - this flag indicates the DLL should just be mapped to the process address space, but its PE image properties disregarded. `DllMain` is not called, and functions such as `GetModuleHandle` or `GetProcAddress` fail on the returned handle.
- `LOAD_LIBRARY_AS_DATAFILE_EXCLUSIVE` - similar to `LOAD_LIBRARY_AS_DATAFILE`, but the file is opened with exclusive access so that other processes cannot modify it while it's loaded.
- `LOAD_LIBRARY_AS_IMAGE_RESOURCE` - loads the DLL as an image file, but does not perform any initialization such as calling `DllMain`. This flag is usually specified with `LOAD_LIBRARY_AS_DATAFILE` for the purpose of extracting resources.
- `LOAD_WITH_ALTERED_SEARCH_PATH` - if the path specified is absolute, that path is used as a basis for search.

When using `LOAD_LIBRARY_AS_IMAGE_RESOURCE`, the returned handle can be used to extract resources with APIs such as `LoadString`, `LoadBitmap`, `LoadIcon`, and similar. Custom resources are supported as well, and in that case the functions to use include `FindResource(Ex)`, `SizeOfResource`, `LoadResource` and `LockResource`.

Miscellaneous Functions

In this section, we'll go briefly over some other DLL-related functions you might find useful. We'll start with `GetModuleFileName` and `GetModuleFileNameEx`:

```
DWORD GetModuleFileName(
    _In_opt_ HMODULE hModule,
    _Out_ LPTSTR lpFilename,
    _In_ DWORD nSize);
DWORD GetModuleFileNameEx(
    _In_opt_ HANDLE hProcess,
    _In_opt_ HMODULE hModule,
    _Out_    LPWSTR lpFilename,
    _In_     DWORD nSize);
```

Both functions return the full path of a loaded module. `GetModuleFileNameEx` can access such information in another process (the handle must have the `PROCESS_QUERY_INFORMATION` or `PROCESS_QUERY_LIMITED_INFORMATION` access mask). If `hModule` is `NULL`, the main module path is returned (the executable path). If a list of modules are needed, `EnumProcessModules` can be used (shown earlier in this chapter).

The `LoadPackagedLibrary` (Windows 8+) is a variant on `LoadLibrary` that may be used by a UWP process to load a DLL which is part of its package:

```
HMODULE LoadPackagedLibrary (
    _In_       LPCWSTR lpwLibFileName,
    _Reserved_ DWORD Reserved);     // must be zero
```

If a thread needs to unload a DLL in which its code is running, it cannot use the function pair `FreeLibrary` and then `ExitThread`, because once `FreeLibrary` returns, the thread's code would no longer be part of the process, and a crash would occur. To solve this, use `FreeLibraryAndExitThread`:

```
VOID FreeLibraryAndExitThread(
    _In_ HMODULE hLibModule,
    _In_ DWORD dwExitCode);
```

The function frees the specified module and then calls `ExitThread`, which means this function never returns.

Summary

In this chapter, we looked at DLLs - how to build them and how to use them. We've also seen the ability to inject a DLL into another process, which gives that DLL a lot of power in the target process.

In the next chapter, we'll turn our attention to a completely different topic: security.

Chapter 16: Security

Windows NT was built with security in mind from its initial design. That was contrary to the Windows 95/98 family of operating systems that did not support security. Windows adheres to the C2 level security, defined by the US Department of Defense. C2 is the highest level a general-purpose operating system can attain.

Some of the requirements of C2 are as follows:

- Logging into the system must require some form of authentication.
- A dead process' memory should never be revealed to another process.
- There must be a file system with the ability to set permissions on file system objects.

In this chapter, we'll examine the foundations of Windows security and look at many of the APIs used to manipulate it.

In this chapter:

Introduction

There are several components in Windows that are related to security. Figure 16-1 shows most of them.

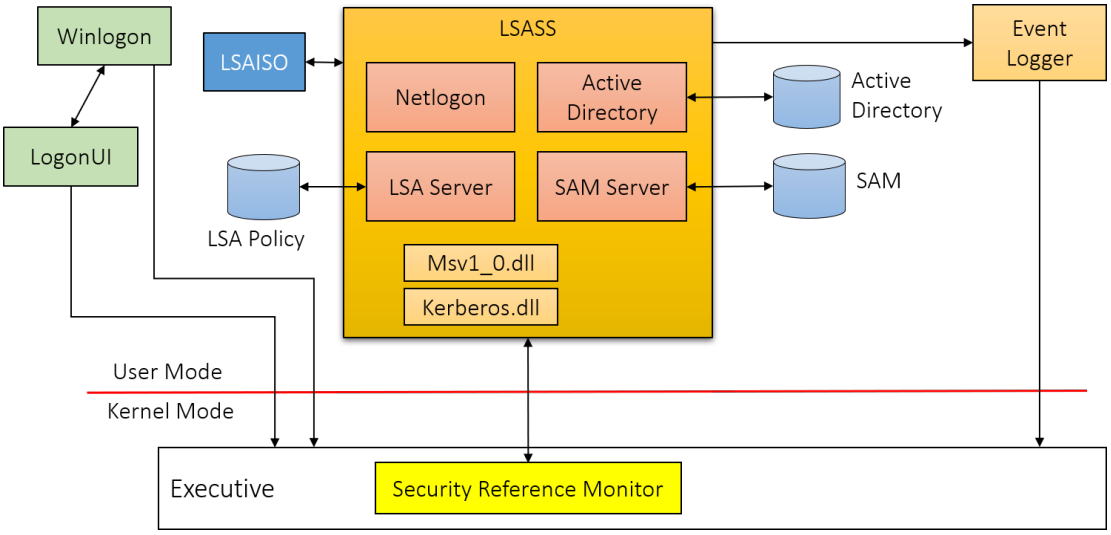


Figure 16-1: Security-related components

Here is a quick rundown of the major components in figure 16-1:

WinLogon

The logon process (*Winlogon.exe*) is responsible for interactive logons. It's also responsible for responding to the *Secure Attention Sequence* (SAS, by default *Ctrl+Alt+Del*) key combination by switching to the *Winlogon* desktop where it shows the familiar options (lock, switch user, sign out, etc.).
Winlogon gets the credentials from the user with a helper process, *LogonUI.exe* (see next section), and sends them to *Lsass.exe* for authentication. If authentication succeeds, *Lsass* creates a logon session and an *access token* (discussed later in this chapter), that represents the security context of the user. Then it creates the startup process (by default *userinit.exe*, read from the Registry), which in turn creates *Explorer.exe* (again, this is just the default in the Registry). The access token is duplicated for each newly created process. Figure 16-2 shows part of a system's initialization process tree, generated with the *Process Monitor* (*ProcMon.exe*) tool from *Sysinternals*. Notice the connection between *WinLogon*, *UserInit* and *Explorer*.

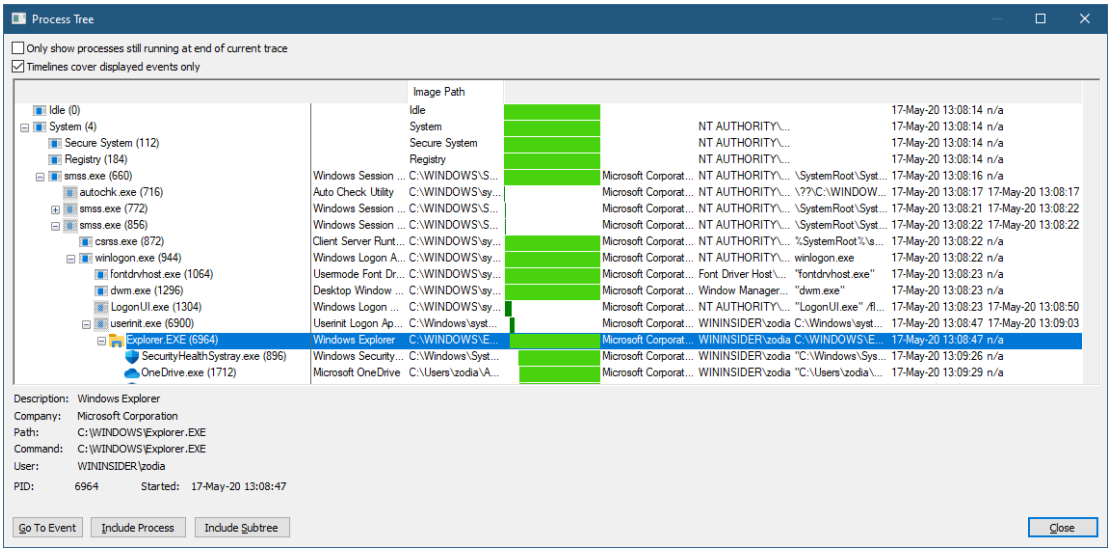


Figure 16-2: Logon process tree

Userinit also runs startup scripts and then terminates (this is why *Explorer* is normally parentless).

The Registry key with the above-mentioned settings is *HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon*.

LogonUI

Windows support several ways to log in: username/password, "Windows Hello" using face recognition and fingerprint to name a few. The *LogonUI.exe* process is launched by *Winlogon* to present the selected authentication method UI. In the old days (prior to Windows Vista), *Winlogon* was in charge of that (*LogonUI* didn't exist). The problem was that if the UI component crashed, it would bring *Winlogon* down with it. Starting with Vista, if the UI crashes, *LogonUI* is terminated, and *Winlogon* allows the user to select a different method for authentication.

LogonUI displays the UI provided by a *Credential Provider* - a COM DLL that implements some user interface for its associated authentication mechanism. Credential providers may be developed by anyone where a custom method for authentication is needed.

LSASS

The *Local Security Authentication Service* (*lsass.exe*) is the cornerstone of authentication management. Its most fundamental role is to authenticate the user. In the common case of a username/password combination, *Lsass* examines the local Registry (in case of a local logon), or communicates with a *Domain Controller* (domain logon), to authenticate the user. Then it returns the result to *Winlogon*. If the authentication succeeded, it creates and populates the access token for the logged-in user.

LsaIso

The *LsaIso.exe* process exists in a Windows 10 (or later) system that has *Virtualization Based Security* (VBS) active with the *Credential Guard* feature turned on. *LsaIso* is known as a *trustlet*, a user-mode process running in *Virtual Trust Level* (VTL) 1, where the *secure kernel* and the *isolated user mode* (IUM) reside. As an IUM process, *Lsaiso* access is protected by the Hyper-V hypervisor, so it's not accessible even from the kernel. The purpose of *Lsaiso* is to keep secrets for *Lsass*, so certain types of attacks like "pass the hash" are mitigated, since no process (not even admin level) and not even the kernel can peer into *Lsaiso*'s address space.

> There are many new terms in the above paragraph that are not explained in this book, as they have very little to do with system programming. For further info on VBS, Hyper-V, VTL, and related concepts, please consult the "Windows Internals 7th edition Part 1" book and/or online resources.

Security Reference Monitor

The SRM is the part of the executive responsible for access checks needed when certain operations occur. For example, trying to open a handle to an existing kernel object requires an access check, performed by the SRM.

Event Logger

The event logger service is one of the standard services provided by Windows. It's not specific to security, but security events are logged using this service. The common way to view information in the log is using the *Event Viewer* built-in application, shown in figure 16-3 (with the *Security* node selected).

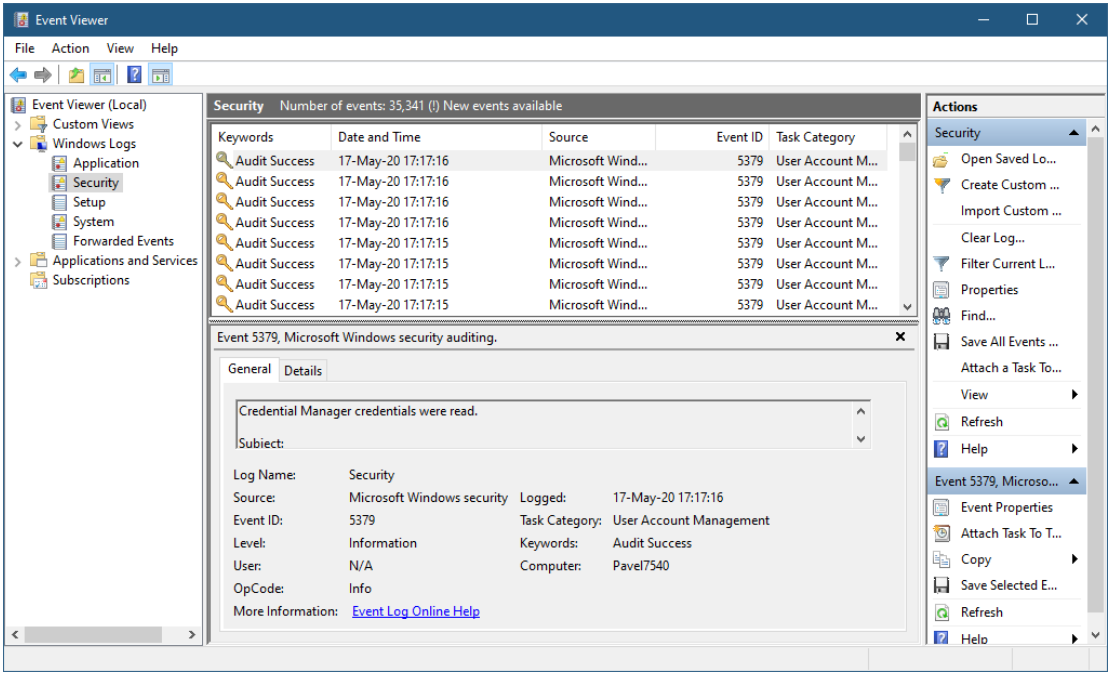


Figure 16-3: The Event Viewer application

SIDs

The term *principal* describes an entity that can be referred to in a security context. For example, permissions can be granted or denied to a principal. Principals can represent users, groups, computers, and more. A principal is uniquely identified by a *Security ID* (SID), which is a variable-sized structure that contains several parts, depicted in figure 16-4.

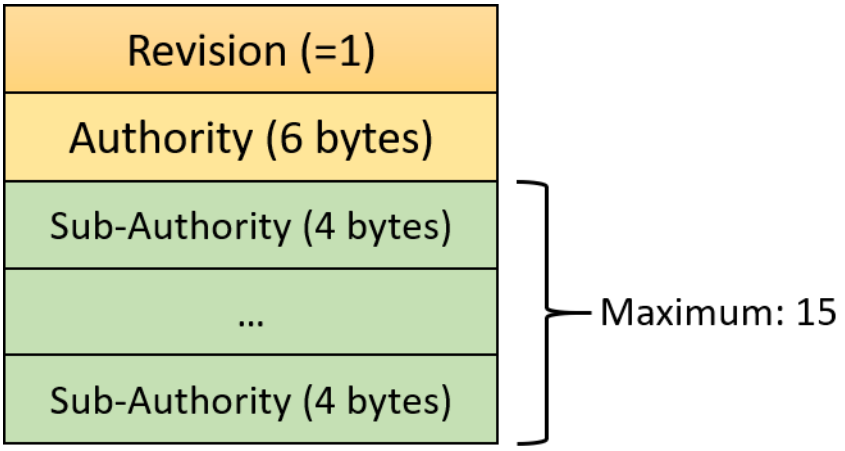


Figure 16-4: SID

When displayed as a string a SID looks like the following:

S-*R*-*A*-*SA*-*SA*-...-*SA*

"S" is a literal S, the revision (R) is always one (the revision never changed), "A" is a 6-byte authority that generated the SID, and "SA" is an array of 4-byte sub-authorities (also called *Relative IDs* - RID) unique to the authority which generated them. Although a SID's size depends on the number of sub-authorities, the maximum count of sub-authorities is currently 15, giving a cap to the size of a SID. This helps in cases where you need to allocate a buffer to hold a SID, and you'd rather allocate it statically rather than dynamically. Here are the definitions from *winnt.h*:

```
#define SID_MAX_SUB_AUTHORITIES          (15)
#define SECURITY_MAX_SID_SIZE  \
(sizeof(SID)-sizeof(DWORD)+(SID_MAX_SUB_AUTHORITIES*sizeof(DWORD)))
// 68 bytes
```

SIDs are typically shown in their string form (it's also usually easier to persist them as strings). Converting between the binary and string forms can be done with the following functions (declared in *<sddl.h>*):

```
BOOL ConvertSidToStringSid(
    _In_ PSID Sid,
    _Outptr_ LPTSTR* StringSid);

BOOL ConvertStringSidToSid(
    _In_ LPCTSTR StringSid,
    _Outptr_ PSID* Sid);
```

Both functions return a result that later must be freed by the caller with `LocalFree`.

Groups and Aliases

A group, represented by a SID, is a collection of principals. Every principal that belongs to the group will have the group SID in its security context (see the section "Tokens" later in this chapter). This helps in modeling organizational structures without specifying individual principles beforehand.

An alias (also called *local group*) is yet another container principal, that can hold a set of groups and other principals (but not other aliases). For example, the *local administrators* group is an alias, that can contain individual SIDs and group SIDs that are all part of the alias. Aliases are always local to a machine and cannot be shared with other machines.

The distinction between groups and aliases is not that important in practice, but is worth keeping in mind nonetheless.

SIDs are guaranteed to be statistically unique when representing different principals. Some SIDs are called "Well-known", as they represent the same principal on every machine. Examples include "S-1-1-0" (the *Everyone* group) and *S-1-5-32-544* (the *Local administrators* alias). *winnt.h* defines an enumeration, `WELL_KNOWN_SID_TYPE`, that holds the list of well-known SIDs. These SIDs exist so it's easy to refer to the same group of principals on any machine. (Imagine what would happen if every machine had its own *local administrators* SID).

Creating a well-known SID is accomplished with `CreateWellKnownSid`:

```
BOOL CreateWellKnownSid(
    _In_ WELL_KNOWN_SID_TYPE WellKnownSidType,
    _In_opt_ PSID DomainSid,
    _Out_ PSID pSid,
    _Inout_ DWORD* cbSid);
```

The `DomainSid` parameter is needed for some types of well-known SIDs. For most, `NULL` is an acceptable value. The returned SID is placed in a caller's allocated buffer. `cbSid` should contain the size of the caller's supplied buffer, and on return stores the actual size of the SID.

We can combine `CreateWellKnownSid` and `ConvertSidToStringSid` to list all well-known SIDs that do not require a domain SID parameter:

```
BYTE buffer[SECURITY_MAX_SID_SIZE];
PWSTR name;

for (int i = 0; i < 120; i++) {
    DWORD size = sizeof(buffer);
    if (!::CreateWellKnownSid((WELL_KNOWN_SID_TYPE)i, nullptr,
        (PSID)buffer, &size))
        continue;
```

```
    ::ConvertSidToStringSid((PSID)buffer, &name);
    printf("Well known sid %3d: %ws\n", i, name);
    ::LocalFree(name);
}
```

The SID buffer is allocated statically with the maximum SID size. Here is a brief output:

```
Well known sid   0: S-1-0-0
Well known sid   1: S-1-1-0
Well known sid   2: S-1-2-0
Well known sid   3: S-1-3-0
...
Well known sid  20: S-1-5-14
Well known sid  22: S-1-5-18
Well known sid  23: S-1-5-19
...
Well known sid  36: S-1-5-32-555
Well known sid  37: S-1-5-32-556
Well known sid  51: S-1-5-64-10
Well known sid  52: S-1-5-64-21
Well known sid  53: S-1-5-64-14
...
Well known sid 118: S-1-18-3
Well known sid 119: S-1-5-32-583
```

The well-known SID `WinLogonIdsSid` (21) cannot be created.

A SID can be checked to see if it matches a specific well-known one with `IsWellKnownSid`:

```
BOOL IsWellKnownSid(
    _In_ PSID pSid,
    _In_ WELL_KNOWN_SID_TYPE WellKnownSidType);
```

We can also get the well-known SID name by using `LookupAccountSid`:

```
BOOL LookupAccountSid(
    _In_opt_ LPCTSTR lpSystemName,
    _In_ PSID Sid,
    _Out_ LPWSTR Name,
    _Inout_ LPDWORD cchName,
    _Out_ LPWSTR ReferencedDomainName,
    _Inout_ LPDWORD cchReferencedDomainName,
    _Out_ PSID_NAME_USE peUse);
```

`lpSystemName` is the machine name for SID lookup, where `NULL` indicates the local machine, and `Sid` is the SID to look up. `Name` is the returned account name, and `cchName` points to the maximum character count accepted by the name. On return, it stores the actual number of characters copied to the buffer. `ReferencedDomainName` and `cchReferencedDomainName` serve the same purpose for the domain name (or more accurately, the authority under which the account resides). Finally, `peUse` returns the type of the SID in question, based on the `SID_NAME_USE` enumeration:

```
typedef enum _SID_NAME_USE {
    SidTypeUser = 1,
    SidTypeGroup,
    SidTypeDomain,
    SidTypeAlias,
    SidTypeWellKnownGroup,
    SidTypeDeletedAccount,
    SidTypeInvalid,
    SidTypeUnknown,
    SidTypeComputer,
    SidTypeLabel,
    SidTypeLogonSession
} SID_NAME_USE, *PSID_NAME_USE;
```

Adding a call to `LookupAccountSid` to the well-known SID iteration looks like the following:

```
WCHAR accountName[64] = { 0 }, domainName[64] = { 0 };
SID_NAME_USE use;

for (int i = 0; i < 120; i++) {
    DWORD size = sizeof(buffer);
    if (!::CreateWellKnownSid((WELL_KNOWN_SID_TYPE)i, nullptr,
        (PSID)buffer, &size))
        continue;

    ::ConvertSidToStringSid((PSID)buffer, &name);
    DWORD accountNameSize = _countof(accountName);
```

```
    DWORD domainNameSize = _countof(domainName);
    ::LookupAccountSid(nullptr, (PSID)buffer, accountName,
        &accountNameSize, domainName, &domainNameSize, &use);

    printf("Well known sid %3d: %-20ws %ws\\%ws (%s)\n", i,
        name, domainName, accountName, SidNameUseToString(use));
    ::LocalFree(name);
}
```

The loop iterates over all the defined well-known SID indices (currently 120 values). There is no static reflection in C++ (yet), so using a number is as good as it gets.

The `SidNameUseToString` helper function converts the `SID_NAME_USE` enumeration to a string. Running this piece of code shows the following:

```
Well known sid   0: S-1-0-0        \NULL SID (Well Known Group)
Well known sid   1: S-1-1-0        \Everyone (Well Known Group)
Well known sid   2: S-1-2-0        \LOCAL (Well Known Group)
Well known sid   3: S-1-3-0        \CREATOR OWNER (Well Known Group)
Well known sid   4: S-1-3-1        \CREATOR GROUP (Well Known Group)
...
Well known sid  22: S-1-5-18       NT AUTHORITY\SYSTEM
    (Well Known group)
Well known sid  23: S-1-5-19       NT AUTHORITY\LOCAL SERVICE
    (Well Known Group)
Well known sid  25: S-1-5-32       BUILTIN\BUILTIN (Domain)
Well known sid  26: S-1-5-32-544   BUILTIN\Administrators (Alias)
Well known sid  27: S-1-5-32-545   BUILTIN\Users (Alias)
Well known sid  28: S-1-5-32-546   BUILTIN\Guests (Alias)
...
Well known sid  65: S-1-16-0       Mandatory Label\Untrusted
    Mandatory Level (Label)
Well known sid  66: S-1-16-4096    Mandatory Label\Low
    Mandatory Level (Label)
Well known sid  67: S-1-16-8192    Mandatory Label\Medium
Mandatory Level (Label)
...
Well known sid  84: S-1-15-2-1     APPLICATION PACKAGE AUTHORITY\
    ALL APPLICATION PACKAGES (Well Known Group)
Well known sid  85: S-1-15-3-1     APPLICATION PACKAGE AUTHORITY\
    Your Internet connection (Well Known Group)
...
Well known sid 117: S-1-18-6       \Key property attestation
    (Well Known Group)
```

```
Well known sid 118: S-1-18-3       \Fresh public key identity
    (Well Known Group)
Well known sid 119: S-1-5-32-583   BUILTIN\Device Owners (Alias)
```

The output clearly shows which SIDs are aliases (rather than being groups).

> The project *wellknownsids* contains the full code. It also has code to retrieve the domain SID, so that more well-known SIDs can be created.

`LookupAccountSid` has a reciprocal function, `LookupAccountName`:

```
BOOL LookupAccountName(
    _In_opt_ LPCTSTR lpSystemName,
    _In_     LPCTSTR lpAccountName,
    _Out_    PSID Sid,
    _Inout_  LPDWORD cbSid,
    _Out_    LPTSTR ReferencedDomainName,
    _Inout_  LPDWORD cchReferencedDomainName,
    _Out_    PSID_NAME_USE peUse);
```

`LookupAccountName` is a hard-working function attempting to locate a name and return its SID. It checks for well-known names first. `lpAccountName` can be a simple name like "joe" or include a domain name ("mydomain\joe", better from a performance standpoint). It also accepts other formats, such as "joe@mydomain.com". It returns the SID in the `Sid` parameter and the domain name in `ReferencedDomainName`, as well as a `SID_NAME_USE` just like `LookupAccountSid`.

The SID APIs include some fairly self-explanatory functions, such as `IsValidSid`, `CopySid`, `EqualSid`, `GetLengthSid`, `AllocatedAndInitializeSid`, `InitializeSid`, `FreeSid`, `GetSidIdentifierAuthority`, `GetSidSubAuthorityCount`, and `GetSidSubAuthority`.

Tokens

When a user logs in successfully, whether interactively or not, a *logon session* object is created behind the scenes, holding information related to the logged in principal. Normally, a logon session is hidden behind an *access token* (or simply *token*), which is an object that maintains several pieces of information and has a pointer to the logon session.

The token is the object developers can interact with and manipulate to some extent. A process always has a token associated with it, known as *primary token*. All threads within a process use that primary token by default when performing operations that require a security context. A thread can perform *impersonation* by assuming a different token, called *impersonation token*. Once a thread is done impersonating, it reverts

to using its process (primary) token. The relationship between a logon session, tokens and processes is depicted in figure 16-6.

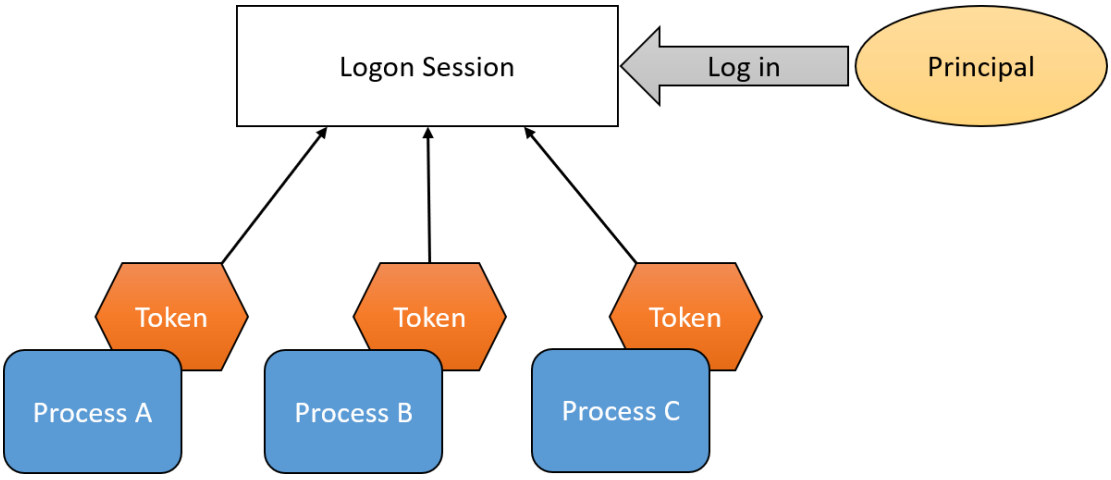


Figure 16-6: A logon session, tokens and processes

> Three logon sessions always exist, used by the three built-in users in Windows - *Local Service*, *Network Service* and *Local System* (also called *SYSTEM*). We'll look at these accounts more closely in chapter 19 ("Services"), which is their primary use by developers.

You may be surprised at how many logon sessions exist at any given time. Run the *logonsessions.exe* tool from *Sysinternals* from an elevated command window. It uses the `LsaEnumerateLogonSessions` API that returns an array of logon session IDs. For each ID, it calls `LsaGetLogonSessionData` to retrieve details about the logon session. Here is an abbreviated output:

```
[0] Logon session 00000000:000003e7:
    User name:    WORKGROUP\PAVEL7540$
    Auth package: NTLM
    Logon type:   (none)
    Session:      0
    Sid:          S-1-5-18
    Logon time:   19-May-20 19:29:32
    Logon server:
    DNS Domain:
    UPN:
...
[2] Logon session 00000000:0001964a:
    User name:    Font Driver Host\UMFD-0
```

```
    Auth package: Negotiate
    Logon type:   Interactive
    Session:      0
    Sid:          S-1-5-96-0-0
    Logon time:   19-May-20 19:29:32
...
[3] Logon session 00000000:000003e5:
    User name:    NT AUTHORITY\LOCAL SERVICE
    Auth package: Negotiate
    Logon type:   Service
    Session:      0
    Sid:          S-1-5-19
    Logon time:   19-May-20 19:29:32
...
[5] Logon session 00000000:000003e4:
    User name:    WORKGROUP\PAVEL7540$
    Auth package: Negotiate
    Logon type:   Service
    Session:      0
    Sid:          S-1-5-20
    Logon time:   19-May-20 19:29:32
...
[7] Logon session 00000000:00023051:
    User name:    Window Manager\DWM-1
    Auth package: Negotiate
    Logon type:   Interactive
    Session:      1
    Sid:          S-1-5-90-0-1
    Logon time:   19-May-20 19:29:32
...
[17] Logon session 00000000:02edfae1:
    User name:    NT VIRTUAL MACHINE\47E3D5AD-77C2-4BCE-AC4F-252E2A6935DA
    Auth package: Negotiate
    Logon type:   Service
    Session:      0
    Sid:          S-1-5-83-1-1206113709-1271822274-774197164-3660933418
    Logon time:   19-May-20 20:14:35
...
```

Normally, a token is created by *Lsass* when a user logs in successfully. *Winlogon* attached it to the first process created in the user's session, and the token is duplicated and propagates from that process to child processes, and so on.

A token contains several pieces of information, some of which are the following:

- The user's SID
- The primary group
- The groups the user is a member of
- The privileges the user has (discussed later)
- The default security descriptor for newly created objects
- Token type (primary or impersonation)

The *primary group* was created for use with the *POSIX* subsystem since this is part of a *NIX security permissions, and so is mostly unused.

Process Explorer allows viewing information for the primary token of a process in its *Security* tab in the process' properties dialog box (figure 16-7).

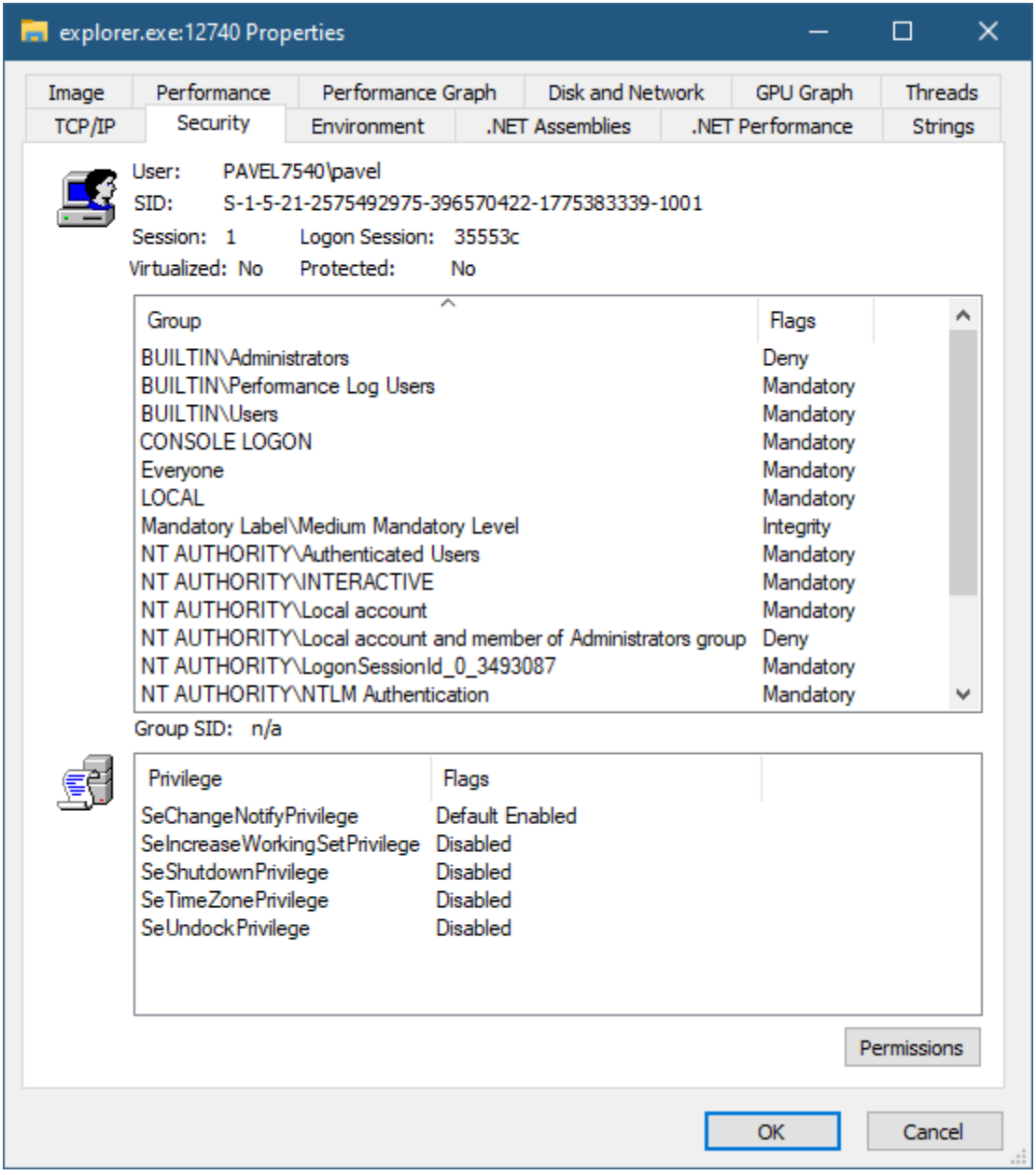


Figure 16-7: The Security tab in *Process Explorer*

Let's start our exploration of tokens by looking at how to get one. Since every process must have a token associated with it (its primary token), a token handle can be opened for a process with `OpenProcessToken`:

```
BOOL OpenProcessToken(
    _In_ HANDLE ProcessHandle,
    _In_ DWORD DesiredAccess,
    _Outptr_ PHANDLE TokenHandle);
```

Before attempting to get a token handle, an open process handle must be obtained (`OpenProcess`) with at least `PROCESS_QUERY_INFORMATION` access mask (of course `GetCurrentProcess` always works). `DesiredAccess` is the access mask requested for the token object. Common values include `TOKEN_QUERY` (query information), `TOKEN_ADJUST_PRIVILEGES` (enable/disable privileges), `TOKEN_ADJUST_DEFAULT` (adjust various defaults), `TOKEN_DUPLICATE` (duplicate the token), and `TOKEN_IMPERSONATE` (impersonate the token). There are other, more powerful, access masks supported, but these cannot normally be granted since they require some powerful privileges (described in the section "Privileges"). Finally, `TokenHandle` returns the actual handle if the call succeeds.

If a thread's token is needed (most likely the thread is impersonating a different token than its process), `OpenThreadToken` can be invoked instead:

```
BOOL OpenThreadToken(
    _In_ HANDLE ThreadHandle,
    _In_ DWORD DesiredAccess,
    _In_ BOOL OpenAsSelf,
    _Outptr_ PHANDLE TokenHandle);
```

`ThreadHandle` is a handle to the thread in question that must have the `THREAD_QUERY_LIMITED_INFORMATION` access mask. Such a handle can be opened with `OpenThread` (`GetCurrentThread` is always OK). `DesiredAccess` is the access required for the returned token handle. `OpenAsSelf` indicates under which token should the access check be made for the retrieval of the token. If `OpenAsSelf` is `TRUE`, the access check is against the process token; otherwise it's against the current thread's token. Of course, this only matters if the current thread is currently impersonating.

With a `TOKEN_QUERY` access mask token handle in hand, `GetTokenInformation` provides a wealth of data stored inside the token:

```
BOOL GetTokenInformation(
    _In_  HANDLE TokenHandle,
    _In_  TOKEN_INFORMATION_CLASS TokenInformationClass,
    _Out_ LPVOID TokenInformation,
    _In_  DWORD TokenInformationLength,
    _Out_ PDWORD ReturnLength);
```

The function is very generic, using the `TOKEN_INFORMATION_CLASS` enumeration to indicate the type of information requested. For each value in the enumeration, the correct buffer size needs to be specified. The last parameter indicates how many bytes were used or needed to complete the operation successfully. The enumeration list is very long. Let's look at a few examples.

To get the user's SID associated with the token, use the `TokenUser` enumeration value like so:

```
BYTE buffer[1 << 12];   // 4KB - should be large enough for anything
if (::GetTokenInformation(hToken, TokenUser, buffer,
    sizeof(buffer), &len)) {
    auto data = (TOKEN_USER*)buffer;
    printf("User SID: %ws\n", SidToString(data->User.Sid).c_str());
}
```

`SidToString` uses the already discussed `ConvertSidToStringSid` function:

```
std::wstring SidToString(const PSID sid) {
    PWSTR ssid;
    std::wstring result;
    if (::ConvertSidToStringSid(sid, &ssid)) {
        result = ssid;
        ::LocalFree(ssid);
    }
    return result;
}
```

Here is another example using the `TokenStatistics` token information class that retrieves some useful stats for the token:

```
TOKEN_STATISTICS stats;
if (::GetTokenInformation(hToken, TokenStatistics,
        &stats, sizeof(stats), &len)) {
    printf("Token ID: 0x%08llX\n", LuidToNum(stats.TokenId));
    printf("Logon Session ID: 0x%08llX\n",
        LuidToNum(stats.AuthenticationId));
    printf("Token Type: %s\n", stats.TokenType == TokenPrimary ?
        "Primary" : "Impersonation");
    if (stats.TokenType == TokenImpersonation)
        printf("Impersonation level: %s\n",
            ImpersonationLevelToString(stats.ImpersonationLevel));

    printf("Dynamic charged (bytes): %lu\n", stats.DynamicCharged);
    printf("Dynamic available (bytes): %lu\n", stats.DynamicAvailable);
    printf("Group count: %lu\n", stats.GroupCount);
    printf("Privilege count: %lu\n", stats.PrivilegeCount);
    printf("Modified ID: %08llX\n\n", LuidToNum(stats.ModifiedId));
}
```

Here are the two helper functions used above:

```
ULONGLONG LuidToNum(const LUID& luid) {
    return *(const ULONGLONG*)&luid;
}

const char* ImpersonationLevelToString(
    SECURITY_IMPERSONATION_LEVEL level) {
    switch (level) {
        case SecurityAnonymous: return "Anonymous";
        case SecurityIdentification: return "Identification";
        case SecurityImpersonation: return "Impersonation";
        case SecurityDelegation: return "Delegation";
    }
    return "Unknown";
}
```

The `TOKEN_STATISTICS` structure is defined like so:

```
typedef struct _TOKEN_STATISTICS {
    LUID TokenId;
    LUID AuthenticationId;
    LARGE_INTEGER ExpirationTime;
    TOKEN_TYPE TokenType;
    SECURITY_IMPERSONATION_LEVEL ImpersonationLevel;
    DWORD DynamicCharged;
    DWORD DynamicAvailable;
    DWORD GroupCount;
    DWORD PrivilegeCount;
    LUID ModifiedId;
} TOKEN_STATISTICS, *PTOKEN_STATISTICS;
```

Some of its members bear explanation. First, there is the `LUID` type, which we have not encountered yet. This is a 64-bit number that is guaranteed to be unique on a particular system while it's running. Here's its definition:

```
typedef struct _LUID {
    DWORD LowPart;
    LONG HighPart;
} LUID, *PLUID;
```

LUIDs are used in several places in Windows, not necessarily related to security. This is just a way to get unique values on a per-machine basis. If you need to generate one, call `AllocateLocallyUniqueId`.

The `TokenId` member is a unique identifier for this token instance (the object, not the handle), so an easy way to compare tokens is by comparing `TokenId` values. `AuthenticationId` is the logon session ID, uniquely identifying the logon session itself.

The logon session IDs for the three built-in logon sessions have well-known IDs: 999 (0x3e7) for *SYSTEM*, 997 (0x3e5) for *Local Service* and 996 (0x3e4) for *Network Service*.

`TokenType` in `TOKEN_STATISTICS` is either `PrimaryToken` (process token) or `ImpersonationToken` (thread token). If the token is an impersonation one, then the `ImpersonationLevel` member has meaning:

```
typedef enum _SECURITY_IMPERSONATION_LEVEL {
  SecurityAnonymous,    // not really used
  SecurityIdentification,
  SecurityImpersonation,
  SecurityDelegation
} SECURITY_IMPERSONATION_LEVEL;
```

The impersonation level indicates what kind of "power" this impersonation token has - from just identifying the user (`SecurityIdentification`) and its properties, to impersonating the user on the server process' machine only (`SecurityImpersonation`), to impersonating the client on remote machines (relative to the server process' machine).

> There is a lot more stored in a token. We'll examine more details in subsequent sections.

The *token.exe* application demonstrates more token information obtained with calls to `GetTokenInformation`.

Some information inside the token can be changed as well. One generic way is to use `SetTokenInformation` that uses the same `TOKEN_INFORMATION_CLASS` enumeration as `GetTokenInformation` does:

```
BOOL SetTokenInformation(
    _In_ HANDLE TokenHandle,
    _In_ TOKEN_INFORMATION_CLASS TokenInformationClass,
    _In_ LPVOID TokenInformation,
    _In_ DWORD TokenInformationLength);
```

Unfortunately, the documentation does not state which information classes are valid, as only a subset is. Some changes to the token are possible with different APIs. Table 16-1 describes the valid information classes for `SetTokenInformation` and the privileges and access mask they require (if any).

Table 16-1: Valid values for `SetTokenInformation`

TOKEN_INFORMATION_CLASS	Access mask required	Privilege required
TokenOwner	TOKEN_ADJUST_DEFAULT	
TokenPrimaryGroup	TOKEN_ADJUST_DEFAULT	
TokenDefaultDacl	TOKEN_ADJUST_DEFAULT	
TokenSessionId	TOKEN_ADJUST_SESSIONID	SeTcbPrivilege
TokenVirtualizationAllowed		SeCreateTokenPrivilege
TokenVirtualizationEnabled	TOKEN_ADJUST_DEFAULT	
TokenOrigin		SeTcbPrivilege
TokenMandatoryPolicy		SeCreateTokenPrivilege

As an example, here is the code necessary to change the *UAC virtualization* state of a process, by manipulating its token. UAC virtualization is discussed in the section "User Access Control" later in this chapter. For now, we'll focus on the mechanics of `SetTokenInformation`. For each information class, the documentation states the associated structure that should be provided. In this case, for `TokenVirtualizationEnabled`, the value is stored in a simple `ULONG`, where 1 is to enable and 0 to disable. Here is the code (error handling omitted):

```
// hProcess is an open process handle with PROCESS_QUERY_INFORMATION
HANDLE hToken;
::OpenProcessToken(hProcess, TOKEN_ADJUST_DEFAULT, &hToken);

ULONG enable = 1;   // enable
::SetTokenInformation(hToken, TokenVirtualizationEnabled,
    &enable, sizeof(enable));
```

According to table 16-1, `TokenVirtualizationEnabled` requires the `TOKEN_ADJUST_DEFAULT` access mask. The *setvirt.exe* file is a command-line application (from this chapter's samples) that allows enabling or disabling UAC virtualization of a process (which is stored in its primary token, of course). Here is an example run to enable UAC virtualization for process 6912:

```
c:\>setvirt 6912 on
```

You can view the UAC virtualization state with *Task Manager* (add the *UAC virtualization* column, figure 16-8) or in *Process Explorer*'s security tab (figure 16-9).

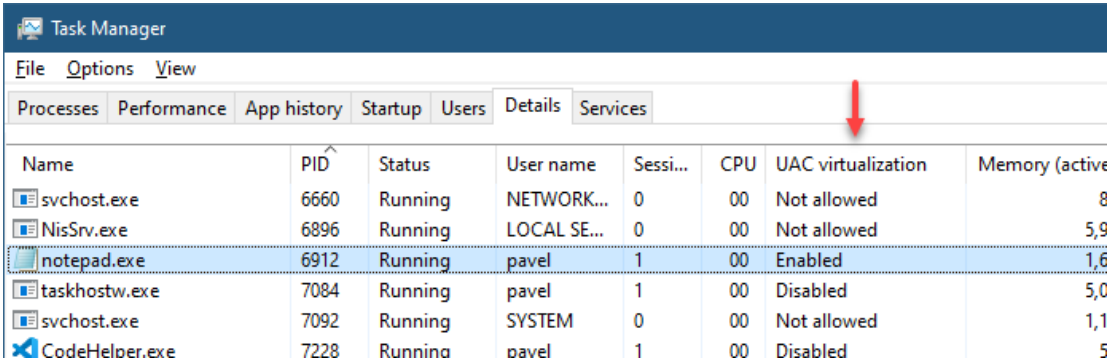


Figure 16-8: *UAC Virtualization* in *Task Manager*

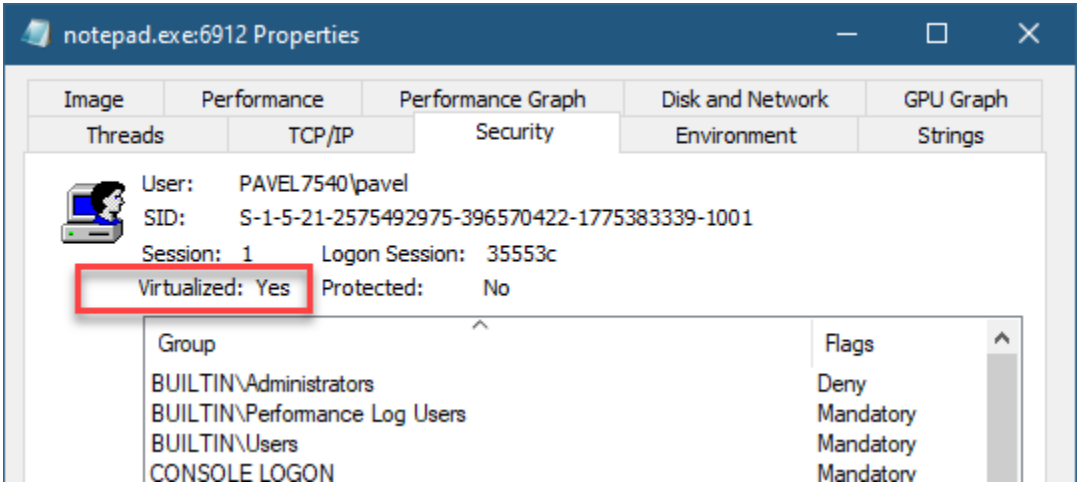


Figure 16-9: *UAC Virtualization* in *Process Explorer*

The Secondary Logon Service

The *Secondary Logon* service (*seclogon*) is a built-in service that allows launching a process under a different user than the caller. It's the service used with the *runas.exe* built-in command-line tool. Invoking the service is done by calling `CreateProcessWithLogonW`:

```
BOOL CreateProcessWithLogonW(
    _In_          LPCWSTR lpUsername,
    _In_opt_      LPCWSTR lpDomain,
    _In_          LPCWSTR lpPassword,
    _In_          DWORD dwLogonFlags,
    _In_opt_      LPCWSTR lpApplicationName,
    _Inout_opt_ LPWSTR lpCommandLine,
    _In_          DWORD dwCreationFlags,
    _In_opt_      LPVOID lpEnvironment,
```

```
    _In_opt_    LPCWSTR lpCurrentDirectory,
    _In_        LPSTARTUPINFOW lpStartupInfo,
    _Out_       LPPROCESS_INFORMATION lpProcessInformation);
```

Notice there is no ANSI version for `CreateProcessWithLogonW`.

Some of the parameters of `CreateProcessWithLogonW` should seem familiar, as they are normally provided to a standard `CreateProcess` call. In actuality, `CreateProcessWithLogonW` is a combination of two separate calls: `LogonUser` and `CreateProcessAsUser`. `LogonUser` allows retrieving a token for a user given proper credentials:

```
BOOL LogonUser(
    _In_     LPCTSTR lpszUsername,
    _In_opt_ LPCTSTR lpszDomain,
    _In_opt_ LPCTSTR lpszPassword,
    _In_     DWORD dwLogonType,
    _In_     DWORD dwLogonProvider,
    _Outptr_ PHANDLE phToken);
```

`lpszUsername` is the username, which can be either a "normal" name or a *User Principal Name* (UPN) - something like "user@domain.com". If the username is a UPN name, `lpszDomain` must be `NULL`. Otherwise, `lpszDomain` should be a domain name, where "." is valid as the local machine (for local logins). `lpszPassword` password is the cleartext password of the user.

`dwLogonType` is the logon type. Here are the common values (check the documentation for the full list):

- `LOGON32_LOGON_INTERACTIVE` - suitable for users that will interactively be using the machine. It caches logon information for disconnected operations, meaning it has higher overhead than other logon types.
- `LOGON32_LOGON_BATCH` - suitable for performing operations on behalf of a user without his/her intervention. This logon type does not cache credentials.
- `LOGON32_LOGON_NETWORK` - similar to batch, but the returned token is an impersonation token rather than a primary token. Such a token cannot be used with

`CreateProcessAsUser`, but can be converted to a primary token with `DuplicateTokenEx` (see the section "impersonation"). This is the fastest logon type.

To succeed, the specific logon type must have been granted to the account logging into.
`dwLogonProvider` selects the logon provider, either `LOGON32_PROVIDER_WINNT50` (kerberos, also called "negotiate"), or `LOGON32_PROVIDER_WINNT40` (NTLM - *NT Lan manager*). Specifying `LOGON32_PROVIDER_DEFAULT` selects NTLM.

`phToken` returns the token handle if the function succeeds. With a token in hand,
`CreateProcessAsUser` is just a function call away:

```
BOOL CreateProcessAsUser(
    _In_opt_ HANDLE hToken,
    _In_opt_ LPCTSTR lpApplicationName,
    _Inout_opt_ LPTSTR lpCommandLine,
    _In_opt_ LPSECURITY_ATTRIBUTES lpProcessAttributes,
    _In_opt_ LPSECURITY_ATTRIBUTES lpThreadAttributes,
    _In_ BOOL bInheritHandles,
    _In_ DWORD dwCreationFlags,
    _In_opt_ LPVOID lpEnvironment,
    _In_opt_ LPCTSTR lpCurrentDirectory,
    _In_ LPSTARTUPINFO lpStartupInfo,
    _Out_ LPPROCESS_INFORMATION lpProcessInformation);
```

The function looks identical to `CreateProcess` except the extra first parameter: a primary token under which the new process should execute. Although it seems simple enough, there is at least one snag: calling `CreateProcessAsUser` requires the `SeAssignPrimaryTokenPrivilege` privilege. This privilege is normally granted to the service-running accounts (*Local Service*, *Network Service* and *Local System*), but not to standard users or even the local administrators alias. This means calling `CreateProcessAsUser` is feasible when called from a service (see chapter 19), but is problematic otherwise.

This is where the Secondary Logon service comes in. Since it runs under the *Local System* account, it can call `CreateProcessAsUser` without any issue. `CreateProcessWithLogonW` is its driver.

In reality, calling `CreateProcessAsUser` is not *that* simple because some security settings needs to be configured for the process to launch successfully.

The `dwLogonFlags` parameter to `CreateProcessWithLogonW` can be one of the following:

- `LOGON_WITH_PROFILE` - causes `CreateProcessWithLogonW` to load the user's profile. This is important if the new process needs access to the `HKEY_CURRENT_USER` registry key.
- Zero (0) - do not load the user's profile.
- `LOGON_NETCREDENTIALS_ONLY` - the new process will execute under the caller's user (not the specified user), but a new network logon session with the specified user is created. This means any network access by the new process will use the other user's token.

The next two parameters, `lpApplicationName` and `lpCommandLine` serve the same purpose as they do in `CreateProcess` (refer to chapter 3 for a refresher if needed). `dwCreationFlags` is a combination of flags that are passed internally to `CreateProcessAsUser` - most flags that are valid in `CreateProcess` are valid here as well. `lpEnvironment` has the same meaning as it does to `CreateProcess`, except the default environment is the other user's default rather than the caller's.

Finally, `lpStartupInfo` and `lpProcessInfo` have the same meaning as they do in `CreateProcess`.

> You may be thinking that creating a process under another user can be accomplished by calling `LogonUser`, then impersonating the new user (`ImpersonateLoggedOnUser`), and finally just call `CreateProcess` normally. This fails, however, because `CreateProcess` always uses the caller's primary (process) token rather the active impersonation token (if any).

There is yet another function that invokes the secondary logon service: `CreateProcessWithTokenW`:

```
BOOL CreateProcessWithTokenW(
    _In_        HANDLE hToken,
    _In_        DWORD dwLogonFlags,
    _In_opt_    LPCWSTR lpApplicationName,
    _Inout_opt_ LPWSTR lpCommandLine,
    _In_        DWORD dwCreationFlags,
    _In_opt_    LPVOID lpEnvironment,
    _In_opt_    LPCWSTR lpCurrentDirectory,
    _In_        LPSTARTUPINFOW lpStartupInfo,
    _Out_       LPPROCESS_INFORMATION lpProcessInformation);
```

This function requires the `SeImpersonatePrivilege`, normally granted to the local administrators alias and the service accounts. It uses an existing token (for example, from a successful `LogonUser` call), but it does require some extra work so the new process does not fail initialization.

Using `CreateProcessWithLogonW` is by far the easiest way to go, with some loss of flexibility compared to `LogonUser` and `CreateProcessAsUser`.

Impersonation

Normally any operation performed by a thread is done with the process' token. What if a thread in the process wants to make temporary changes to the token before performing some operation? Any change to the token would be reflected on the process level, affecting all threads in the process. The thread might want its own private token.

Using something like `DuplicateHandle` will not work as perhaps expected, because the new handle still refers to the same object. Instead, the `DuplicateTokenEx` function can be used:

```
BOOL DuplicateTokenEx(
    _In_ HANDLE hExistingToken,
    _In_ DWORD dwDesiredAccess,
    _In_opt_ LPSECURITY_ATTRIBUTES lpTokenAttributes,
    _In_ SECURITY_IMPERSONATION_LEVEL ImpersonationLevel,
    _In_ TOKEN_TYPE TokenType,
    _Outptr_ PHANDLE phNewToken);
```

`DuplicateTokenEx` is unique in the sense of being the only function that can duplicate an object. There are no such functions for mutexes or semaphores, for example.

`hExistingToken` is the existing token to duplicate. It must have the `TOKEN_DUPLICATE` access mask. `dwDesiredAccess` is the requested access mask for the new token. Specifying zero results in the same access mask as the original token. `TOKEN_IMPERSONATE` is needed if this new token is going to be used for impersonation.

`lpTokenAttributes` is the standard `SECURITY_ATTRIBUTES` (discussed in the section "Security Descriptors"), usually set to `NULL`. `ImpersonationLevel` indicates the impersonation information level inherent in the new token (see later in this section for more on this). `TokenType` is `TokenPrimary` (for attaching to a process) or `TokenImpersonation` for attaching to a thread. Finally, `phNewToken` receives the new token handle if the call is successful.

The new token can now be manipulated (enabling/disabling privileges, changing UAC virtualization state, etc.), without affecting the original token. To make the new token work, it needs to be attached to the current thread (assuming it was duplicated as an impersonation token) by calling `SetThreadToken`:

```
BOOL SetThreadToken(
    _In_opt_ PHANDLE Thread,
    _In_opt_ HANDLE Token);
```

`Thread` is a pointer to a thread's handle, where `NULL` means the current thread.

This is a rather unusual way to specify a thread handle; usually a direct handle is used where `GetCurrentThread` is used to indicate the current thread.

`Token` is an impersonation token to use. `NULL` is acceptable as a way to stop using any existing token. Alternatively, the `RevertToSelf` function can be called instead, if the impersonation is on the current thread:

```
BOOL RevertToSelf();
```

The following example duplicates the process token for impersonation, and makes a "local" change to the impersonation token (error handling omitted):

```
HANDLE hProcToken;
::OpenProcessToken(GetCurrentProcess(), TOKEN_DUPLICATE,
    &hProcToken);

HANDLE hImpToken;
::DuplicateTokenEx(hProcToken, MAXIMUM_ALLOWED, nullptr,
    SecurityIdentification, TokenImpersonation, &hImpToken);
::CloseHandle(hProcToken);
// enable UAC virtualization on the new token
ULONG virt = 1;
::SetTokenInformation(hImpToken, TokenVirtualizationEnabled,
    &virt, sizeof(virt));
// impersonate
::SetThreadToken(nullptr, hImpToken);
// do work...

::RevertToSelf();
::CloseHandle(hImpToken);
```

This whole procedure of taking the current process token and duplicating as an impersonation token and attaching it to the current thread can be achieved with one stroke:

```
BOOL ImpersonateSelf(_In_ SECURITY_IMPERSONATION_LEVEL Level);
```

`ImpersonateSelf` duplicates the process token to create an impersonation token and then calls `SetThreadToken`.

Another shorthand function can be used when impersonating with some token (not necessarily the process token) on the current thread:

```
BOOL ImpersonateLoggedOnUser(_In_ HANDLE hToken);
```

`ImpersonateLoggedOnUser` accepts a primary or impersonation token, duplicates the token (if needed) and calls `SetThreadToken` on the current thread.

In the following example, `ImpersonateLoggedOnUser` is used after `LogonUser`:

```
HANDLE hToken;
::LogonUser(L"alice", L".", L"alicesecretpassword",
    LOGON32_LOGON_BATCH, LOGON32_PROVIDER_DEFAULT, &hToken);
// impersonate alice
::ImpersonateLoggedOnUser(hToken);

// do work as alice...

::RevertToSelf();
::CloseHandle(hToken);
```

Impersonation in Client/Server

The classic usage of impersonation is in client/server scenarios. Imagine there is a server process running under user A. Multiple clients connect to the server process using some communication mechanism (COM, named pipes, RPC, and some others), asking the server process to perform some operation on their behalf (figure 16-10).

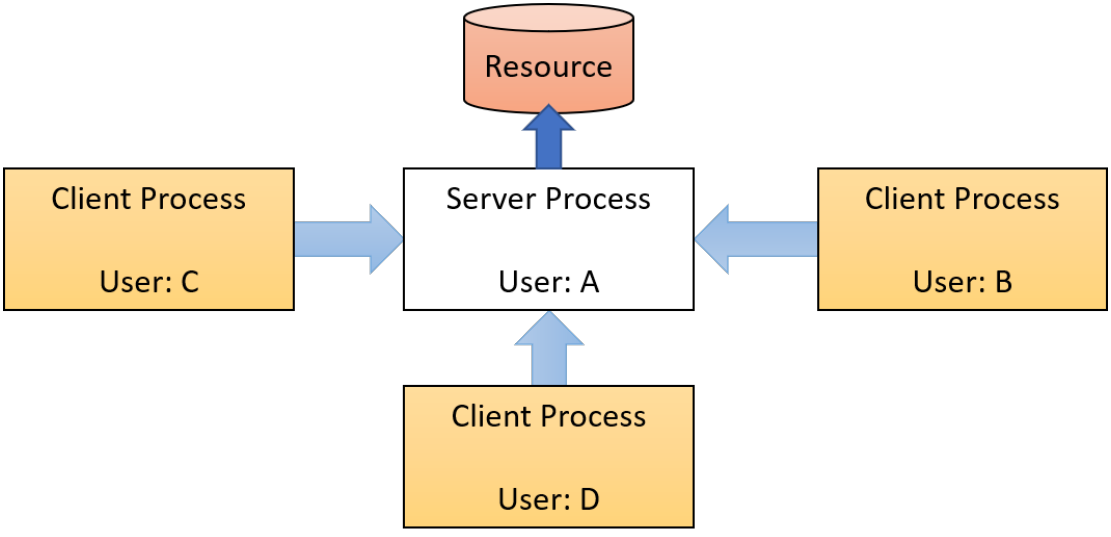


Figure 16-10: Client/server

If the server process performs the requested operation using its own identity (A), that wouldn't be right. Perhaps client B asked to perform an operation on a file B has no access to, but A does. Instead, the server should impersonate the requesting client before attempting to perform the operation.

When a token is duplicated, the `SECURITY_IMPERSONATION_LEVEL` impersonation level indicates what "power" is inherent in the resulting token when it's sent to a server on another machine:

```
typedef enum _SECURITY_IMPERSONATION_LEVEL {
    SecurityAnonymous,
    SecurityIdentification,
    SecurityImpersonation,
    SecurityDelegation
} SECURITY_IMPERSONATION_LEVEL;
```

With `SecurityAnonymous`, the server has no idea who is the client, and so cannot impersonate it. This could be fine for some types of operations the server is asked to perform. `SecurityIdentification` is the next level where the server can query properties of the client, but still cannot impersonate it (unless the server process is on the same machine as the client process). With `SecurityImpersonation`, the server can impersonate the client on the server's machine only (and no further). The last (and most permissive) impersonation level, `SecurityDelegation`, allows the server to call another server on another machine and propagate the token so that the other server can impersonate the original client, and this can go on for any number of hops.

The token propagation mechanism depends on the communication mechanism used. Table 16-2 shows the impersonation and reverting APIs for some of these mechanisms. Check the documentation for more details (named pipes are discussed in chapter 18 and COM is discussed in chapter 21).

Table 16-2: APIs for remote client impersonation

Communication mechanism	Impersonation	Reverting
Named pipes	`ImpersonateNamedPipeClient`	`RevertToSelf`
RPC	`RpcImpersonateClient`	`RpcRevertToSelf`
COM	`CoImpersonateClient`	`CoRevertToSelf`

Privileges

A *privilege* is the right (or denied right) to perform some system-level operation, that is not tied to a particular object. Example privileges include: loading device drivers, debug processes from other users, take ownership of an object, and more. A full list can be viewed in the *Local Security Policy* snapin (figure 16-11). For each privilege ("policy" column), the tool lists the accounts that are granted that privilege.

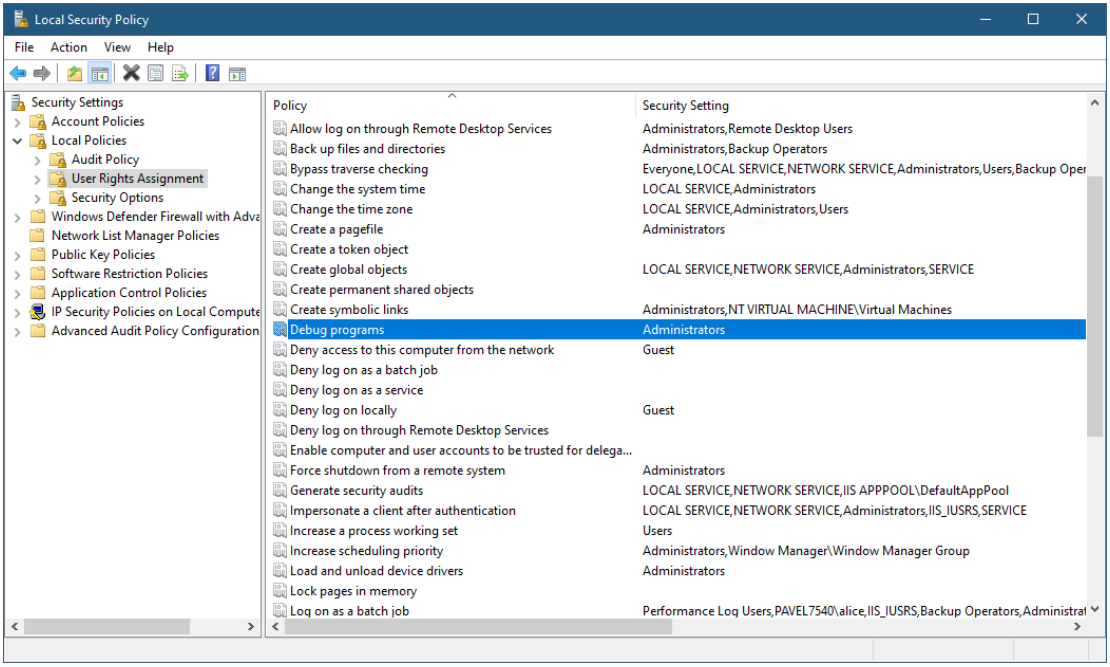


Figure 16-11: Privileges in the *Local Security Policy* editor

Technically, figure 16-11 shows both *privileges* and *user rights*. The distinction is the following: user rights apply to accounts - that is, the data stored in the users' database. User rights are always about allowing or denying some form of login. Privileges, on the other hand, (also stored as static data in the account database), only apply after the user has logged in. Privileges are stored in the user's access token, while user rights are not, since they have meaning only before the user logs in.

> Examples of user rights include: "Deny log on as a batch job", "Allow log on locally" and "Allow log on through Remote Desktop Services".

Once a token is created (or duplicated), new privileges cannot be added to the token. An administrator could add privileges to the account (database) itself, but that has no effect on existing tokens. Once the user logs off and logs on again, the new privileges will be available in its token.

Most privileges are disabled by default. This prevents accidental (unintended) usage of those privileges. Figure 16-12 shows the list of privileges (shown in *Process Explorer*) for an *Explorer.exe* process. The only enabled privilege (and this one is enabled by default) is *SeChangeNotifyPrivilege* - the rest are disabled.

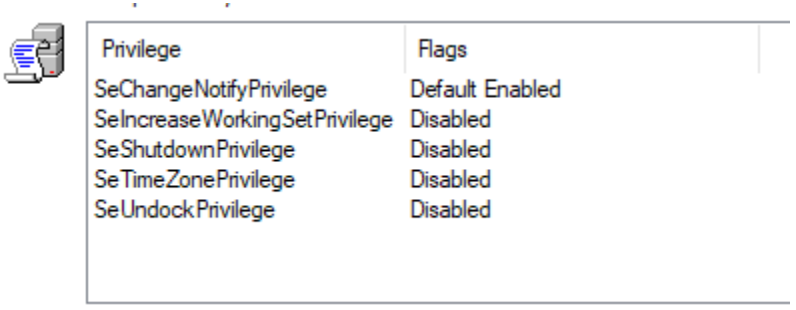


Figure 16-12: Privileges in *Explorer*'s token

> The curiously named *SeChangeNotifyPrivilege* privilege is granted and enabled by default for all users. Its descriptive name is "Bypass traverse checking". It allows access to a file in a directory where some of the parent directories are themselves inaccessible to that user. For example, the file *c:\A\B\c.txt* is accessible (if its security descriptor allows it) for the user even if the directory *A* is not. Traversing the security descriptors of all parent directories is costly, which is why this privilege is enabled by default.

Getting the list of privileges in a token is possible with `GetTokenInformation` we met already. To enable, disable, or remove a privilege, a call to `AdjustTokenPrivileges` is required:

```
BOOL AdjustTokenPrivileges(
    _In_ HANDLE TokenHandle,
    _In_ BOOL DisableAllPrivileges,
    _In_opt_ PTOKEN_PRIVILEGES NewState,
    _In_ DWORD BufferLength,
    _Out_opt_ PTOKEN_PRIVILEGES PreviousState,
    _Out_opt_ PDWORD ReturnLength);
```

`TokenHandle` must have the `TOKEN_ADJUST_PRIVILEGES` access mask for the call to have a chance at success. If `DisableAllPrivileges` is `TRUE`, the function disables all privileges in the token, and the next two parameters are ignored. The privilege(s) to change are provided by a `TOKEN_PRIVILEGES` structure defined like so:

```
typedef struct _LUID_AND_ATTRIBUTES {
    LUID Luid;
    DWORD Attributes;
} LUID_AND_ATTRIBUTES;

typedef struct _TOKEN_PRIVILEGES {
    DWORD PrivilegeCount;
    LUID_AND_ATTRIBUTES Privileges[ANYSIZE_ARRAY];
} TOKEN_PRIVILEGES, *PTOKEN_PRIVILEGES;
```

Privileges are represented in the Windows API as strings. For example, `SE_DEBUG_NAME` is defined as `TEXT("SeDebugPrivilege")`. However, each privilege is also given an `LUID` when the system starts, which could be different every time the system restarts.
`AdjustTokenPrivileges` requires the privileges to manipulate as `LUID`s, rather than strings. So we have to make a little effort to get the `LUID` of a privilege with `LookupPrivilegeValue`:

```
BOOL LookupPrivilegeValue(
    _In_opt_ LPCTSTR lpSystemName,
    _In_     LPCTSTR lpName,
    _Out_    PLUID   lpLuid);
```

The function accepts a machine name (`NULL` is fine for the local machine), the name of a privilege, and returns its `LUID`.

Back to `AdjustTokenPrivileges` - `TOKEN_PRIVILEGES` requires an array of `LUID_AND_ATTRIBUTES` structures, each of which contains an `LUID` and the attributes to use. Possible values are `SE_PRIVILEGE_ENABLED` to enable the privilege, zero to disable it, and `SE_PRIVILEGE_REMOVED` to remove it.

`NewState` is a pointer to `TOKEN_PRIVILEGES` and `BufferLength` is the size of the data, since multiple privileges can be modified at the same time. Finally, `PreviousState` and `ReturnedLength` are optional parameters that can return the previous state of the modified privileges. Most callers just specify `NULL` for both parameters.

The return value of `AdjustTokenPrivileges` is somewhat tricky. It returns `TRUE` on any kind of success, even if only some of the privileges have been changed successfully. The correct thing to do (if the call returns `TRUE`) is call `GetLastError`. If zero is returned, all went well, otherwise `ERROR_NOT_ALL_ASSIGNED` may be returned which indicates something went wrong. If only one privilege was requested, this really indicates a failure.

We already used `AdjustTokenPrivileges` a number of times, in chapter 13 and several chapters in part 1, without a full explanation. Now we can write a generic function to enable or disable any privilege in the caller's token by leveraging `AdjustTokenPrivileges` and `LookupPrivilegeValue`:

```
bool EnablePrivilege(PCWSTR privName, bool enable) {
    HANDLE hToken;
    if (!::OpenProcessToken(::GetCurrentProcess(),
        TOKEN_ADJUST_PRIVILEGES, &hToken))
        return false;

    bool result = false;
    TOKEN_PRIVILEGES tp;
    tp.PrivilegeCount = 1;
    tp.Privileges[0].Attributes = enable ? SE_PRIVILEGE_ENABLED : 0;
    if (::LookupPrivilegeValue(nullptr, privName,
        &tp.Privileges[0].Luid)) {
        if (::AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(tp),
            nullptr, nullptr))
            result = ::GetLastError() == ERROR_SUCCESS;
    }
    ::CloseHandle(hToken);
    return result;
}
```

Using a single privilege makes the call to `AdjustTokenPrivileges` easy, since there is room for exactly one privilege in `TOKEN_PRIVILEGES` without the need for extra allocations.

If an application needs to check if a token contains one or more privileges, it can call `PrivilegeCheck`:

```
BOOL PrivilegeCheck(
    _In_ HANDLE ClientToken,
    _Inout_ PPRIVILEGE_SET RequiredPrivileges,
    _Out_ LPBOOL pfResult);
```

The function accepts the token to check and a pointer to a `PRIVILEGE_SET` structure:

```
typedef struct _PRIVILEGE_SET {
    DWORD PrivilegeCount;
    DWORD Control;
    LUID_AND_ATTRIBUTES Privilege[ANYSIZE_ARRAY];
} PRIVILEGE_SET, *PPRIVILEGE_SET;
```

`PrivilegeCount` is the number of privileges to check, listed in the `Privileges` array. Note the array has just a single item by default (`ANYSIZE_ARRAY`), so if more than one privilege is required, the structure should be allocated dynamically as needed. `Control` has currently a single flag defined, `PRIVILEGE_SET_ALL_NECESSARY`, which if set, indicates all the listed privileges must be held by the client for a

TRUE return value in pfResult. Otherwise, *pfResult* is TRUE if at least one privilege in the array is enabled.

Each privilege in the array consists of the privilege LUID. The attributes accompanying LUID_AND_ATTRIBUTES should be zeroed on input. When the function returns, if it returns TRUE for success, then *pfResult indicates if the privilege(s) are enabled. An enabled privilege has its attribute set to the SE_PRIVILEGE_USED_FOR_ACCESS (0x8000000). Otherwise, it's zero, meaning the privilege is disabled.

if the function as a whole returns FALSE, it means either some error in arguments, or the privilege(s) checked don't exit in the token at all. Here is a function that checked if a specified privilege is enabled:

```
bool IsPrivilegeEnabled(HANDLE hToken, PCWSTR name) {
    PRIVILEGE_SET set{};
    set.PrivilegeCount = 1;

    if (!::LookupPrivilegeValue(nullptr, name, &set.Privilege[0].Luid))
        return false;

    BOOL result;
    return ::PrivilegeCheck(hToken, &set, &result) && result;
}
```

Super Privileges

An exhaustive discussion of all available privileges is beyond the scope of this book. Some privileges, however, do bear special mention. There is a set of privileges that are so powerful that having any of them allows almost complete control of the system (some allow even complete control). These are sometimes affectionally called "Super Privileges", although this is not an official term.

Take Ownership

An object's owner can always set who-can-do-what with the object (see the "Security Descriptors" section for more details). The *SeTakeOwnershipPrivilege* (SE_TAKE_OWNERSHIP_NAME) allows its yielder to set itself as an owner of any kernel object (file, mutex, process, etc.). Being an owner, the user can now give himself/herself full access to the object.

This privilege is normally given to administrators, which makes sense, as an administrator should be able to take control of any object if needed. For example, suppose an employee leaves a company, and some files/folders he/she owned are now inaccessible. An administrator can exercise the take ownership privilege and set the administrator as the owner, allowing full access to the object to be set.

One way to see this in action is to use the security descriptor of an object, such as a file (figure 16-13). Clicking the *Advanced* button shows the dialog in figure 16-14, where the owner is shown.

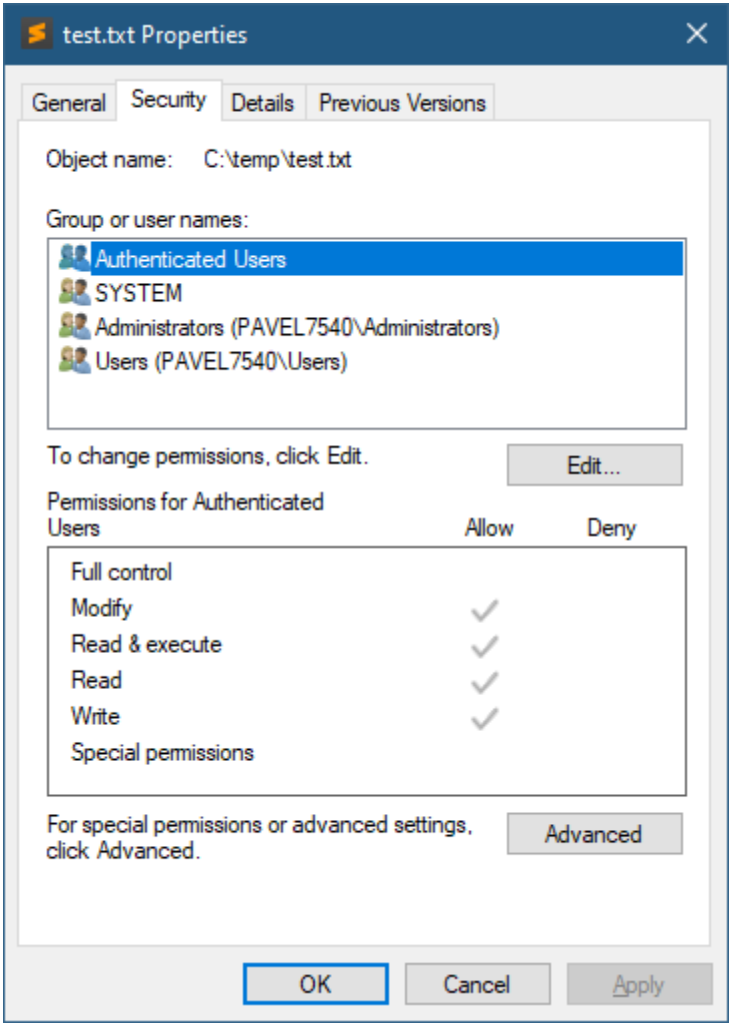


Figure 16-13: Security settings of a kernel object

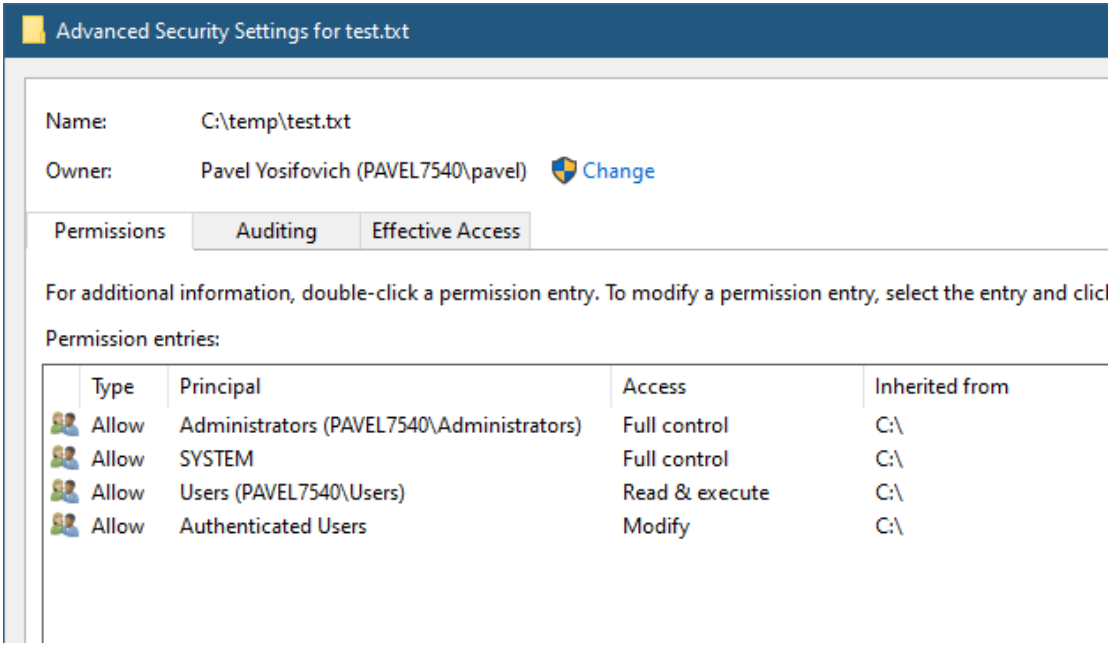


Figure 16-14: Advanced security settings dialog

Clicking the *Change* button causes the dialog handler to enable the take ownership privilege (if exists in the caller's token, in this case the user running *Explorer*), and replace the owner.

Backup

The backup privilege (`SE_BACKUP_NAME`) gives its user read access to any object, regardless of its security descriptor. Normally, this should be given to applications that perform some form of backup. Administrators and the *Backup operators* group have this privilege by default. To use it for files, specify `FILE_BACKUP_SEMANTICS` when opening a file with `CreateFile`.

Restore

The restore privilege (`SE_RESTORE_NAME`) is the opposite of backup, and provides write access to any kernel object.

Debug

The debug privilege (`SE_DEBUG_NAME`) allows debugging and memory manipulation of any process. This includes calling `CreateRemoteThread` to inject a thread to any process. This excludes protected and PPL processes.

TCB

The TCB privilege (*Trusted Computing Base*, `SE_TCB_NAME`), described as "Act part of the operating system", is one of the most powerful privileges. A testament to that is the fact that by default it's not given

to any user or group. Having this privilege allows a user to impersonate any other user, and generally have the same access the kernel has.

Create Token

The create token privilege (`SE_CREATE_TOKEN_NAME`) allows creating a token, thus populating it with any privilege or group. This privilege is not given to any user by default. However, the *Lsass* process must have it as it's required after a successful logon. If you examine *Lsass* security properties in *Process Explorer*, you'll find it has this privilege. How can this be if the privilege is given to no user? We'll answer that question in chapter 19.

Access Masks

We've encountered access masks many times before. At first, they might appear to use random values for the various access bits, but there is a logical grouping of bits, depicted in figure 16-15.

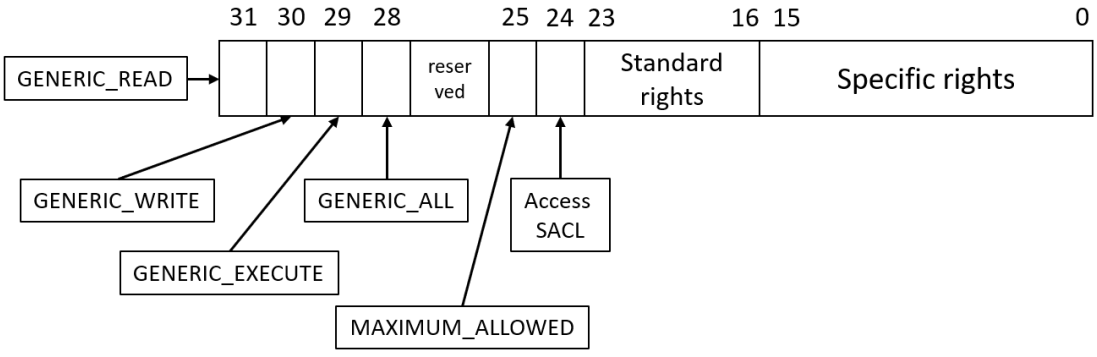


Figure 16-15: Components of an access mask

The *specific rights* part (low 16 bits) represents just that: specific rights that are different for each object type. For example, `PROCESS_TERMNNATE` and `PROCESS_CREATE_THREAD` are two specific rights for process objects.
Next, there are standard rights, appropriate for many types of objects. For example,
`SYNCHRONIZE`, `DELETE` and `WRITE_DAC` are examples of standard rights. They don't necessarily have meaning for all object types. For example, `SYNCHRONIZE` only has meaning for dispatcher (waitable) objects.

The next bit (24) is the `ACCESS_SYSTEM_SECURITY` access right, that allows access to the *System Access Control List* (discussed in the next section). `MAXIMUM_ALLOWED` (bit 25) is a special value, that if used, provides the maximum access mask the client can obtain. For example:

```
HANDLE hProcess = ::OpenProcess(MAXIMUM_ALLOWED, FALSE, pid);
```

If a handle can be obtained, the resulting access mask is the highest possible for the caller. The above example is not that useful in practice since the caller knows which access mask is needed to get the job done.

Bits 28-31 represent generic rights. These rights (if used) must be translated, or mapped, to specific access rights. For example, specifying `GENERIC_WRITE` must be mapped to what "write" means for the object type in question. This is performed internally with the following structure:

```
typedef struct _GENERIC_MAPPING {
    ACCESS_MASK GenericRead;
    ACCESS_MASK GenericWrite;
    ACCESS_MASK GenericExecute;
    ACCESS_MASK GenericAll;
} GENERIC_MAPPING;
```

The default mappings can be viewed with the *System Explorer* tool (figure 16-13), but they are fairly intuitive, and some are indirectly defined in the headers. For example, `FILE_GENERIC_READ` is the `GENERIC_READ` mapping for files.

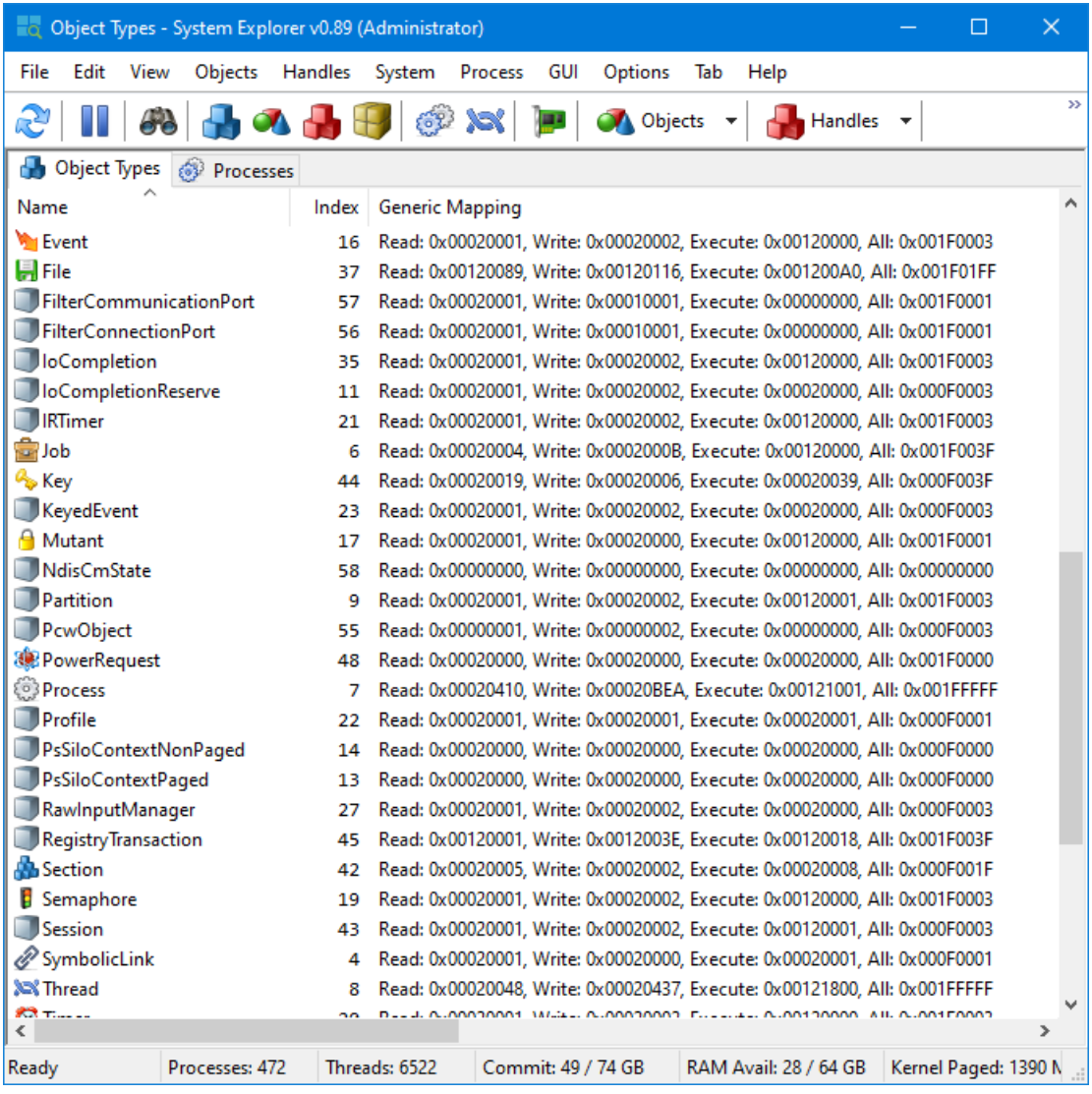


Figure 16-16: Generic mappings in *System Explorer*

Security Descriptors

A *Security Descriptor* is a variable-length structure that includes information on *who-can-do-what* with the object it's attached to. A security descriptor contains these pieces of information:

- Owner SID - the owner of an object.
- Primary Group SID - used in the past for group security in POSIX subsystem applications.
- *Discretionary Access Control List* (DACL) - a list of *Access Control Entries* (ACE), specifying who-can-do-what with the object.

- *System Access Control List* (SACL) - a list of ACEs, indicating which operations should cause an audit entry to be written to the security log.

The owner of the object always has the `WRITE_DAC` (and `READ_CONTROL`) standard access rights, meaning it can read and change the object's DACL. This is important, otherwise a careless call can make the object completely inaccessible. Having `WRITE_DAC` for the owner ensures that the owner can change the DACL no matter what.

Getting the security descriptor of any kernel object for which you have an open handle can be done with `GetKernelObjectSecurity`:

```
BOOL GetKernelObjectSecurity(
    _In_ HANDLE Handle,
    _In_ SECURITY_INFORMATION RequestedInformation,
    _Out_ PSECURITY_DESCRIPTOR pSecurityDescriptor,
    _In_ DWORD nLength,
    _Out_ LPDWORD lpnLengthNeeded);
```

The handle must have the `READ_CONTROL` standard access mask. `SECURITY_INFORMATION` is an enumeration specifying what kind of information to return in the resulting security descriptor (more than one can be specified with the OR operator). The most common ones are `OWNER_SECURITY_INFORMATION` and `DACL_SECURITY_INFORMATION`. The result is stored in `PSECURITY_DESCRIPTOR`. `SECURITY_DESCRIPTOR` structure is defined, but should be treated opaquely, and this is why `PSECURITY_DESCRIPTOR` (pointer to that structure) is `typedefed` as `PVOID`. `GetKernelObjectSecurity` requires the caller to allocate a large-enough buffer, specify its length in the `nLength` parameter and get back the actual length in `lpnLengthNeeded`.

Requesting the SACL with `SACL_SECURITY_INFORMATION` requires the *SeSecurityPrivilege*, normally given to administrators.

With the `PSECURITY_DESCRIPTOR` in hand, several functions exist to extract the data stored in it:

```
DWORD GetSecurityDescriptorLength(_In_ PSECURITY_DESCRIPTOR pSD);
BOOL GetSecurityDescriptorControl(      // control flags
    _In_ PSECURITY_DESCRIPTOR pSecurityDescriptor,
    _Out_ PSECURITY_DESCRIPTOR_CONTROL pControl,
    _Out_ LPDWORD lpdwRevision);
BOOL GetSecurityDescriptorOwner(        // owner
    _In_ PSECURITY_DESCRIPTOR pSecurityDescriptor,
    _Outptr_ PSID* pOwner,
    _Out_ LPBOOL lpbOwnerDefaulted);
BOOL GetSecurityDescriptorGroup(     // primary group (mostly useless)
    _In_ PSECURITY_DESCRIPTOR pSecurityDescriptor,
```

```
    _Outptr_ PSID* pGroup,
    _Out_ LPBOOL lpbGroupDefaulted);
BOOL GetSecurityDescriptorDacl(         // DACL
    _In_ PSECURITY_DESCRIPTOR pSecurityDescriptor,
    _Out_ LPBOOL lpbDaclPresent,
    _Outptr_ PACL* pDacl,
    _Out_ LPBOOL lpbDaclDefaulted);
BOOL GetSecurityDescriptorSacl(         // SACL
    _In_ PSECURITY_DESCRIPTOR pSecurityDescriptor,
    _Out_ LPBOOL lpbSaclPresent,
    _Outptr_ PACL* pSacl,
    _Out_ LPBOOL lpbSaclDefaulted);
```

For example, here is some code that shows the owner of a process given its ID:

```
bool DisplayProcessOwner(DWORD pid) {
    HANDLE hProcess = ::OpenProcess(READ_CONTROL, FALSE, pid);
    if (!hProcess)
        return false;

    BYTE buffer[1 << 10];
    auto sd = (PSECURITY_DESCRIPTOR)buffer;
    DWORD len;
    BOOL success = ::GetKernelObjectSecurity(hProcess,
        OWNER_SECURITY_INFORMATION,
        sd, sizeof(buffer), &len);
    ::CloseHandle(hProcess);

    if(!success)
        return false;

    PSID owner;
    BOOL isDefault;
    if (!::GetSecurityDescriptorOwner(sd, &owner, &isDefault))
        return false;

    printf("Owner: %ws (%ws)\n", GetUserNameFromSid(owner).c_str(),
        SidToString(owner).c_str());
    return true;
}
```

Another function to retrieve an object's security descriptor is `GetNamedSecurityInfo` (#include <*AclAPI.h*>):

```
DWORD GetNamedSecurityInfo(
    _In_  LPCTSTR               pObjectName,
    _In_  SE_OBJECT_TYPE         ObjectType,
    _In_  SECURITY_INFORMATION   SecurityInfo,
    _Out_opt_       PSID         * ppsidOwner,
    _Out_opt_       PSID         * ppsidGroup,
    _Out_opt_       PACL         * ppDacl,
    _Out_opt_       PACL         * ppSacl,
    _Out_ PSECURITY_DESCRIPTOR   * ppSecurityDescriptor);
```

This function can only be used with named objects (mutexes, events, semaphores, sections) and objects that have a "path" of some sort (files and registry keys). It does not require an open handle to the object - just its name and type. The function fails if the caller cannot obtain a `READ_CONTROL` access mask, of course.

`pObjectName` is the object's name, in a format appropriate for the object type given by the `SE_OBJECT_TYPE` enumeration. The function can return the security descriptor as a whole, or just selected parts. The return value from the function is error code itself, where `ERROR_SUCCESS` (0) means all is well (there is no point in calling `GetLastError`).

The following example displays the owner of a given file:

```
bool DisplayFileOwner(PCWSTR filename) {
    PSID owner;
    DWORD error = ::GetNamedSecurityInfo(filename, SE_FILE_OBJECT,
        OWNER_SECURITY_INFORMATION, &owner,
        nullptr, nullptr, nullptr, nullptr);
    if (error != ERROR_SUCCESS)
        return false;

    printf("Owner: %ws (%ws)\n", GetUserNameFromSid(owner).c_str(),
        SidToString(owner).c_str());
    return true;
}
```

Notice that you don't free the returned information from `GetNamedSecurityInfo` - that will cause an exception to be raised.

Specifically for files, there is another function that returns the file's security descriptor: `GetFileSecurity`. The various parts then need to be retrieved with one of the aforementioned functions such as `GetSecurityDescriptorOwner`. Also, for desktop and window station objects, another convenience function exists - `GetUserObjectSecurity`.

The most important part of a security descriptor is the DACL. This directly affects who is allowed access to the object and in what way. The most well-known view of a DACL is the security property dialog box available with various tools, such as for files and directories in *Explorer* (figure 16-17).

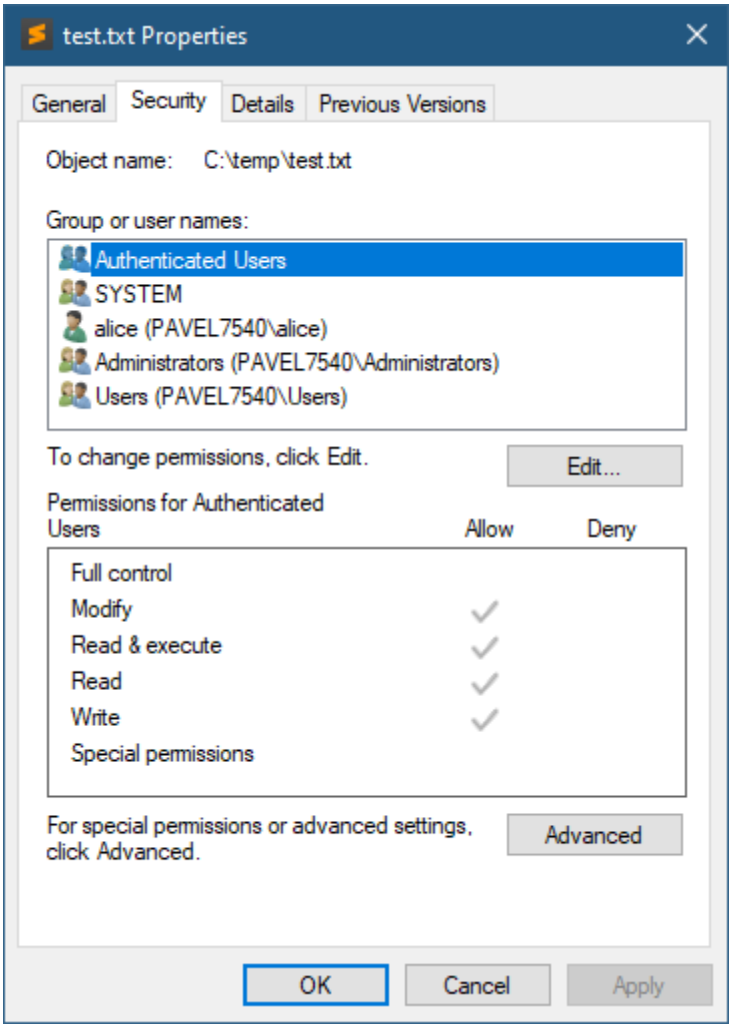


Figure 16-17: Security properties dialog box

The dialog in figure 16-17 shows the DACL. For each user or group, it shows the allowed or denied operations. Each item in the DACL is an *Access Control Entry* (ACE) that includes the following pieces of information:

- The SID to which this ACE applies (e.g. a user or a group).
- The access mask this ACE controls (e.g. `PROCESS_TERMINATE` for a process object) (could be more than one bit)
- The ACE type, most commonly *Allow* or *Deny*.

When a caller tries to gain certain access to the file, the security reference monitor in the kernel must check whether the requested access (based on the access mask) is allowed for the caller. It does so by traversing the ACEs in the DACL, looking for a definitive result. Once found, the traversal terminates. Figure 16-18 shows an example of a DACL on some file object.

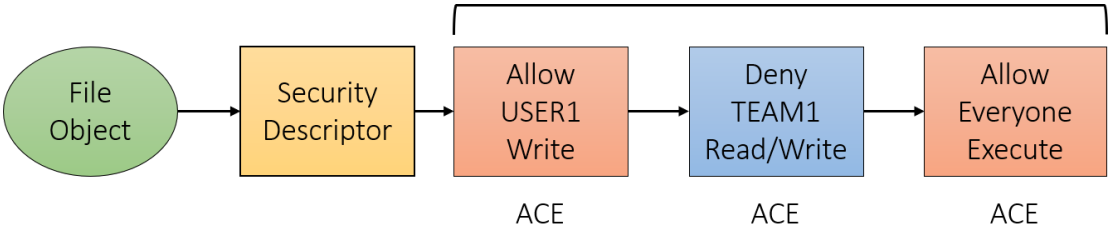


Figure 16-18: An example security descriptor

Suppose we have two users, USER1 and USER2 that are part of two groups, TEAM1 and TEAM2, and another user, USER3, that is part of group TEAM2 only. Here are some questions we can ask:

1. If USER1 wants to open the file for read access, will that succeed?
2. If USER1 wants to open the file for write access, will that succeed?
3. If USER2 wants to open the file for write access, will that succeed?
4. If USER2 wants to open the file for execute access, will that succeed?
5. If USER3 wants to open the file for read access, will that succeed?

The ACEs are traversed in order. If no definitive answer exists, the next ACE is consulted. If no definitive result is available after all ACEs are consulted, the final verdict is Access Denied.

> If the security descriptor itself is `NULL`, this means the object has no protection and all access is allowed. If the security descriptor exists, but the `DACL` itself is `NULL`, it means the same thing - no protection. If, on the other hand, the DACL is empty (i.e. no ACEs), it means no-one has access to the object (except the owner).

Let's answer the above questions:

1. USER1 is denied read access, because of the second ACE. The first ACE doesn't say anything about read access.
2. USER1 is allowed write access - the order of ACEs matter.
3. USER2 is denied write access.
4. USER2 is allowed execute access, because it's part of the Everyone group, and previous ACEs had nothing to say about execute.
5. USER3 is denied read access, because no ACE provides a definitive answer, so the final verdict is Access Denied.

There are other factors that come into play for access check, including other types of ACEs, such as inheritance ACEs. Consult the documentation for the gory details.

If you edit the DACL with the security dialog box shown by *Explorer* and other tools, the ACEs built by the security dialog box always places Deny ACEs before Allow ACEs. The example in figure 16-18 will never be constructed (but it can be constructed in this order programmatically), since ACE 2 will be placed first because it's a Deny ACE.

How would the answers to the above questions change if deny ACEs would be placed before Allow ACEs?

You can show the security properties dialog programmatically with the `EditSecurity` API. This is not easy because you need to provide an implementation for the `ISecurityInformation` COM interface that returns appropriate information for the dialog box' operation. You can find an example implementation in my *System Explorer* tool's source code and other resources online.

A DACL (and SACL for that matter) is represented by the ACL structure, a pointer to which is returned when a DACL is retrieved with `GetSecurityDescriptorDacl` or `GetNamedSecurityInfo`:

```
typedef struct _ACL {
    BYTE  AclRevision;
    BYTE  Sbz1;
    WORD  AclSize;
    WORD  AceCount;
    WORD  Sbz2;
} ACL;
typedef ACL *PACL;
```

The only interesting member is `AceCount`. The `ACL` object is followed immediately by an array of ACEs. The size of each ACE may be different (depending on the type of ACE), but each ACE always starts with an `ACE_HEADER`:

```
typedef struct _ACE_HEADER {
    BYTE  AceType;
    BYTE  AceFlags;
    WORD  AceSize;
} ACE_HEADER;
typedef ACE_HEADER *PACE_HEADER;
```

There is no need to manually calculate where each ACE starts - just call `GetAce`:

```
BOOL GetAce(
    _In_ PACL pAcl,
    _In_ DWORD dwAceIndex,
    _Outptr_ LPVOID* pAce);
```

`pAcl` is the DACL pointer, `dwAceIndex` is the ACE index (starting from zero), and the returned pointer points to the ACE itself, that always starts with `ACE_HEADER`. With the type of ACE in hand (from the header), the returned pointer from `GetAce` can be cast to the specific ACE structure. Here are the two most common ACE types: allowed and denied:

```
typedef struct _ACCESS_ALLOWED_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask;
    DWORD SidStart;
} ACCESS_ALLOWED_ACE;
typedef struct _ACCESS_DENIED_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask;
    DWORD SidStart;
} ACCESS_DENIED_ACE;
```

Here you can see the three parts of the ACE: its type, the access mask, and the SID. The SID follows the access mask immediately, so `SidStart` is really a dummy value - only its address matters. Here is a function displaying information for the two common ACE types:

```
void DisplayAce(PACE_HEADER header, int index) {
    printf("ACE %2d: Size: %2d bytes, Flags: 0x%02X Type: %s\n",
        index, header->AceSize, header->AceFlags,
        AceTypeToString(header->AceType));  // simple enum to string
    switch (header->AceType) {
        case ACCESS_ALLOWED_ACE_TYPE:
        case ACCESS_DENIED_ACE_TYPE:    // have the same binary layout
        {
            auto data = (ACCESS_ALLOWED_ACE*)header;
            printf("\tAccess: 0x%08X %ws (%ws)\n", data->Mask,
                GetUserNameFromSid((PSID)&data->SidStart).c_str(),
                SidToString((PSID)&data->SidStart).c_str());
        }
        break;
    }
}
```

The *sd.exe* application allows viewing security descriptors of threads, processes, files, registry keys and named objects (mutexes, events, etc.). The above code is an excerpt from that application.

Here are some examples running *sd.exe*:

```
c:\>sd.exe
Usage: sd [[-p <pid>] | [-t <tid>] | [-f <filename>] | [-k <regkey>]
| [objectname]]
If no arguments specified, shows the current process security
descriptor
SD Length: 116 bytes
SD: O:BAD:(A;;0x1fffff;;;BA)(A;;0x1fffff;;;SY)
(A;;0x121411;;;S-1-5-5-0-687579)
Control: DACL Present, Self Relative
Owner: BUILTIN\Administrators (S-1-5-32-544)
DACL: ACE count: 3
ACE  0: Size: 24 bytes, Flags: 0x00 Type: ALLOW
    Access: 0x001FFFFF BUILTIN\Administrators (S-1-5-32-544)
ACE  1: Size: 20 bytes, Flags: 0x00 Type: ALLOW
    Access: 0x001FFFFF NT AUTHORITY\SYSTEM (S-1-5-18)
ACE  2: Size: 28 bytes, Flags: 0x00 Type: ALLOW
    Access: 0x00121411 NT AUTHORITY\LogonSessionId_0_687579
    (S-1-5-5-0-687579)

c:\>sd -p 4936
SD Length: 100 bytes
SD: O:S-1-5-5-0-340923D:(A;;0x1fffff;;;S-1-5-5-0-340923)
(A;;0x1400;;;BA)
Control: DACL Present, Self Relative
Owner: NT AUTHORITY\LogonSessionId_0_340923 (S-1-5-5-0-340923)
DACL: ACE count: 2
ACE  0: Size: 28 bytes, Flags: 0x00 Type: ALLOW
    Access: 0x001FFFFF NT AUTHORITY\LogonSessionId_0_340923
    (S-1-5-5-0-340923)
ACE  1: Size: 24 bytes, Flags: 0x00 Type: ALLOW
    Access: 0x00001400 BUILTIN\Administrators (S-1-5-32-544)

c:\>sd -f c:\temp\test.txt
SD Length: 180 bytes
SD: O:S-1-5-21-2575492975-396570422-1775383339-1001D:AI
    (D;;CCDCLCSWRPWPLOCRSDRC;;;S-1-5-21-2575492975-396570422-
    1775383339-1009)
    (A;ID;FA;;;BA)(A;ID;FA;;;SY)(A;ID;0x1200a9;;;BU)(A;ID;0x1301bf;;;AU)
Control: DACL Present, DACL Auto Inherited, Self Relative
Owner: PAVEL7540\pavel
    (S-1-5-21-2575492975-396570422-1775383339-1001)
```

```
DACL: ACE count: 5
ACE  0: Size: 36 bytes, Flags: 0x00 Type: DENY
        Access: 0x000301BF PAVEL7540\alice (S-1-5-21-2575492975-39657042\
2-1775383339-1009)
ACE  1: Size: 24 bytes, Flags: 0x10 Type: ALLOW
        Access: 0x001F01FF BUILTIN\Administrators (S-1-5-32-544)
ACE  2: Size: 20 bytes, Flags: 0x10 Type: ALLOW
        Access: 0x001F01FF NT AUTHORITY\SYSTEM (S-1-5-18)
ACE  3: Size: 24 bytes, Flags: 0x10 Type: ALLOW
        Access: 0x001200A9 BUILTIN\Users (S-1-5-32-545)
ACE  4: Size: 20 bytes, Flags: 0x10 Type: ALLOW
        Access: 0x001301BF NT AUTHORITY\Authenticated Users (S-1-5-11)
```

Some of the above output bears some explanation. A security descriptor has a string representation based on the *Security Descriptor Definition Language* (SDDL). There are functions to convert from the binary to the string representation and vice versa: `ConvertSecurityDescriptorToStringSecurityDescriptor` and `ConvertStringSecurityDescriptorToSecurityDescriptor`.

Security descriptors exist in two formats: *Self-relative* and *Absolute*. With Self-relative, the various pieces of the security descriptor are packed into one structure that is easy to move around. Absolute format has internal pointers to the security descriptor parts, so it cannot move around without modifying the internal pointers. The actual format doesn't matter in most cases, and conversion between the two formats is possible with `MakeAbsoluteSD` and `MakeSelfRelativeSD`.

The Default Security Descriptor

Nearly every kernel object creation function has a `SECURITY_ATTRIBUTES` structure as a parameter. As a reminder, this is what this looks like:

```
typedef struct _SECURITY_ATTRIBUTES {
    DWORD nLength;
    LPVOID lpSecurityDescriptor;
    BOOL bInheritHandle;
} SECURITY_ATTRIBUTES;
```

We used `bInheritHandle` as one way to implement handle inheritance, but `lpSecurityDescriptor` was always `NULL`. If the entire structure is not provided, this implies `NULL` in `lpSecurityDescriptor`. Does that mean the object has no protection? Not necessarily.

We can check the security descriptor attached to an object after creation by using `GetKernelObjectSecurity` like so:

```
BYTE buffer[1 << 10];
DWORD len;
HANDLE hEvent = ::CreateEvent(nullptr, FALSE, FALSE, nullptr);
::GetKernelObjectSecurity(hEvent,
    DACL_SECURITY_INFORMATION | OWNER_SECURITY_INFORMATION,
    (PSECURITY_DESCRIPTOR)buffer, sizeof(buffer), &len);
```

Now we can examine the resulting security descriptor. It turns out that objects that can have a name but don't have one (mutexes, events, semaphores, file mapping objects, ALPC ports) get a security descriptor with no DACL (everyone has access). However, named objects (including files and registry keys), do get a security descriptor with a default DACL. This default DACL comes from the access token, so we can look at it and even change it.

> Why are unnamed objects unprotected? The reasoning is probably that since the handle is private, it cannot be accessed from outside the process. Only injected code can touch these handles. And since such handles don't have any "identifying marks", it's unlikely a malicious agent would know what they're used for. Named objects, on the other hand, are visible. Other processes can attempt to open them by name, so some form of protection is prudent.

Querying the default DACL is just a matter of calling `GetTokenInformation` with the correct value:

```
HANDLE hToken;
::OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &hToken);
::GetTokenInformation(hToken, TokenDefaultDacl, buffer,
    sizeof(buffer), &len);
auto dacl = ((TOKEN_DEFAULT_DACL*)buffer)->DefaultDacl;
```

You can view the DACL of kernel objects using *Process Explorer*'s handles view or with my *System Explorer* in its handles and objects views.

Building Security Descriptors

The default security descriptor is usually fine, but sometimes you may want to tighten security or provide extra permissions to certain users or groups. For this, you'll need to build a new security descriptor or alter an existing one, before applying it to an object.

The following example builds a security descriptor for an event object, with an owner being the Administrators alias, with two ACEs in its DACL:

- The first allows all possible access to the event for the Administrators alias.

- The second allows only `SYNCHRONIZE` access to the event.

The code is not pretty, but here is one way to do this (error handling omitted):

```
BYTE sdBuffer[SECURITY_DESCRIPTOR_MIN_LENGTH];
auto sd = (PSECURITY_DESCRIPTOR)sdBuffer;
// initialize an empty security descriptor
::InitializeSecurityDescriptor(sd, SECURITY_DESCRIPTOR_REVISION);

// build an owner SID
BYTE ownerSid[SECURITY_MAX_SID_SIZE];
DWORD size;
::CreateWellKnownSid(WinBuiltinAdministratorsSid, nullptr,
    (PSID)ownerSid, &size);
// set the owner
::SetSecurityDescriptorOwner(sd, (PSID)ownerSid, FALSE);

// everyone SID
BYTE everyoneSid[SECURITY_MAX_SID_SIZE];
b = ::CreateWellKnownSid(WinWorldSid, nullptr,
    (PSID)everyoneSid, &size);

// build the DACL
EXPLICIT_ACCESS ea[2];
ea[0].grfAccessPermissions = EVENT_ALL_ACCESS;  // all access
ea[0].grfAccessMode = SET_ACCESS;
ea[0].grfInheritance = NO_INHERITANCE;
ea[0].Trustee.ptstrName = (PWSTR)ownerSid;
ea[0].Trustee.TrusteeForm = TRUSTEE_IS_SID;
ea[0].Trustee.TrusteeType = TRUSTEE_IS_ALIAS;

ea[1].grfAccessPermissions = SYNCHRONIZE;   // just SYNCHRONIZE
ea[1].grfAccessMode = SET_ACCESS;
ea[1].grfInheritance = NO_INHERITANCE;
ea[1].Trustee.ptstrName = (PWSTR)everyoneSid;
ea[1].Trustee.TrusteeForm = TRUSTEE_IS_SID;
ea[1].Trustee.TrusteeType = TRUSTEE_IS_WELL_KNOWN_GROUP;

PACL dacl;
// create the DACL with 2 entries (don't get existing DACL)
::SetEntriesInAcl(_countof(ea), ea, nullptr, &dacl);
// set the DACL in the security descriptor
::SetSecurityDescriptorDacl(sd, TRUE, dacl, FALSE);
```

```
// finally, create the object with the created SD
SECURITY_ATTRIBUTES sa = { sizeof(sa) };
sa.lpSecurityDescriptor = sd;

HANDLE hEvent = ::CreateEvent(&sa, FALSE, FALSE, nullptr);

// the DACL was allocated by SetEntriesInAcl
::LocalFree(dacl);
```

The APIs used in this example are not the only ones that are available for building a DACL and SIDs. One simple way (if you know SDDL) is to create the required security descriptor as a string and call `ConvertStringSecurityDescriptorToSecurityDescriptor` to convert it to a "real" security descriptor that can be directly used.

> SDDL is fully documented in the Microsoft documentation.

The above code created a security descriptor to be used when creating a kernel object. If an object already exists, there are several APIs that can be used to change existing values (read the docs for the details):

```
BOOL SetKernelObjectSecurity(   // most generic
    _In_ HANDLE Handle,
    _In_ SECURITY_INFORMATION SecurityInformation,
    _In_ PSECURITY_DESCRIPTOR SecurityDescriptor);

DWORD SetSecurityInfo(          // uses components of SD
    _In_     HANDLE                 handle,
    _In_     SE_OBJECT_TYPE         ObjectType,
    _In_     SECURITY_INFORMATION   SecurityInfo,
    _In_opt_ PSID                   psidOwner,
    _In_opt_ PSID                   psidGroup,
    _In_opt_ PACL                   pDacl,
    _In_opt_ PACL                   pSacl);

BOOL SetFileSecurity(          // specific for files
    _In_ LPCTSTR lpFileName,
    _In_ SECURITY_INFORMATION SecurityInformation,
    _In_ PSECURITY_DESCRIPTOR pSecurityDescriptor);

DWORD SetNamedSecurityInfo(     // named objects
    _In_ LPTSTR                pObjectName,
```

```
_In_ SE_OBJECT_TYPE        ObjectType,
_In_ SECURITY_INFORMATION  SecurityInfo,
_In_opt_ PSID              psidOwner,
_In_opt_ PSID              psidGroup,
_In_opt_ PACL              pDacl,
_In_opt_ PACL              pSacl);
```

User Access Control

User Access Control (UAC) is a feature introduced in Windows Vista that caused quite a few headaches to users and developers alike. In the pre-Vista days, users were created as local administrators, which is bad from a security standpoint. Any executing code had admin rights which means any malicious code could take control of the system.
Running with admin rights is convenient, as most operations just work, but can cause damage by writing to sensitive files or *HKEY_LOCAL_MACHINE* registry hive.

Windows Vista changed that. A created user was not necessarily an administrator, and even the first user, which must be a local administrator, was not operating with admin rights by default. In fact, *Lsass* created (upon a successful login), two access tokens for local admin users - one is the full admin token, and the other token was with standard user rights. By default, processes ran with the standard user right token.

Most processes don't need to execute with admin rights. Consider applications such as *Notepad*, *Word* or *Visual Studio*. In most cases, they can run perfectly fine with standard user rights. There may be cases, however, when it's desirable they run with admin rights. This can be done with *elevation* (discussed later).

Windows Vista made it easier to run with standard user rights by easing the requirements for some operations:

- The change time privilege was split into two privileges: change time and change time zone. "Change time zone" is granted to all users, while change time is only granted to administrators (this makes sense, as changing time zone is just changing the *view* of time, not time itself).
- Some configurations that previously were permitted to admin users only, were now allowed to standard users (e.g. wireless settings, some power options).
- Virtualization (discussed later).

If a process needs to run with admin rights, it requests elevation. This shows one of two dialogs - either a Yes/No approval (if the user is a true administrator), or a dialog box that requires username/password (the user has no admin rights and should call an admin from the IT department or some other user which is an administrator on the machine). The color of the dialog box indicates the level of danger of allowing the application to elevate to have admin rights:

- If the binary is signed by Microsoft, it appears in light blue color (blue is known as a relaxing color).
- If the binary is signed by another entity except Microsoft, it appears in a light gray color. (recent versions of window only use blue-ish color for signed binaries from any company)
- If the binary is unsigned, it appears in a bright orange/yellow color that draws the user's attention as this may be a dangerous executable.

The UAC dialog in the Control Panel has 4 levels to indicate in which circumstances should the elevation dialog box pop up (figure 16-19).

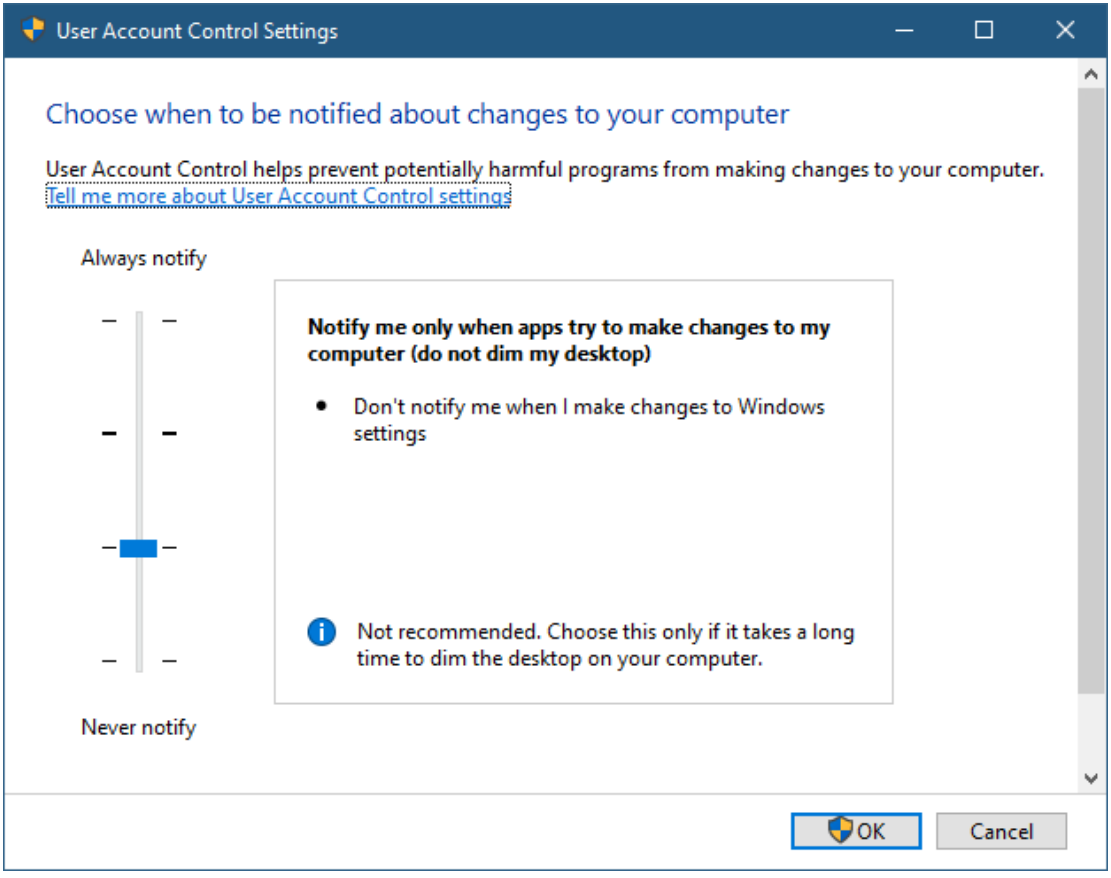


Figure 16-19: The UAC dialog

Although it looks like 4 options, there are actually 3 where elevation dialog is concerned:

- Always notify - any elevation is prompted by the appropriate dialog box.
- Never notify - if the user is a true administrator, just perform the elevation automatically. Otherwise, show the dialog for an admin username/password entry (also called *Administrator Approval Mode* - AAM)
- The middle options - if the user is a true admin, don't pop the consent dialog (Yes/No) for Windows components.

What does "Windows components" mean? It means applications created by the kernel team or very close to it. Example applications include *Task Manager*, *Task Scheduler*, *Device Manager* and *Performance Monitor*. Examples of applications not included (these are *Microsoft* components that are created by outside teams) are *cmd.exe*, *notepad.exe*, and *regedit.exe*. The latter get a consent dialog if the mid-levels are selected.

The difference between the two middle options is that the upper one (which is the default) shows the elevation dialog box in an alternate desktop (with the background set to a faded bitmap of the original wallpaper), while the lower one uses the default desktop.

On Windows versions before Windows 8, "Never notify" causes the system to use the pre-Vista model where there is just an admin token (if the user is a real admin). Starting with Windows 8, UAC cannot be turned off completely, because UWP processes always run with the standard token.

Elevation

Elevation is the act of running an executable with admin rights. The process of elevation is depicted in figure 16-20.

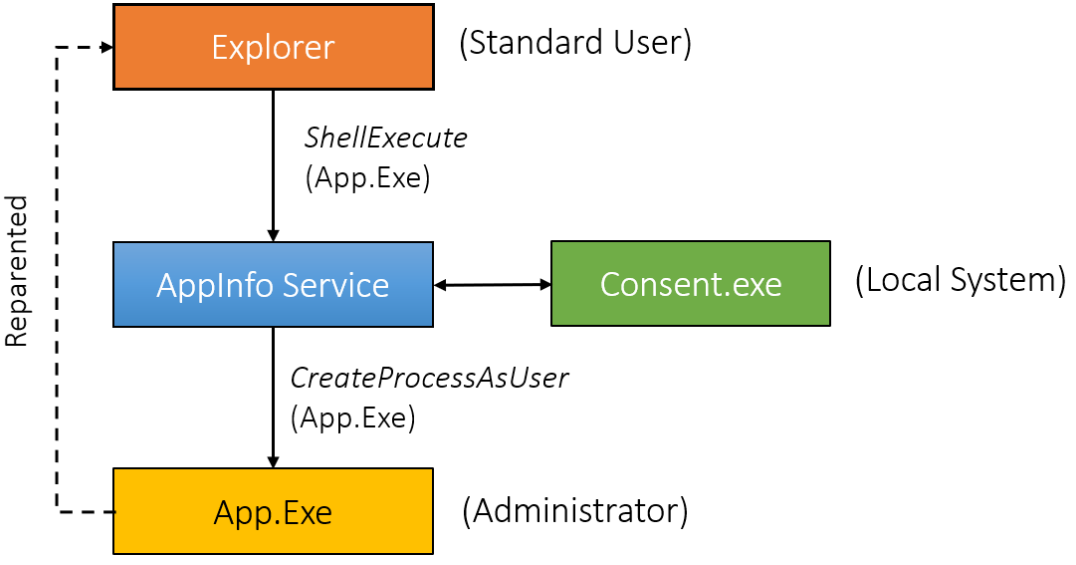


Figure 16-20: Elevation

The only documented way to launch a process elevated is to use the shell function *ShellExecute* or *ShellExecuteEx* (include <*shellapi.h*>):

```
HINSTANCE ShellExecute(
    _In_opt_ HWND hwnd,
    _In_opt_ LPCTSTR lpOperation,
    _In_ LPCTSTR lpFile,
    _In_opt_ LPCTSTR lpParameters,
    _In_opt_ LPCTSTR lpDirectory,
    _In_ INT nShowCmd);

typedef struct _SHELLEXECUTEINFO {
    DWORD cbSize;
    ULONG fMask;
    HWND hwnd;
    LPTSTR   lpVerb;
    LPTSTR   lpFile;
    LPTSTR   lpParameters;
    LPTSTR   lpDirectory;
    int nShow;
    HINSTANCE hInstApp;
    void *lpIDList;
    LPCTSTR  lpClass;
    HKEY hkeyClass;
    DWORD dwHotKey;
    union {
        HANDLE hIcon;
        HANDLE hMonitor;
    };
    HANDLE hProcess;
} SHELLEXECUTEINFO, *LPSHELLEXECUTEINFO;

BOOL ShellExecuteExW(_Inout_ SHELLEXECUTEINFOW *pExecInfo);
```

These functions are used by the shell (*Explorer*) and other applications to launch an executable based on something other than an executable's path (`CreateProcess` can do this just fine). These functions can accept any file, lookup its extension in the registry, and launch the relevant executable. For example, calling `ShellExecute(Ex)` with a *txt* file extension, launches *Notepad* (if that default has not been changed by the user) by calling `CreateProcess` with the correct values behind the scenes.

The crucial parameter for elevation purposes is `lpVerb`, which must be set to "runas". Here is an example for launching *notepad* elevated:

```
::ShellExecute(nullptr, L"runas", L"notepad.exe",
    nullptr, nullptr, SW_SHOWDEFAULT);
```

The process of elevation (figure 16-20) causes a message to be sent to the *AppInfo* service (hosted in a standard *Svchost.exe*). The service calls a helper executable, *consent.exe* that shows the relevant elevation

dialog box. If all goes well (the elevation is approved), the *AppInfo* service calls `CreateProcessAsUser` to launch the executable with the elevated token, and then the new process is "reparented", so that it looks like the original process created it (*Explorer* in figure 16-20, but it could be anyone calling `ShellExecute(Ex)`).

> This "reparenting" is fairly unique. UWP processes, for example (discussed in chapter 3), are always launched by the *DCOM Launch* service (also hosted in a standard *Svchost.exe* instance), but no reparenting is attempted.

Some applications provide a "Run as Administrator" option (see *WinObj* from *Sysinternals* as an example). Although it may seem the process suddenly becomes elevated, this is never the case. The current process exits and a new process is launched with the elevated. There is no way to elevate the token in place (if there was a way, UAC was useless).

Running As Admin Required

Some applications just cannot properly function when running with standard user rights. Such executables must somehow notify the system that they require elevation no matter what. This is accomplished by using a manifest file (discussed in chapter 1). One such part deals with elevation requirements. Here is the relevant parts:

```
<trustInfo xmlns="urn:schema-microsoft-com:asm.v3">
  <security>
    <requestedPrivileges>
      <requestedExecutionLevel Level="requireAdministrator" />
    </requestedPrivileges>
  </security>
</trustInfo>
```

The `Level` value indicates the type of elevation requested. These are the possible values:

- `asInvoker` - the default if no value is specified. Indicates the executable should launch elevated if its parent runs elevated, otherwise runs with standard user rights.
- `requireAdministrator` - indicates admin elevation is required. Without it, the process will not launch.
- `highestAvailable` - the in between value. It indicates that if the launching user is a true admin, then attempt elevation. Otherwise, run with standard user rights.

An example of `highestAvailable` is the registry editor (*regedit.exe*). If the user is a local admin, it requests elevation. Otherwise, it still runs. Some parts of *regedit* will not function, such as making changes in *HKEY_LOCAL_MACHINE*, which is not permitted for standard users; but the application is still usable.

In Visual Studio, it's easy to set one of these options without manually creating an XML file and specifying it as a manifest (figure 16-21).

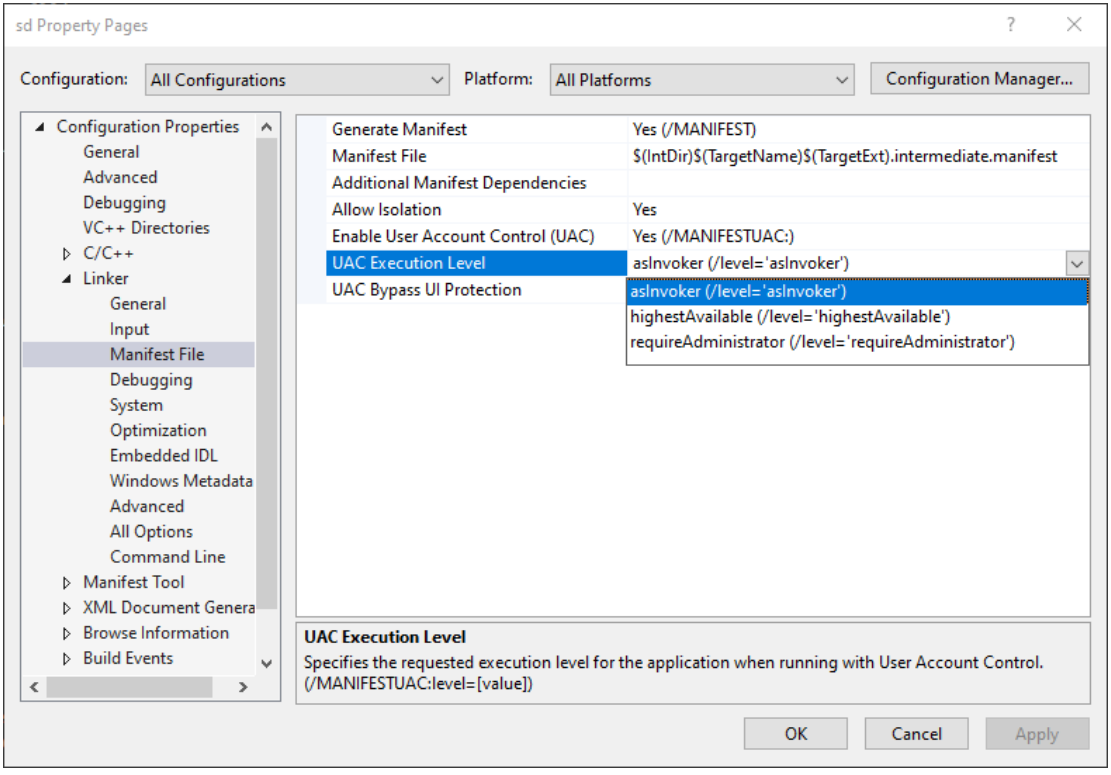


Figure 16-21: Elevation requirement in Visual Studio

UAC Virtualization

Many applications from the pre-Vista days were assuming (consciously or not) the user is a local administrator. Some of these applications perform operations that only succeed with admin rights like writing to system directories or writing to the *HKEY_LOCAL_MACHINE\Software* registry key. What should happen when these applications run on Vista or later, where the default token used is the standard user token?

The simplest option would be to return "access denied" errors to the application, but that would cause these applications to malfunction, since they have not been updated to take UAC into consideration. Microsoft's solution was *UAC Virtualization*, where such applications are redirected to a private area of the file system / registry under the user's files / *HKEY_CURRENT_USER* hive, so the calls don't fail.

This is a double-edged sword, however. If such an application makes a change to system files (or thinks it did), these are not really changed, so other applications that are not virtualized will not pick up the changes. The application will see its own changes - the system first looks at the private store and if not found, looks in the real locations.

For the file system, the private store is located at *C:\Users\<username>\AppData\Local\VirtualStore*. You can find more details by searching for "UAC Virtualization" online.

UAC Virtualization is applied automatically to executables that are dimmed "legacy", where "legacy" means: 32-bit executables, with no manifest indicating it's a Vista+ application. Regardless, UAC virtualization can be turned on for other processes by enabling it in the access token. You can view the *UAC virtualization* column in *Task Manager*. Three values are possible:

- Not Allowed - virtualization is disabled and cannot be enabled. This is the setting for system processes and services.
- Disabled - virtualization is not active.
- Enabled - virtualization is active.

You can perform a simple experiment to see UAC virtualization in action.

- Open *Notepad*, write something, and attempt to save the file to the *System32* directory. This will fail, as the process does not have permission to write to that directory.
- Go to *Task Manager*, right-click the *Notepad* process and enable UAC virtualization.
- Now attempt to save the file again in *Notepad*. This time this will succeed.
- Open *Explorer* and navigate to the *System32* directory. Notice the file you saved is not there. It's been saved to the virtual store because of virtualization.
- Disable virtualization for *Notepad* and open the file from *System32*. It won't be there.

Integrity Levels

Integrity Level is yet another feature introduced in Windows Vista (officially called *Mandatory Integrity Control*). One reason for its existence is to separate processes running with a standard rights token from processes running with an elevated token, under the same user. Clearly, the process running with the elevated token is more powerful and is a preferred target for attackers. The way to differentiate between them is by using integrity levels.

Integrity levels are represented by SIDs, and for processes are stored in the process access token. Table 16-3 shows the defined integrity levels.

Table 16-3: Standard integrity levels

Integrity level	SID	Remarks
System	S-1-16-16384	Highest, used by system processes and services
High	S-1-16-12288	Used by processes running with elevated token
Medium Plus	S-1-16-8448	
Medium	S-1-16-8192	Used by processes running with standard user rights
Low	S-1-16-4096	Used by UWP processes and most browsers

Process Explorer has an *Integrity Level* column that shows integrity levels for processes (figure 16-22).

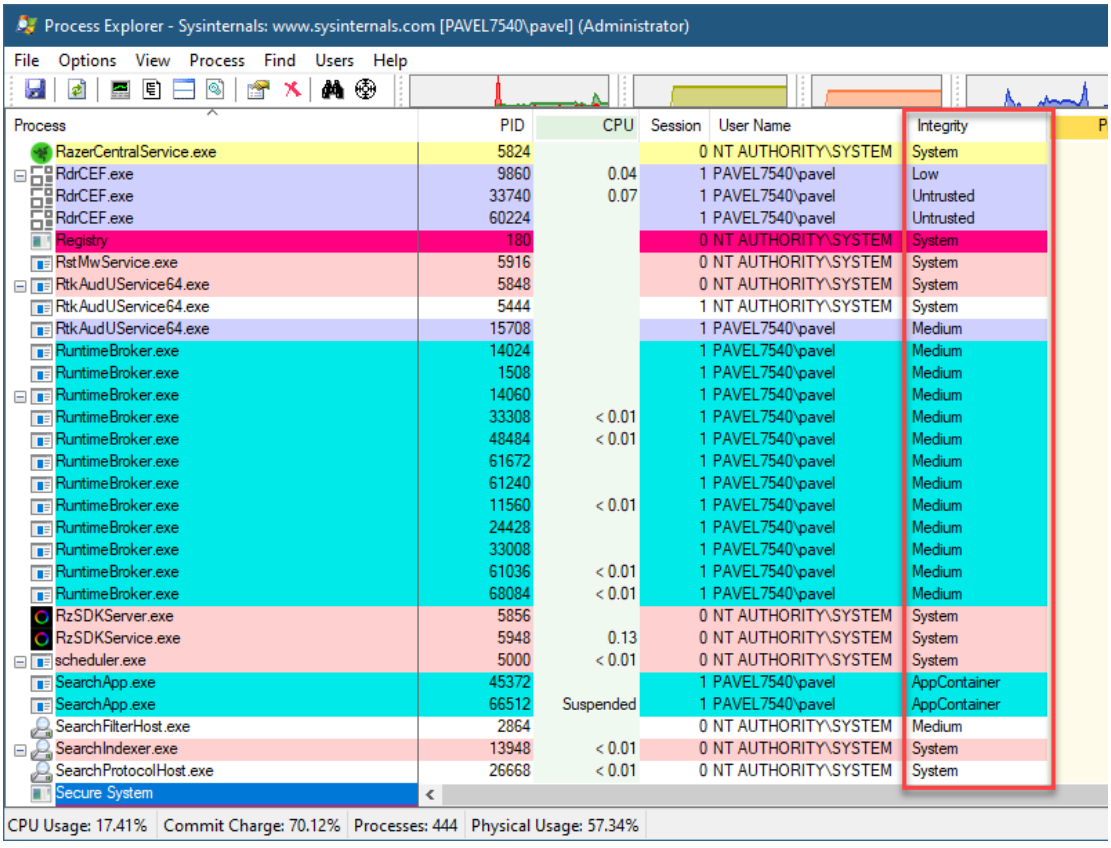


Figure 16-22: Integrity Levels in *Process Explorer*

Other integrity levels can be defined, using the last RID as as the "value". This is how integrity levels are compared.

> The *AppContainer* value for integrity level shown in figure 16-22 is equal to "Low", but the term *AppContainer* is used for the sandbox in which UWP processes live.

A related term to Integrity Level is *Mandatory Policy*, which indicates how the difference in integrity levels actually affects operations. The default is *No Write Up*, which means that when a process tries to access an object with a higher integrity level, a write type of access is not allowed. For example, process A with integrity level medium that wants to open a handle to a process with integrity level of high can only

be granted the following access masks: `PROCESS_QUERY_LIMITED_INFORMATION`, `SYNCHRONIZE` and `PROCESS_TERMINATE`.

What about objects that are not processes? All objects (including files) have an integrity level of *Medium* by default, unless explicitly changed by adding an ACE of type "Mandatory Label" with a different value. For processes, it's always possible to set a lower integrity level than the caller's token, but trying to set a higher integrity level is only possible if the caller has the *SeRelabelPrivilege*, normally not granted to anyone.

As another example, the fact that UWP processes run with low integrity means they cannot save data in common file locations like the user's documents or pictures, because these have medium integrity level. Most browsers today run their processes with low integrity level as well. This way, if a malicious file is downloaded and executed by such a browser, it will execute with a low integrity level, limiting its ability to do damage.

When an executable is launched, the integrity level of the new process is the minimum of the integrity level of the executable file and the caller's process token.

You can read the integrity level of a process by calling `GetTokenInformation` with the `TokenIntegrityLevel` enumeration value, and set with `SetTokenInformation`.

How does integrity level fit in with DACLs? The integrity level takes precedence. If the integrity level of the caller is equal or greater than the target object, a normal access check is made using DACLs. Otherwise, the "No Write-up" policy takes precedence.

For more information on integrity levels and related terms, consult the official documentation at https://docs.microsoft.com/en-us/windows/win32/secauthz/mandatory-integrity-control.

UIPI

User Interface Privilege Isolation (UIPI) is a feature based on integrity levels. Suppose there is a process with high integrity level that has windows (GUI). What happens if other processes, perhaps with lower integrity levels, send messages to those windows? Sending uncontrolled messages to a window can cause the thread that created the window to perform operations that should not normally be allowed by the calling process.

UIPI prevents such occurrences. A process cannot send messages to a window owned by another process with a higher integrity level, except for a few benign messages (for example: `WM_NULL`, `WM_GETTEXT` and `WM_GETICON`).

The higher integrity level process can allow certain messages to go through by calling `ChangeWindowMessageFilter` or `ChangeWindowMessageFilterEx`:

```
BOOL ChangeWindowMessageFilter(
    _In_ UINT message,
    _In_ DWORD dwFlag);

BOOL ChangeWindowMessageFilterEx(
    _In_ HWND hwnd,
    _In_ UINT message,
    _In_ DWORD action,
    _Inout_opt_ PCHANGEFILTERSTRUCT pChangeFilterStruct);
```

`ChangeWindowMessageFilter` accepts a message to let through or block and `dwFlag` can be `MSGFLT_ADD` (allow) or `MSGFLT_REMOVE` (block). This call affects all windows in the process.

Windows 7 added `ChangeWindowMessageFilterEx` to allow fine-grained control over each individual window. `action` can be `MSGFLT_ALLOW` (allow), `MSGFLT_DISALLOW` (block, unless allowed by a process-wide filter), or `MSGFLT_RESET` (reset to the process-wide setting). `pChangeFilterStruct` is a pointer to an optional structure that returns detailed information on the effects of calling these two functions. Consult the documentation for more information.

Specialized Security Mechanisms

The fundamental security mechanisms, such as security descriptors and privileges have been in place since the first version of Windows NT. Along the way, more security mechanisms were added to Windows, such as mandatory integrity control. Today, the attacks by malicious actors are more powerful than ever, and Windows tries to keep up by including new defense mechanisms. Many of these are beyond the scope of this book (e.g. Virtualization Based Security). In this section, we'll look at some mechanisms that can be leveraged programmatically.

Control Flow Guard

Control Flow Guard (CFG) was introduced in Windows 10 and Server 2016 to mitigate a certain type of attack, related to indirect calls. For example, a C++ virtual function call is done using a virtual table pointer that points to a virtual table where the actual target functions are stored. Figure 16-23 shows how an example C++ object looks like in memory if it has any virtual functions.

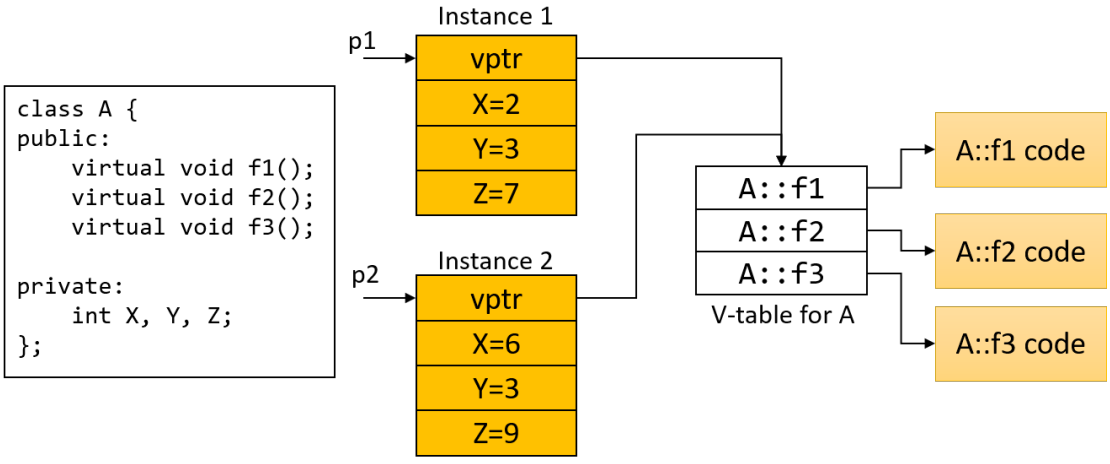


Figure 16-23: C++ objects with virtual functions

Each object starts with a *virtual table pointer* (vptr) that points to the virtual table for class A. A malicious agent that is injected into the process can write over the vptr (since it's read/write memory), and redirect the vptr to an alternate vtable of its choice (figure 16-24).

> The V-table mechanism is also used by COM classes, so CFG is relevant for such objects as well.

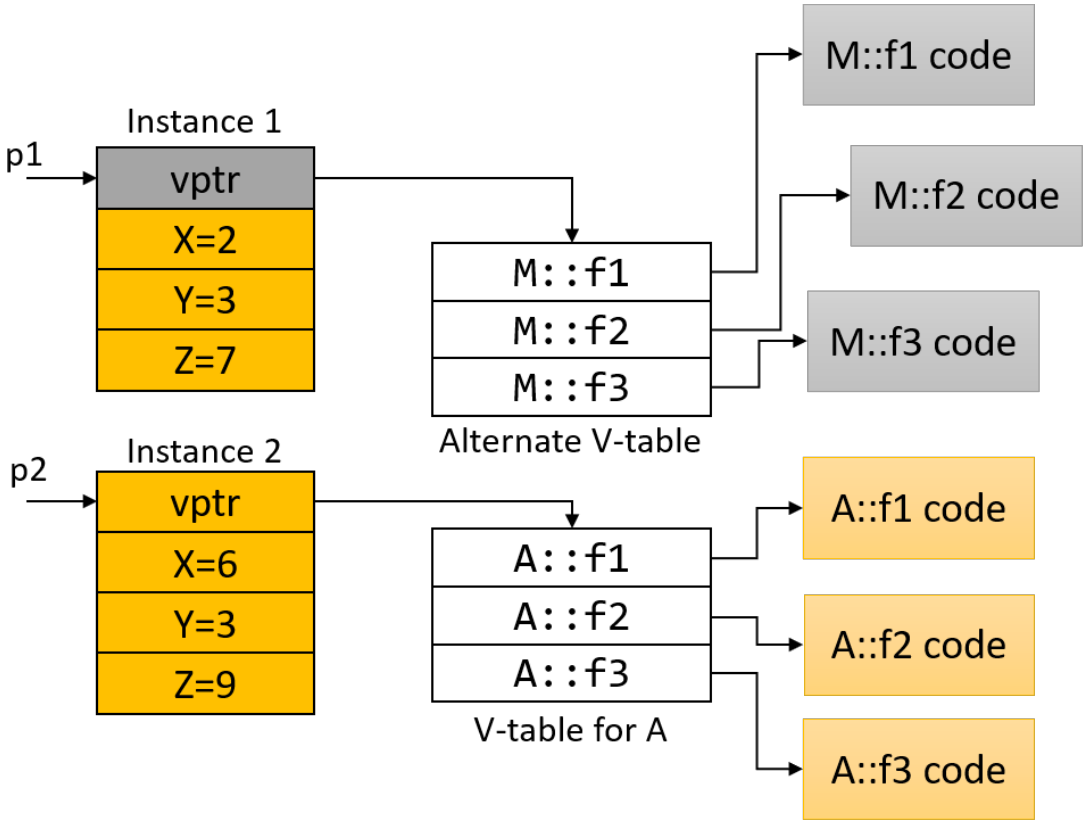


Figure 16-24: V-table redirection

CFG provides an extra check to be made before any indirect call. If the indirect call's target is not a function in one of the modules (DLLs and EXE) in the process, then it must have been redirected by some shellcode injected into the process, and in that case the process terminates. This procedure is depicted in figure 16-25.

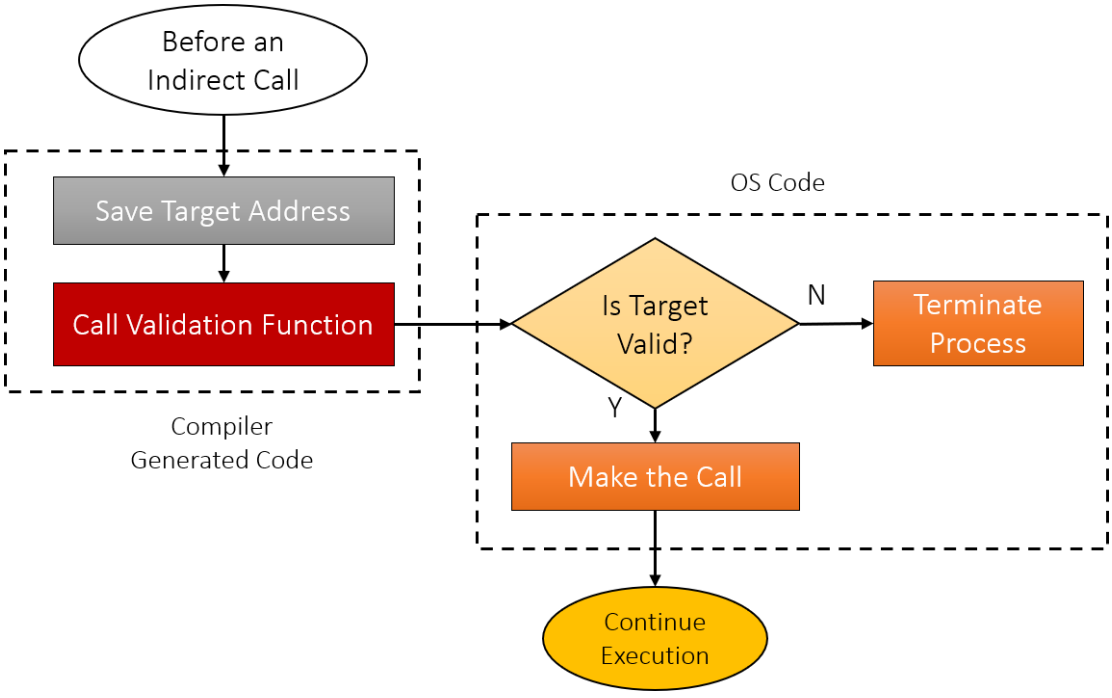


Figure 16-25: CFG at work

Getting CFG support is fairly straightforward, by selecting the CFG option in the project's properties in Visual Studio (figure 16-26). Note that CFG conflicts with "Debug Information for Edit and Continue", so the latter must be changed to "Program Database".

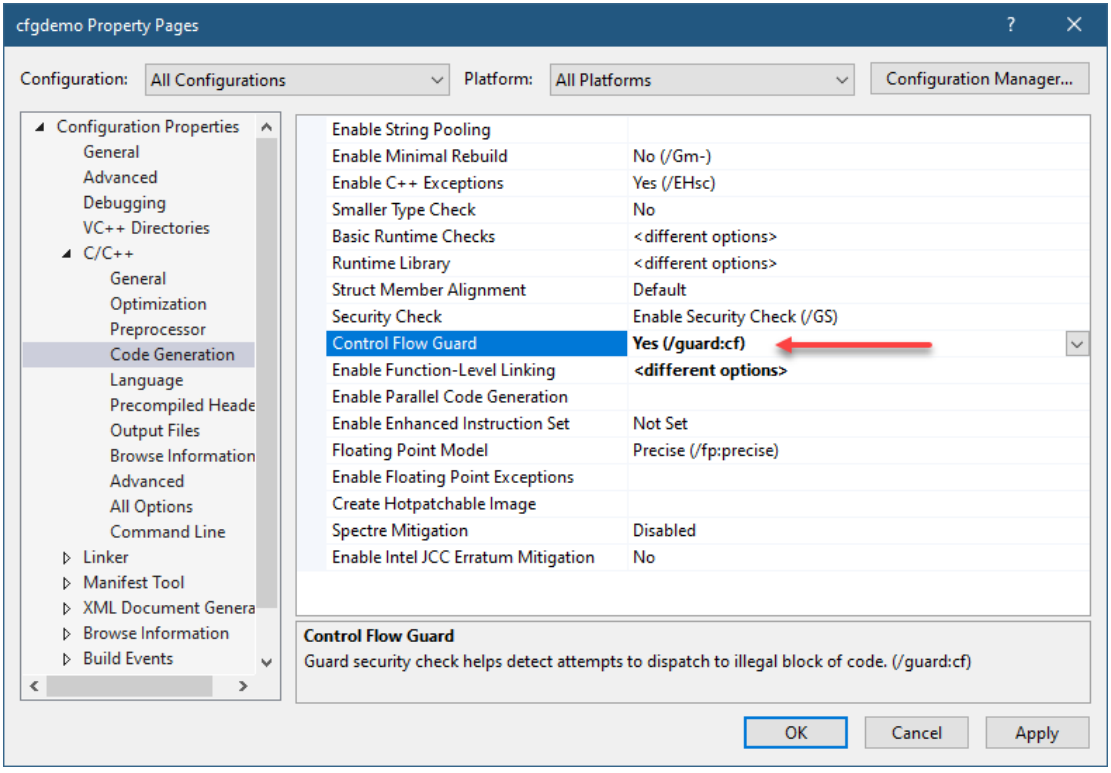


Figure 16-26: CFG options in Visual Studio

Here is an example of some C++ code (available in the *CfgDemo* application):

```
class A {
public:
    virtual ~A() = default;
    virtual void DoWork(int x) {
        printf("A::DoWork %d\n", x);
    }
};

class B : public A {
public:
    void DoWork(int x) override {
        printf("B::DoWork %d\n", x);
    }
};

void main() {
    A a;
```

```
    a.DoWork(10);
    B b;
    b.DoWork(20);

    A* pA = new B;
    pA->DoWork(30);

    delete pA;
}
```

Class A defines two virtual methods - the destructor and `DoWork`, so its v-table has two function pointers, in this order.

The calls to `a.DoWork` and `b.DoWork` don't need to happen polymorphically. The call `pA->DoWork` must be made polymorphically. Here is the assembly output for `pA->DoWork` before CFG is applied (x64, Debug):

```
; 29   :     pA->DoWork(30);

  000a5   mov    rax, QWORD PTR pA$[rsp]
  000aa   mov    rax, QWORD PTR [rax]
  000ad   mov    edx, 30
  000b2   mov    rcx, QWORD PTR pA$[rsp]
  000b7   call   QWORD PTR [rax+8]      ; normal call
```

You can see the value 30 put in `EDX`. `RCX` is the `this` pointer (all part of the x64 calling convention). `RAX` point to the vtable, and the call itself is made indirectly 8 bytes into the v-table, because `DoWork` is the second function (each function pointer is 8 bytes in 64-bit processes).

If assembly language is not your thing, you can safely skip this part and just know CFG works.

After CFG is applied, here is the resulting code for `pA->DoWork`:

```
; 29   :     pA->DoWork(30);

  000a5   mov  rax, QWORD PTR pA$[rsp]
  000aa   mov  rax, QWORD PTR [rax]
  000ad   mov  rax, QWORD PTR [rax+8]
  000b1   mov  QWORD PTR tv70[rsp], rax ; tv70=128
  000b9   mov  edx, 30
  000be   mov  rcx, QWORD PTR pA$[rsp]
  000c3   mov  rax, QWORD PTR tv70[rsp]
  000cb   call QWORD PTR __guard_dispatch_icall_fptr
```

The last line is the important one. It calls a function in *NtDll.dll* that checks if the call target is valid. If it is, it makes the call. Otherwise, it terminates the process.

Verify that the `delete pA` call (which invokes the destructor) is also invoked via CFG.

Binaries that support CFG have extra information in their PE that list the valid functions in the binary. This includes not just the exported functions, but all functions. You can view this information with *dumpbin.exe*:

```
C:\>dumpbin /loadconfig cfgdemo.exe
Microsoft (R) COFF/PE Dumper Version 14.27.28826.0
Copyright (C) Microsoft Corporation.  All rights reserved.

Dump of file c:\dev\temp\ConsoleApplication6\x64\Debug\cfgdemo.exe

File Type: EXECUTABLE IMAGE

  Section contains the following load config:
...
    0000000140000F008 Security Cookie
    0000000140015000 Guard CF address of check-function pointer
    0000000140015020 Guard CF address of dispatch-function pointer
    0000000140015010 Guard XFG address of check-function pointer
    0000000140015030 Guard XFG address of dispatch-function pointer
    0000000140015040 Guard XFG address of dispatch-table-function
        pointer
    0000000140013000 Guard CF function table
                  1C Guard CF function count
            00014500 Guard Flags
                       CF instrumented
                       FID table present
                       Export suppression info present
                       Long jump target table present
                0000 Code Integrity Flags
                0000 Code Integrity Catalog
            00000000 Code Integrity Catalog Offset
            00000000 Code Integrity Reserved
    0000000000000000 Guard CF address taken IAT entry table
                   0 Guard CF address taken IAT entry count
    0000000000000000 Guard CF long jump target table
                   0 Guard CF long jump target count
    0000000000000000 Guard EH continuation table
```

```
                  0 Guard EH continuation count
    0000000000000000 Dynamic value relocation table
    0000000000000000 Hybrid metadata pointer
    0000000000000000 Guard RF address of failure-function
    0000000000000000 Guard RF address of failure-function pointer
            00000000 Dynamic value relocation table offset
                0000 Dynamic value relocation table section
                0000 Reserved2
    0000000000000000 Guard RF address of stack pointer verification func\
tion pointer
            00000000 Hot patching table offset
                0000 Reserved3
    0000000000000000 Enclave configuration pointer
    0000000000000000 Volatile metadata pointer

    Guard CF Function Table

      Address
      --------
        0000000140001040  @ILT+48(??_EB@@UEAAPEAXI@Z)
        0000000140001050  @ILT+64(mainCRTStartup)
        0000000140001100  @ILT+240(??_Ebad_array_new_length@std@@UEAAPEA\
XI@Z)
        0000000140001110  @ILT+256(?DoWork@A@@UEAAXH@Z)
...
        00000001400013F0  @ILT+992(?DoWork@B@@UEAAXH@Z)
...
```

Notice the two `DoWork` mangled functions and the various CFG information.

How does CFG work? The loader creates a large reserved bit map, where each valid function "punches" a "1" bit into this large bit map. Checking if a function is valid is an *O(1)* operation, where the function pointer is quickly shifted to the right to get to the bit representing it in the bit map. If the bit is "1", the function is valid. If the bit is "0" or the memory is not committed, the address is bad and the process terminates.

The above explanation is not completely accurate, but it's good enough for this book's purposes. To get the exact details, consult the *Windows Internals, 7th edition, part 1* book.

Process Explorer has a CFG column for processes and modules. For processes, you can see the *Virtual Size* column is roughly 2 TB for 64-bit processes. Most of that memory is the CFG bit map, where most of the memory is reserved. You can verify this by opening the process in the *VMMap Sysinternals* tool. Figure 16-27 shows *VMMap* for a *Notepad* instance and its CFG bitmap.

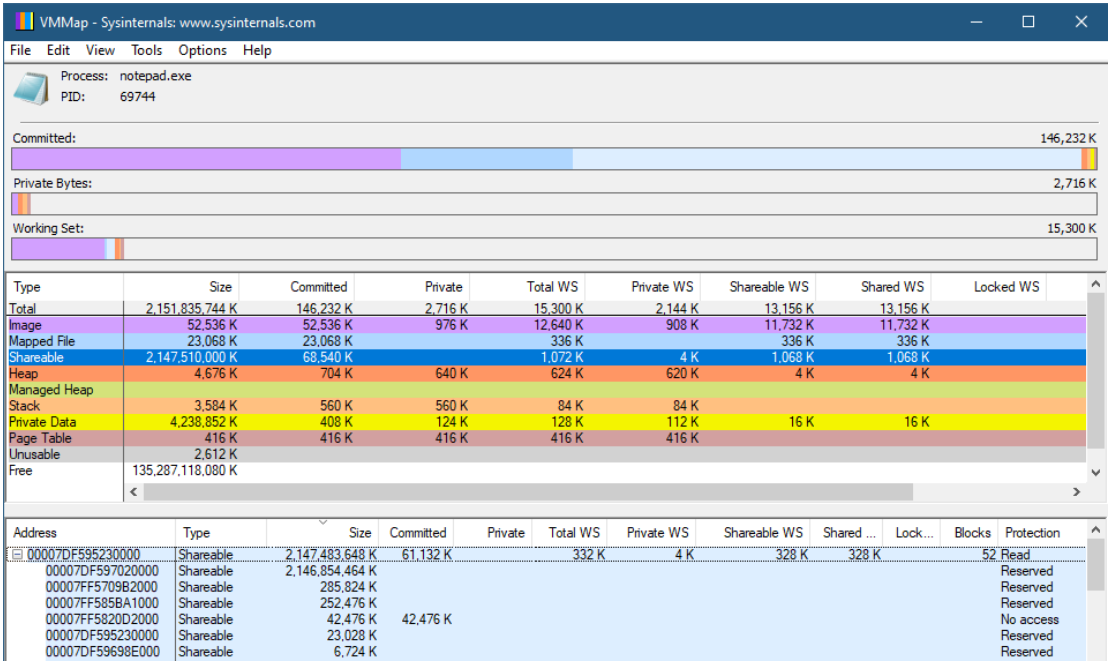


Figure 16-27: CFG bit map in *VMMap*

Process Mitigations

Windows 8 introduced *process mitigations*, the ability to set various security-related properties on a process, in a one-way fashion; once a mitigation is set, it cannot be revoked. (If it could, then malicious code could turn these mitigations off). The list of mitigations grows with almost every release of Windows.

There are four ways to set process mitigations:

- Using group policy settings controlled by administrators in an organization.
- Using the *Image File Execution Options* registry key based on an executable's name only (not its full path).
- By calling `CreateProcess` with a process attribute to set mitigations on the created process.
- By calling `SetProcessMitigationPolicy` from within the process.

Using group policy is not interesting for the purpose of this book. We met the *Image File Execution Options* (IFEO)registry key in chapter 13. The full key path is *HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options*. This is a key read by the loader when a process starts up to set various properties for the process. One of these is related to process mitigations. A convenient way to experiment with many of these mitigations is using my *GFlagsX* tool. Figure 16-28 shows *GFlagsX* open with the *Image* tab selected. Here you can see the list of executables that currently have some settings in their IFEO key. You can click *New Image...* and create a key for some executable (*Notepad.exe* in the example) (the extension is mandatory).

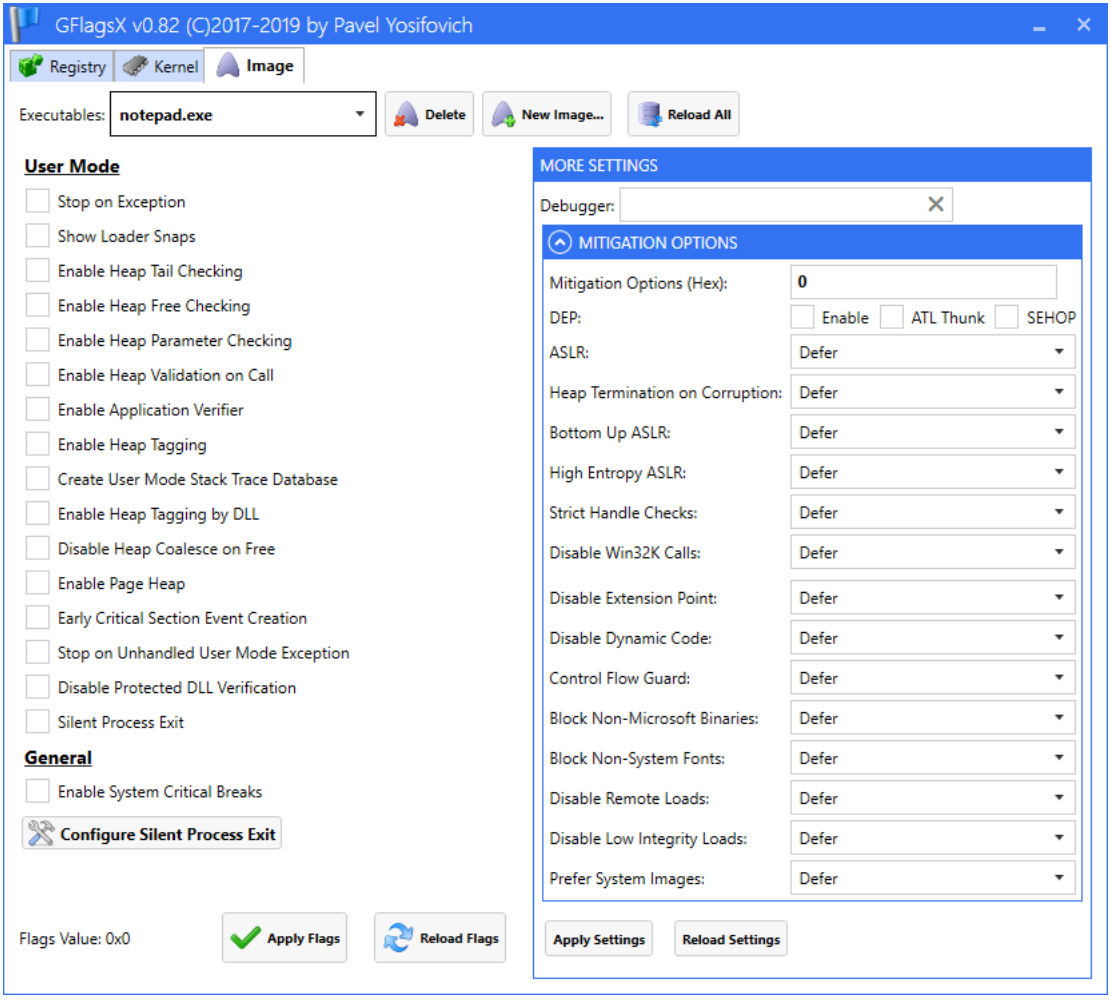


Figure 16-28: ***GFlagsX***

The left side of the window is related to *NT Global Flags*, that are not in the scope of this chapter. Some of them will be discussed in chapter 20. The right side shows the *Mitigation options* list. A detailed examination of all mitigation types is beyond the scope of this book (check the documentation for full details). Here is a brief explanation for a few of them:

- *Strict Handle Checks* - if an invalid handle is used, terminate the process instead of just returning an error. An invalid handle could be the result of injected malicious code closing the handle or misusing it.
- *Disable Win32K calls* - raises an exception if any *user32.dll* or *gdi32.dll* call is made. *Win32k.sys* is the kernel component of the Windows subsystem, and it has been used for various attacks in the past (and probably present and future). If the process is just a worker that does need a GUI, using this mitigation prevents Win32k exploits from this process.

- *Control Flow Guard* - requires all DLLs loaded to support CFG. Without it, non-CFG DLLs are loaded normally, and their entire memory has to be set as a valid target for CFG in that case.
- *Prefer System Images* - ensures that any DLL loaded that exists in the *System32* directory is preferred to any other location (This does not include *Known DLLs* which are always obtained from *System32*).

As a simple experiment, select *Disable Win32K Calls* for *Notepad.exe* and set it to *Always On* and click *Apply Settings*. Now try to launch *Notepad*. It should fail, since *Notepad* requires *user32.dll* and *gdi32.dll*. The value written to the registry by *GFlagsX* in this case is shown in figure 16-29.

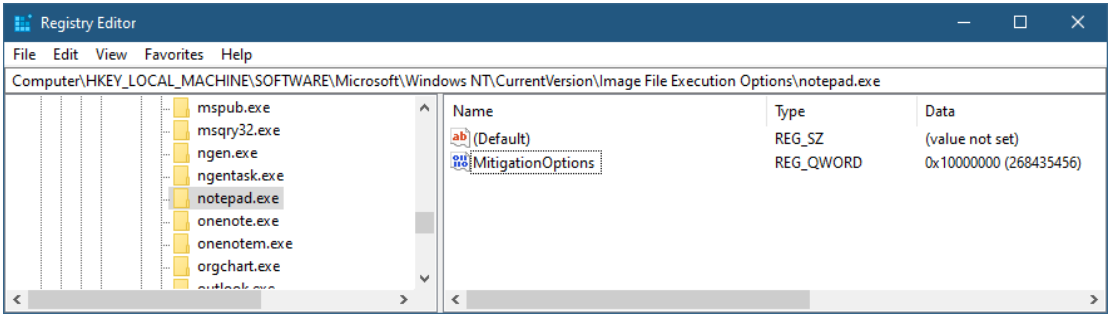


Figure 16-29: IFEO value settings for mitigation options

Make sure you remove this mitigation for *Notepad* (or erase the key completely) so *Notepad* can execute properly.

A parent process can set process mitigations to a child process by using the `PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY` process attribute. (Process attributes were discussed in chapter 3).

The following is an example that uses `CreateProcess` to launch an executable with the CFG mitigation in place:

```
HANDLE LaunchWithCfgMitigation(PWSTR exePath) {
    PROCESS_INFORMATION pi;
    STARTUPINFOEX si = { sizeof(si) };
    SIZE_T size;

    // the mitigation
    DWORD64 mitigation =
     PROCESS_CREATION_MITIGATION_POLICY_CONTROL_FLOW_GUARD_ALWAYS_ON;

    // get required size for one attribute
    ::InitializeProcThreadAttributeList(nullptr, 1, 0, &size);

    // allocate
    si.lpAttributeList = (PPROC_THREAD_ATTRIBUTE_LIST)::malloc(size);
```

```
    // initialize with one attribute
    ::InitializeProcThreadAttributeList(si.lpAttributeList, 1, 0,
        &size);

    // add the attribute we want
    ::UpdateProcThreadAttribute(si.lpAttributeList, 0,
        PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY,
        &mitigation, sizeof(mitigation), nullptr, nullptr);

    // create the process
    BOOL created = ::CreateProcess(nullptr, exePath, nullptr, nullptr,
        FALSE, EXTENDED_STARTUPINFO_PRESENT, nullptr, nullptr,
        (STARTUPINFO*)&si, &pi);

    // free resources
    ::DeleteProcThreadAttributeList(si.lpAttributeList);
    ::free(si.lpAttributeList);
    ::CloseHandle(pi.hThread);

    return created ? pi.hProcess : nullptr;
}
```

The child process has no "say" in the mitigations applied to it - it must be able to cope.

The last way to set mitigations is from the process itself by calling `SetProcessMitigationPolicy`. However, not all mitigation options are settable this way (check the docs for each mitigation's deatils).

```
BOOL SetProcessMitigationPolicy(
    _In_ PROCESS_MITIGATION_POLICY MitigationPolicy,
    _In_ PVOID lpBuffer,
    _In_ SIZE_T dwLength);
```

`PROCESS_MITIGATION_POLICY` is an enumeration with the various mitigations supported. `lpBuffer` is a pointer to the relevant structure depending on the type of mitigation. Finally, `dwLength` is the size of the buffer.

The following example shows how to set the load image policy mitigation options:

```
PROCESS_MITIGATION_IMAGE_LOAD_POLICY policy = { 0 };
policy.NoRemoteImages = true;
policy.NoLowMandatoryLabelImages = true;

::SetProcessMitigationPolicy(ProcessImageLoadPolicy,
    &policy, sizeof(policy));
```

Summary

Security in Windows is a big topic, that probably requires a book onto itself. In this chapter, we looked at the major concepts in the Windows security system, and examined various APIs for working with security. More information can be found in the official documentation and various online resources.

In the next chapter, we'll turn our attention to the most famous database in Windows - the Registry.

Chapter 17: The Registry

The Windows *Registry* is a foundational piece of Windows NT from its inception. It's a hierarchical database that stores information relevant to the system and its users. Some of the data is *volatile*, meaning it's generated while the system is running, and deleted when the system shuts down. *Non-volatile* data, on the other hand, is persisted in files.

Windows has several built-in tools for examining and manipulating the Registry. The primary GUI tool is *Regedit.exe*, while the classic command-line tool is *reg.exe*. There is another option for batch / command-line style manipulation using *PowerShell*.

Working with *reg.exe* and *PowerShell* is beyond the scope of this chapter.

The Registry was used quite heavily in the past by software developers to store various pieces of information for their applications, and this is still done to some extent. The recommendation is not to use the Registry to store application or user-related data. Instead, applications should store information in the file-system, typically in some convenient format, like INI, XML, JSON or YAML (to name a few of the common ones). The Registry should be left for Windows only. That said, it's sometimes convenient to store some information in the Registry if it's small. One common technique is to store a path to a file in a Registry value so that the bulk of the information is stored in that file, only pointed to by the Registry.

In this chapter, we'll examine the most important parts of the Registry and how to program against it.

In this chapter:

The Hives

The Registry is divided into *hives*, each one exposing certain pieces of information. Although *RegEdit.exe* shows 5 hives, there are just two "real" ones, *HKEY_USERS* and *HKEY_LOCAL_MACHINE*. All the others are made of some combination of data within these two "real" hives. Figure 17-1 shows the hives in *Regedit.exe*.

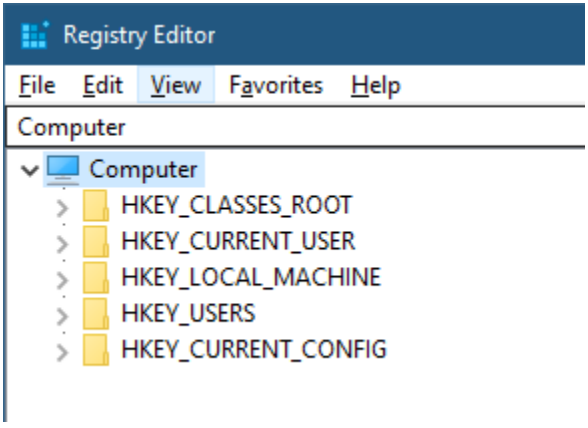


Figure 17-1: The hives

The following sections provide a brief description of the hives.

HKEY_LOCAL_MACHINE

This hive stores machine-wide information that is not specific to any user. Much of the data is very important for proper system startup, that care must be taken when any changes are made. By default, only admin-level users can make changes to this hive. Here are some of the important subkeys:

- *SOFTWARE* - this is where installed applications typically store their non-user-specific information. The common subkey pattern is

SOFTWARE\[CompanyName]\[ProductName]\[Version] (the "Version" part is not always used), which may be followed by more subkeys. For example, Microsoft Office stores its machine-wide information at *SOFTWARE\Microsoft\Office* with some pieces of information stored in a version subkey.

- *SYSTEM* - this is where most system parameters are stored and read by various system components when these start up. Here are some examples of subkeys with useful information for developers:
  - *SYSTEM\CurrentControlSet\Services* - stores information on services and device drivers installed on the system. We'll take a deeper look at this key in chapter 19 ("Services").
  - *SYSTEM\CurrentControlSet\Enum* - this subkey is the parent key for hardware device drivers.
  - *SYSTEM\CurrentControlSet\Control* - this subkey is the parent key for many knobs that various system components look at, such as the kernel itself, the session manager (*Smss.exe*), the Win32 subsystem process (*csrss.exe*), the service control manager (*Services.exe*) and others.

- *SYSTEM\BCD00000000* - stores *Boot Configuration Data* (BCD) information.
- *SYSTEM\SECURITY* - stores information for the local security policy (this key is inaccessible by default for administrators but is accessible to the *SYSTEM* account).
- *SYSTEM\SAM* - stores local user and group information (same access limitation as the above key).

You can view the *SAM* and *SECURITY* subkeys by running *Regedit.exe* with the *SYSTEM* account by using (for example) the *PsExec Sysinternals* tool like so: `psexec -s -i -d regedit`. The alternative is to exercise the *Take Ownership* privilege and change the DACL on these keys to allow administrator access (see chapter 16), but this is not recommended.

The subkey *SYSTEM\CurrentControlSet* is a link to the *SYSTEM\ControlSet001* subkey. The reason for this indirection has to do with an old feature of Windows NT called *Last Known Good*. In some cases, there may be more than one "control set". The subkey *SYSTEM\Select* holds values to indicate which is the "current" control set.

> Some of the details of *Last Known Good* are discussed in chapter 19.

The information in this hive is mostly persisted in files located in the *%SystemRoot%\System32\Config* directory. Table 17-1 lists the subkeys and their corresponding storage file (the format of these files is undocumented).

Table 17-1: Backup files for some hives

Subkey	File name
HKEY_LOCAL_MACHINE\SAM	SAM
HKEY_LOCAL_MACHINE\Security	SECURITY
HKEY_LOCAL_MACHINE\Software	SOFTWARE
HKEY_LOCAL_MACHINE\System	SYSTEM

The full list of hives and their storage files can be found in the Registry itself under the key *HKLM\System\CurrentControlSet\Control\hivelist*.

HKEY_USERS

The *HKEY_USERS* hive stores all per-user information for each user that ever logged on on the local system. Figure 17-2 shows an example of such a hive. Each user is represented by its SID (as a string).

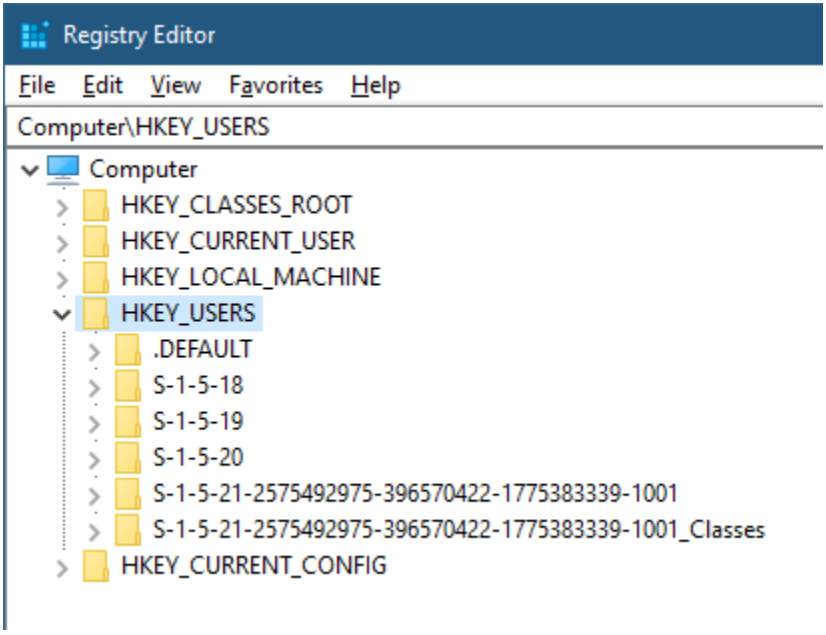


Figure 17-2: *HKEY_USERS* hive

The *.DEFAULT* subkey stores the default values newly created users get. The next three short SID values should look familiar from chapter 16 - they are for the *SYSTEM*, *Local Service* and *Network Service* accounts. Then the long random-like SID represents a "normal" user. The second SID that looks like the above one with the suffix "_Classes" is related to the *HKEY_CLASSES_ROOT* hive, described in a subsequent section.

If you open one of the SID subkeys, you'll discover various per-user settings related to the desktop, console, environment variables, colors, keyboard, printers, and more. These settings are read by various components, such as Windows Explorer, to tailor the environment to the user's wishes.

HKEY_CURRENT_USER (HKCU)

The *HKEY_CURRENT_USER* hive is a link to the current user running *RegEdit.exe*, showing the same information from *HKEY_USERS* for that user. The data in this hive is persisted in a hidden file named *NtUser.dat*, located in the user's directory (e.g. *c:\users\username*)

HKEY_CLASSES_ROOT (HKCR)

This is a rather curious hive, built from existing keys, combining the following:

- *HKEY_LOCAL_MACHINE\Software\Classes*
- *HKEY_CURRENT_USER\Software\Classes* (*HKEY_USERS\{UserSid}_Classes*)

In case of conflict, *HKEY_CURRENT_USER* settings override *HKEY_LOCAL_MACHINE*, since the user's choice should have higher priority than the machine default. *HKEY_CLASSES_ROOT* contains two pieces of information:

- Explorer Shell data: file types and associations, as well as shell extension information.
- *Component Object Model* (COM)-related information.

The Explorer shell information includes file type and associated actions. For example, searching for *.txt* in *HKEY_CLASSES_ROOT* finds a key, in which the default value is *txtfile*. Looking for a *txtfile* key locates the subkey *shell\open\command*, where its default value is *%SystemRoot%\System32\NOTEPAD.EXE %1*, clearly indicating *Notepad* as the default application to open *txt* files.

Other shell-related keys include the various shell extensions supported by the Explorer shell, such as custom icons, custom context menus, item previews, and even full-blown shell extensions. (Shell customization is beyond the scope of this book).

The more fundamentally important information in *HKEY_CLASSES_ROOT* has to do with COM registration. The details are important and discussed in length in chapter 21.

HKEY_CURRENT_CONFIG (HKCC)

This hive is just a link to *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Hardware Profiles\Current*. It's mostly uninteresting in the context of this book.

HKEY_PERFORMANCE_DATA

This hive is not visible in *Regedit.exe*, and for good reason. This hive serves as a legacy mechanism to consume *Performance Counters* (discussed in chapter 20). Starting with Windows 2000, there is a new API for working with performance counters, and it's preferred over using Registry APIs with *HKEY_PERFORMANCE_DATA*.

32-bit Specific Hives

On 64-bit systems, some parts of the Registry should have distinct keys for 32-bit vs. 64-bit. For example, application installation information is commonly stored in *HKLM\Software\{CompanyName}\{AppName}*. Some applications can be installed as 32-bit and 64-bit on the same system. In such cases, there must be a way to distinguish the 32-bit settings from the 64-bit settings.

32-bit processes that open the above-mentioned key receive another key that starts with *HKLM\Software\Wow6432Node*. This redirection is transparent and does not require the application to do anything special. 64-bit processes see the Registry as it is, with no such redirection taking place.

> This redirection is often referred to as *Registry Virtualization*.

HKLM\Software is not the only key that goes through redirection for 32-bit processes. Some COM-related information is also redirected from *HKCR* to *HKCR\Wow3264Node*. Here are some example subkeys that get redirected:

- *HKCR\CLSID* is redirected for all in-process (DLL) COM components (see chapter 21) to

HKCR\Wow6432Node\CLSID.

- *HKCR\AppID*, *HKCR\Interface*, and *HKCR\TypeLib* are redirected similarly to the Wow64 subkey.

32-bit processes get this redirection automatically, but such processes can opt-out of this redirection by specifying the `KEY_WOW64_64KEY` access flag in `RegCreateKeyEx` and `RegOpenKeyEx` functions (see next section). The opposite flag exists as well (`KEY_WOW64_32KEY`) to allow 64-bit processes to access the 32-bit registry parts without specifying *Wow6432Node* in key names.

Working with Keys and Values

Opening an existing Registry key is accomplished with `RegOpenKeyEx`:

```
LSTATUS RegOpenKeyEx(
    _In_ HKEY hKey,
    _In_opt_ LPCTSTR lpSubKey,
    _In_opt_ DWORD ulOptions,
    _In_ REGSAM samDesired,
    _Out_ PHKEY phkResult);
```

The first visible change from most other APIs is the return value. It's a 32-bit signed integer, returning the error code of the operation. This is the same value returned by `GetLastError` used with other APIs returning `BOOL`, `HANDLE` and similar. This means success is `ERROR_SUCCESS` (0); also, no point in calling `GetLastError` - in fact, this should be avoided, as its value does not change because of the Registry API call.

Now let's get to the actual function. `hKey` is the base key from which to interpret `lpSubKey`. This could be one of the predefined keys (`HKEY_LOCAL_MACHINE`, `HKEY_CURRENT_USER`, etc.) or a key handle from an earlier Registry API call. The subkey, by the way, is case insensitive.

`ulOptions` can be zero or `REG_OPTION_OPEN_LINK`, in the latter case if the key is a link (to another key), the link key itself is opened, rather than the link's target. Specifying zero is the common case.

`samDesired` is the required access mask to open the key with. If the access cannot be granted, the call fails with an "access denied" error code (5). Common access masks include `KEY_READ` for all query/enumerate operations and `KEY_WRITE` for write/modify values and creating subkeys operations. This is also where the 32-bit or 64-bit Registry view can be specified as described in the previous section. You can find the complete list of Registry key access masks in the documentation.

Finally, `phkResult` is the returned key handle if the call is successful. Notice that Registry keys have their own type (`HKEY`), which is no different than other `HANDLE` types (all are opaque `void*`), but Registry keys have their own close function:

```
LSTATUS RegCloseKey(_In_ HKEY hKey);
```

You may be wondering why Registry key handles are "special". One reason has to do with the fact that Registry keys may be opened to another machine's Registry (using the *Remote Registry* service, if enabled), which means that some close operations may involve communication with the remote system.

The non-Ex function (`RegOpenKey`) still exists and supported, mainly for compatibility with 16-bit Windows. There is no good reason to use this (and other similar) functions.

`RegOpenKeyEx` opens an existing key and fails if the key does not exist. To create a new key, call `RegCreateKeyEx`:

```
LSTATUS RegCreateKeyEx(
    _In_ HKEY hKey,
    _In_ LPCTSTR lpSubKey,
    _Reserved_ DWORD Reserved,
    _In_opt_ LPTSTR lpClass,
    _In_ DWORD dwOptions,
    _In_ REGSAM samDesired,
    _In_opt_ CONST LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    _Out_ PHKEY phkResult,
    _Out_opt_ LPDWORD lpdwDisposition);
```

`hKey` is the base key to start with, which can be a value obtained from a previous call to `RegCreateKeyEx` / `RegOpenKeyEx` or one of the standard predefined keys. The ability of the caller to create a new subkey depends on the security descriptor of the key `hKey`, rather than the access mask with which `hKey` was opened. `lpSubKey` is the subkey to create. It must be under `hKey` (directly or indirectly), meaning the subkey can have multiple subkeys separated by a backslash. If successful, the function creates all intermediate subkeys.

`Reserved` should be zero and `lpClass` should be `NULL`; both serve no purpose. `dwOptions` is usually zero (equivalent to `REG_OPTION_NON_VOLATILE`). This value indicates a non-volatile key, that is saved when the hive key is persisted. Additionally, the following combination of values can be specified:

- `REG_OPTION_VOLATILE` - the opposite of `REG_OPTION_NON_VOLATILE` - the key is created as volatile, meaning it's stored in memory, but is discarded once the hive key is unloaded.
- `REG_OPTION_CREATE_LINK` - creates a symbolic link key, rather than a "real" key. (See the section "Creating Registry Links", later in this chapter)
- `REG_OPTION_BACKUP_RESTORE` - the function ignores the `samDesired` parameter, and instead creates/opens the key with `ACCESS_SYSTEM_SECURITY` and `KEY_READ` if the caller has the *SeBackupPrivilege* in its token. If the caller has *SeRestorePrivilege* in its token, then `ACCESS_SYSTEM_SECURITY`, `DELETE` and `KEY_WRITE` are granted. If both privileges exist, then the resulting access is a union of both, effectively granting full access to the key.

`samDesired` is the usual access mask the caller requests. `lpSecurityAttributes` is the usual `SECURITY_ATTRIBUTES` we are familiar with. `phkResult` is the resulting key if the operation is successful. Finally, the last (optional) parameter, `lpdwDisposition` returns whether the key was actually created (`REG_CREATED_NEW_KEY`) or an existing key was opened (`REG_OPENED_EXISTING_KEY`). A newly created key has no values.

Reading Values

With an open key (whether created or opened), several operations are possible. The most basic is reading and writing values. Reading a value is possible with `RegQueryValueEx`:

```
LSTATUS RegQueryValueEx(
    _In_ HKEY hKey,
    _In_opt_ LPCTSTR lpValueName,
    _Reserved_ LPDWORD lpReserved,
    _Out_opt_ LPDWORD lpType,
    _Out_ LPBYTE lpData,
    _Inout_opt_ LPDWORD lpcbData);
```

`hKey` is the key from which to read, which can be a previously opened key or one of the predefined keys (including the less common ones, like `HKEY_PERFORMANCE_DATA`). `lpValueName` is the value name to query. If it's `NULL` or an empty string, the default value of the key is retrieved (if any).

`lpReserved` is just that, and should be set to `NULL`. `lpType` is an optional pointer returning the type of the returned data, one of the values shown in table 17-2.

Table 17-2: Registry value types

Value	Description
`REG_NONE` (0)	No value type
`REG_SZ` (1)	`NULL`-terminated Unicode string
`REG_EXPAND_SZ` (2)	`NULL`-terminated Unicode string (may contain unexpanded environment variables in %%)
`REG_BINARY` (3)	Binary (any) data
`REG_DWORD` (4)	32-bit number (little endian)
`REG_DWORD_LITTLE_ENDIAN` (4)	Same as above
`REG_DWORD_BIG_ENDIAN` (5)	32-bit number (big endian)
`REG_LINK` (6)	Symbolic link (Unicode)
`REG_MULTI_SZ` (7)	Multiple Unicode strings separated by `NULL`, second `NULL` terminates
`REG_RESOURCE_LIST` (8)	`CM_RESOURCE_LIST` structure (useful in kernel mode only)

Table 17-2: Registry value types

Value	Description
`REG_FULL_RESOURCE_DESCRIPTOR` (9)	`CM_FULL_RESOURCE_DESCRIPTOR` (useful in kernel mode only)
`REG_RESOURCE_REQUIREMENTS_LIST` (10)	Useful in kernel mode only
`REG_QWORD` (11)	64-bit number (little endian)
`REG_QWORD_LITTLE_ENDIAN` (11)	Same as above

`lpType` can be specified as `NULL` in case the caller is not interested in this information. This is typically the case when the caller knows what to expect. `lpData` is a caller-allocated buffer to the data itself. For some value types, the size is constant (such as 4 bytes for `REG_DWORD`), but others are dynamic (e.g. `REG_SZ`, `REG_BINARY`), meaning the caller needs to allocate a large-enough buffer, otherwise only part of the data is copied, and the function returns `ERROR_MORE_DATA`. The size of the caller's buffer is specified with the last parameter. On input, it should contain the caller's buffer size. On output, it contains the number of bytes written. If the caller needs the data size, it can pass `NULL` for `lpData` and get back the size in `lpcbData`.

The following example shows how to read the string value at *HKCU\Console\FaceName* (error handling omitted):

```
HKEY hKey;
::RegOpenKeyEx(HKEY_CURRENT_USER, L"Console", 0, KEY_READ, &hKey);
DWORD type;
DWORD size;
// first call to get size
::RegQueryValueEx(hKey, L"FaceName", nullptr, &type, nullptr, &size);
assert(type == REG_SZ);
// returned size includes the NULL terminator
auto value = std::make_unique<BYTE[]>(size);
::RegQueryValueEx(hKey, L"FaceName", nullptr, &type,
    value.get(), &size);
::RegCloseKey(hKey);
printf("Value: %ws\n", (PCWSTR)value.get());
```

Another function is available for retrieving values, `RegGetValue`:

```
LSTATUS RegGetValue(
    _In_ HKEY hkey,
    _In_opt_ LPCSTR lpSubKey,
    _In_opt_ LPCSTR lpValue,
    _In_ DWORD dwFlags,
    _Out_opt_ LPDWORD pdwType,
    _Out_ PVOID pvData,
    _Inout_opt_ LPDWORD pcbData);
```

The function is similar to `RegQueryValueEx`, but adds the nice option (via the `dwFlags` parameter) of restricting the value type(s) that may be returned. This allows a caller to get a failure if the value it expects is not of the expected type (and saves the time it takes to retrieve the data). The `dwFlags` value can be a combination of the values shown in table 17-3.

Table 17-3: Flags to **`RegGetValue`**

Value	Description
`RRF_RT_REG_NONE` (1)	Allow `REG_NONE` type
`RRF_RT_REG_SZ` (2)	Allow `REG_SZ` type
`RRF_RT_REG_EXPAND_SZ` (4)	Allow `REG_EXPAND_SZ` type. Expand environment variables unless the flag `RRF_NOEXPAND` is specified
`RRF_RT_REG_BINARY` (8)	Allow `REG_BINARY` type
`RRF_RT_REG_DWORD` (0x10)	Allow `REG_DWORD` type
`RRF_RT_REG_MULTI_SZ` (0x20)	Allow `REG_MULTI_SZ` type
`RRF_RT_REG_QWORD` (0x40)	Allow `REG_QWORD` type
`RRF_RT_DWORD`	`RRF_RT_REG_BINARY | RRF_RT_REG_DWORD`
`RRF_RT_QWORD`	`RRF_RT_REG_BINARY | RRF_RT_REG_QWORD`
`RRF_RT_ANY` (0x0000ffff)	no type restriction
`RRF_SUBKEY_WOW6464KEY` (0x10000)	(Win 10+) open the 64-bit key (if subkey is not `NULL`)
`RRF_SUBKEY_WOW6432KEY` (0x20000)	(Win 10+) open the 32-bit key (if subkey is not `NULL`)
`RRF_NOEXPAND` (0x10000000)	do not expand a `REG_EXPAND_SZ` result
`RRF_ZEROONFAILURE` (0x20000000)	fill the buffer with zeros on failure

Writing Values

To write a value to a Registry key, call `RegSetValueEx`:

```
LSTATUS RegSetValueEx(
    _In_ HKEY hKey,
    _In_opt_ LPCTSTR lpValueName,
    _Reserved_ DWORD Reserved,
    _In_ DWORD dwType,
    _In_ CONST BYTE* lpData,
    _In_ DWORD cbData);
```

Most of the parameters should be self-explanatory at this point. `hKey` is the key to write to, which must have at least the `KEY_SET_VALUE` access mask (typically, the key is opened with the `KEY_WRITE` access mask that includes `KEY_SET_VALUE`). `lpValueName` is the value name to set, and is not case sensitive. If this name is `NULL` or an empty string, it sets the default value (which can be of any type). The `Reserved` argument must be zero.

The data type is specified with the `dwType` parameter, and must be one of the values in table 17-2 (see the section *Creating Registry Links* for the special case of `REG_LINK`). The data itself consists of a generic pointer (`lpData`) and size. The data must be appropriate to the type specified. Some of the types have fixed sizes (e.g. `REG_DWORD`), while others can be of arbitrary length (e.g. `REG_SZ`, `REG_BINARY`, `REG_MULTI_SZ`). Be sure to end a string (`REG_SZ`) with a `NULL` terminator, and end `MULTI_SZ` with two `NULL` terminators. Note that the size specified (`cbData`) is always in bytes, regardless of the value type. For strings, this size must include the terminating `NULL`(s).

If the value specified already exists, it's overwritten with the new value.

The following example changes the *FaceName* value at *HKEY_CURRENT_USER\Console* to "Arial" (error handling omitted):

```
HKEY hKey;
::RegOpenKeyEx(HKEY_CURRENT_USER, L"Console", 0, KEY_WRITE, &hKey);
WCHAR value[] = L"Arial";
::RegSetValueEx(hKey, L"FaceName", 0, REG_SZ,
    (const BYTE*)value, sizeof(value));
::RegCloseKey(hKey);
```

An alternative function is available for the same purpose, `RegSetKeyValue`:

```
LSTATUS RegSetKeyValue(
    _In_ HKEY hKey,
    _In_opt_ LPCTSTR lpSubKey,
    _In_opt_ LPCTSTR lpValueName,
    _In_ DWORD dwType,
    _In_reads_bytes_opt_(cbData) LPCVOID lpData,
    _In_ DWORD cbData);
```

The function is almost identical to `RegSetValueEx`. It's sometimes more convenient to use because it allows specifying a subkey ('lpSubKey) `relative to` hKey', which is the final key where to set the value. This avoids the need to open the subkey explicitly if a handle to it is not readily available.

Deleting Keys and Values

A Registry key can be deleted by calling `RegDeleteKey` or `RegDeleteKeyEx`:

```
LSTATUS RegDeleteKey (
    _In_ HKEY hKey,
    _In_ LPCTSTR lpSubKey);
LSTATUS RegDeleteKeyEx(
    _In_ HKEY hKey,
    _In_ LPCTSTR lpSubKey,
    _In_ REGSAM samDesired,
    _Reserved_ DWORD Reserved);
```

`RegDeleteKey` is the simplest function, deleting the subkey (and all its values) relative to the open `hKey`. The access mask used to open the key does not matter - it's the security descriptor on the key that indicates whether the caller can perform the delete operation.

`RegDeleteKeyEx` adds the option to change the registry view by specifying `KEY_WOW64_32KEY` or `KEY_WOW64_64KEY` for the `samDesired` parameter. See the discussion in the section "32-bit Specific Hives" earlier in this chapter.

The deleted key is marked for deletion, and is only deleted once all open handles to the key are closed. The above delete functions can only delete a key with no-sub keys. If the key has subkeys, the functions fail and return `ERROR_ACCESS_DENIED` (5).

To delete a key with all its subkeys, call `RegDeleteTree`:

```
LSTATUS RegDeleteTree(
    _In_ HKEY hKey,
    _In_opt_ LPCTSTR lpSubKey);
```

`hKey` must be opened with the following rights: `DELETE`, `KEY_ENUMERATE_SUB_KEYS`, `KEY_QUERY_VALUE` and (if the key has any values) `KEY_SET_VALUE`. If `lpSubKey` is `NULL`, all keys and values are removed from the key specified by `hKey`.

Deleting values is simple enough with `RegDeleteKeyValue` or `RegDeleteValue`:

```
LSTATUS RegDeleteValue(
    _In_ HKEY hKey,
    _In_opt_ LPCTSTR lpValueName);
LSTATUS RegDeleteKeyValue(
    _In_ HKEY hKey,
    _In_opt_ LPCTSTR lpSubKey,
    _In_opt_ LPCTSTR lpValueName);
```

`hKey` must be opened with `KEY_SET_VALUE` access mask. `RegDeleteValue` deletes the given value in `hKey`, and `RegDeleteKeyValue` allows specifying a subkey to reach from the given key from which to delete a value.

`RegDeleteValue` is better for performance reason, because internally, whenever a subkey is provided, the API opens the subkey, performs the operation and closes the subkey. This is also true for other functions where a subkey is optional. If you have a direct key handle, it's always faster to use.

Creating Registry Links

The Registry supports links - keys that point to other keys. We've already met a few of these. For example, the key *HKEY_LOCAL_MACHINE\System\CurrentControlSet* is a symbolic link to *HKEY_LOCAL_MACHINE\System\ControlSet001* (in most cases). Looking at such keys with *RegEdit.exe*, symbolic links look like normal keys, in the sense that they behave as the link's target. Figure 17-3 shows the above-mentioned keys - they look and behave exactly the same.

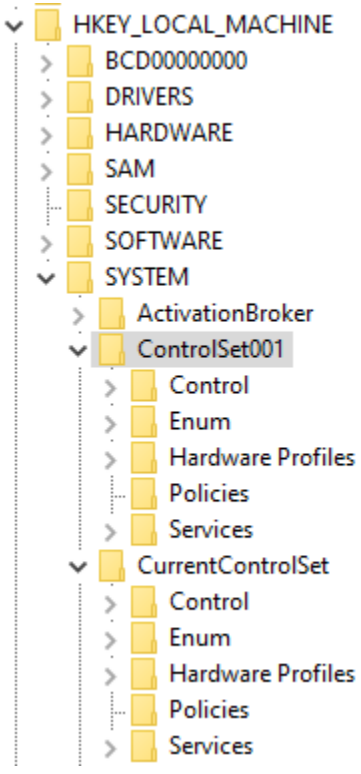


Figure 17-3: A key and a link to that key

My own Registry editor/viewer, *RegExp.exe* (Registry Explorer) does show links with a different icon (figure 17-4). It also reveals the way the link's target is stored - using the value name *SymbolicLinkName*.

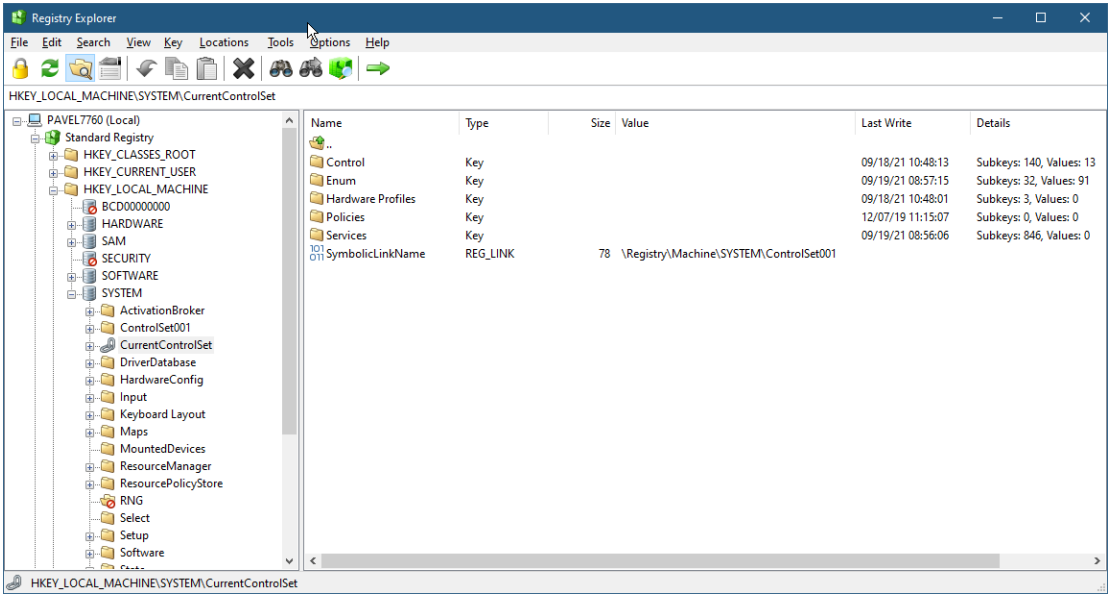


Figure 17-4: A key and a link to that key in *RegExp.exe*

The Microsoft documentation does provide full information on how to create Registry links. The first step is to create the key and specify it to be a link rather than a normal key. The following example assumes our intention is to create a link under *HKEY_CURRENT_USER* named *DesktopColors* that links to *HKEY_CURRENT_USER\Control Panel\Desktop\Colors*. The following snippet creates the required key as a link by specifying the option `REG_OPTION_CREATE_LINK` in the call to `RegCreateKeyEx` (error handling omitted):

```
HKEY hKey;
::RegCreateKeyEx(HKEY_CURRENT_USER, L"DesktopColors", 0, nullptr,
    REG_OPTION_CREATE_LINK, KEY_WRITE, nullptr, &hKey, nullptr);
```

Now comes the first tricky part. The documentation states that the link's target should be written to a value named *SymbolicLinkValue* and it must be an absolute registry path. The issue here is the "absolute path" required is **not** something like *HKEY_CURRENT_USER\Control Panel\Desktop\Colors*. Instead, it must be the absolute path in the way the kernel sees the Registry. You can see what this looks like in *RegExp.exe* if you open the tree node named "Registry" (figure 17-5).

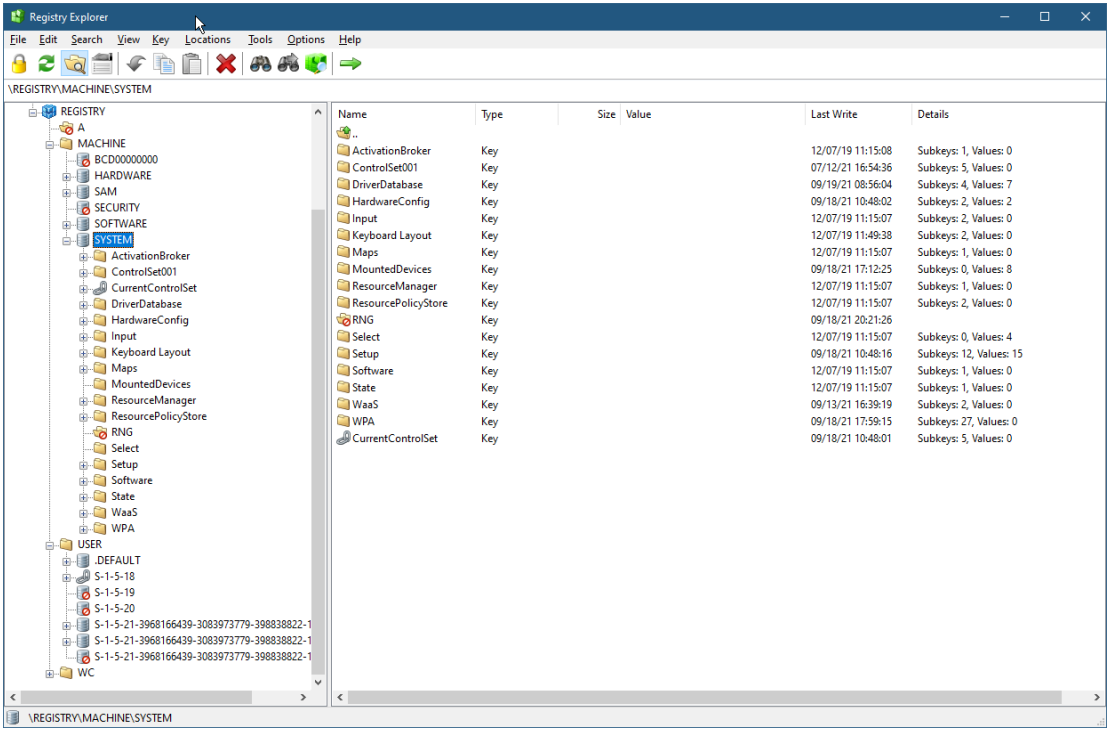


Figure 17-5: The "real" Registry in *RegExp.exe*

This means that *HKEY_CURRENT_USER* must be translated to *HKEY_USERS*. This can be done with a hardcoded SID string, but it would be better to obtain this dynamically. Fortunately, based on the information detailed in chapter 16, we can get the current user's SID as a string like so (error handling omitted yet again):

```
HANDLE hToken;
::OpenProcessToken(::GetCurrentProcess(), TOKEN_QUERY, &hToken);
// Win8+: HANDLE hToken = ::GetCurrentProcessToken();
BYTE buffer[sizeof(TOKEN_USER) + SECURITY_MAX_SID_SIZE];
DWORD len;
::GetTokenInformation(hToken, TokenUser, buffer,
    sizeof(buffer), &len);
::CloseHandle(hToken);
auto user = (TOKEN_USER*)buffer;
PWSTR stringSid;
::ConvertSidToStringSid(user->User.Sid, &stringSid);

// use stringSid...

::LocalFree(stringSid);
```

Now we can compose the absolute path like so:

```
// using std::wstring for convenience
std::wstring path = L"\\REGISTRY\\USER\\";
path += stringSid;
path += L"\\Control Panel\\Desktop\\Colors";
```

The second tricky part is that the link's path must be written to the registry **without** the terminating `NULL` bytes to the value *SymbolicLinkValue*:

```
::RegSetValueEx(hKey, L"SymbolicLinkValue", 0, REG_LINK,
    (const BYTE*)path.c_str(),
    path.size() * sizeof(WCHAR));   // NULL terminator not counted
```

This gets the job done. Deleting a Registry link cannot be accomplished with `RegDeleteKey(Ex)` discussed in the previous section. If you try it, it deletes the target of the link, rather than the link itself. A link can only be deleted by using the pseudo-documented native function `NtDeleteKey` (from *Ntdll.dll*). To use it, we first must declare it and link against the *ntdll* import library (the other option for linking is to dynamically call `GetProcAddress` to discover the address of `NtDeleteKey`):

```
extern "C" int NTAPI NtDeleteKey(HKEY);
#pragma comment(lib, "ntdll")
```

Now we can delete the symbolic link key like so:

```
HKEY hKey;
::RegOpenKeyEx(HKEY_CURRENT_USER, L"DesktopColors",
    REG_OPTION_OPEN_LINK, DELETE, &hKey);
::NtDeleteKey(hKey);
::RegCloseKey(hKey);
```

Lastly, the `RegCreateKeyEx` cannot open an existing link - it can only create one. This is contrast to "normal" keys that can be opened or created by `RegCreateKeyEx`. Opening a link must be done with `RegOpenKeyEx` with the flag `REG_OPTION_OPEN_LINK` in the options parameter.

Enumerating Keys and Values

Tools such as *RegEdit.exe* need to enumerate subkeys under a certain key or the values set on that key. Enumerating the subkeys is done with `RegEnumKeyEx`:

```
LSTATUS RegEnumKeyEx(
    _In_ HKEY hKey,
    _In_ DWORD dwIndex,
    _Out_ LPTSTR lpName,
    _Inout_ LPDWORD lpcchName,
    _Reserved_ LPDWORD lpReserved,
    _Out_ LPTSTR lpClass,
    _Inout_opt_ LPDWORD lpcchClass,
    _Out_opt_ PFILETIME lpftLastWriteTime);
```

The `hKey` handle must be opened with the `KEY_ENUMERATE_SUB_KEYS` access mask for the call to work. The way enumeration is performed is by specifying an index of zero for `dwIndex` and incrementing it in a loop, until the call returns `ERROR_NO_MORE_ITEMS`, indicating there are no more keys. The results returned always include the key's name (`lpName`), which must be accompanied by its size (`lpcchName`, set on input to the maximum number of characters the buffer can store including the `NULL` terminator, and changed by the function on output to the actual number of characters written excluding the terminating `NULL`). The name is the key's simple name, not the absolute name from the hive root. If the `lpName` buffer is not large enough to contain the key name, the function fails with `ERROR_MORE_DATA` and nothing is written to the `lpName` buffer.

The maximum key name length is 255 charachters.

Two other optional pieces of information may be returned: the class name (an optional user-defined value that is rarely used) and the last time the key was modified.

The following example, taken from the *DumpKey.exe* application (part of this chapter's code samples) shows how to enumerate keys:

```
#include <atltime.h>

void DumpKey(HKEY hKey, bool dumpKeys, bool dumpValues,
    bool recurse) {
    FILETIME modified;
//...
    if (dumpKeys) {
        printf("Keys:\n");
        WCHAR name[256];
        for (DWORD i = 0; ; i++) {
            DWORD cname = _countof(name);
            auto error = ::RegEnumKeyEx(hKey, i, name, &cname,
                nullptr, nullptr, nullptr, &modified);
            if(error == ERROR_NO_MORE_ITEMS)
```

```
                // enumeration complete
                break;

            // cannot happen, as the key name buffer is large enough
            if (error == ERROR_MORE_DATA) {
                printf(" (Key name too long)\n");
                continue;
            }

            if (error == ERROR_SUCCESS)
                printf(" %-50ws Modified: %ws\n", name,
                    (PCWSTR)CTime(modified).Format(L"%c"));

            if (recurse) {
                HKEY hSubKey;
                if (ERROR_SUCCESS == ::RegOpenKeyEx(hKey, name, 0,
                    KEY_READ, &hSubKey)) {
                    printf("--------\n");
                    printf("Subkey: %ws\n", name);
                    DumpKey(hSubKey, dumpKeys, dumpValues, recurse);
                    ::RegCloseKey(hSubKey);
                }
            }
        }
    }
}
```

Enumerating values within a given key is done in a similar manner, with `RegEnumValue`:

```
LSTATUS RegEnumValue(
    _In_ HKEY hKey,
    _In_ DWORD dwIndex,
    _Out_ LPTSTR lpValueName,
    _Inout_ LPDWORD lpcchValueName,
    _Reserved_ LPDWORD lpReserved,
    _Out_opt_ LPDWORD lpType,
    _Out_opt_ LPBYTE lpData,
    _Inout_opt_ LPDWORD lpcbData);
```

The function works in a similar way to `RegEnumKeyEx`, where `dwIndex` should start at zero and incremeneted in a loop until the function returns `ERROR_NO_MORE_ITEMS`. `hKey` must have the `KEY_QUERY_VALUE` access mask (also part of `KEY_READ`), otherwise the function returns `ERROR_ACCESS_DENIED`. The only mandatory return result is the value's name, specified by the `lpValueName` buffer and

its size (`lpcchValueName`). The same rules apply here as with `RegEnumKeyEx`: if the value name buffer is not large enough, nothing is returned in the name. This is trickier than the case for `RegEnumKeyEx`, because the maximum length of a value's name is 16383 characters. The simplest way to resolve this is to allocate a buffer 16384 characters in size (adding the `NULL` terminator), but this could be considered inefficient. An alternative way to handle this is to call `RegQueryInfoKey` before enumeration begins, that can return the maximum value name's length within the given key:

```
LSTATUS RegQueryInfoKey(
    _In_ HKEY hKey,
    _Out_opt_ LPWSTR lpClass,
    _Inout_opt_ LPDWORD lpcchClass,
    _Reserved_ LPDWORD lpReserved,
    _Out_opt_ LPDWORD lpcSubKeys,           // # if subkeys
    _Out_opt_ LPDWORD lpcbMaxSubKeyLen,     // max subkey length
    _Out_opt_ LPDWORD lpcbMaxClassLen,
    _Out_opt_ LPDWORD lpcValues,            // # values
    _Out_opt_ LPDWORD lpcbMaxValueNameLen,  // max value name length
    _Out_opt_ LPDWORD lpcbMaxValueLen,      // max value size
    _Out_opt_ LPDWORD lpcbSecurityDescriptor,
    _Out_opt_ PFILETIME lpftLastWriteTime);
```

This function contains too many parameters for my taste, it would have been better to set all these values in a structure. Nevertheless, it's fairly easy to use. Most of the parameters are optional, allowing the caller to retrieve only the information it cares about. The maximum value name length is provided, allowing the caller to allocate a large enough buffer for any value's name in the enumeration, which is most likely smaller than 16384 characters.

Back to `RegEnumValue` - the function optionally returns the value's type (`lpType`) and the value itself (`lpData`). If the value is needed (`lpData` is not `NULL`), the buffer for the value must be large enough to contain the entire value, otherwise the function fails with `ERROR_MORE_DATA` and nothing is written to the buffer.

The following example (taken from the *DumpKey* sample), enumerates values and displays the value's name and value (for the most common types):

```
void DumpKey(HKEY hKey, bool dumpKeys, bool dumpValues,
    bool recurse) {
    DWORD nsubkeys, nvalues;
    DWORD maxValueSize;
    DWORD maxValueNameLen;
    FILETIME modified;

    if (ERROR_SUCCESS != ::RegQueryInfoKey(hKey, nullptr, nullptr,
        nullptr, &nsubkeys, nullptr, nullptr, &nvalues,
        &maxValueNameLen, &maxValueSize, nullptr, &modified))
```

```
        return;

    printf("Subkeys: %u Values: %u\n", nsubkeys, nvalues);

    if (dumpValues) {
        DWORD type;
        auto value = std::make_unique<BYTE[]>(maxValueSize);
        auto name = std::make_unique<WCHAR[]>(maxValueNameLen + 1);

        printf("values:\n");
        for (DWORD i = 0; ; i++) {
            DWORD cname = maxValueNameLen + 1;
            DWORD size = maxValueSize;
            auto error = ::RegEnumValue(hKey, i, name.get(),
                &cname, nullptr, &type, value.get(), &size);
            if (error == ERROR_NO_MORE_ITEMS)
                break;

            auto display = GetValueAsString(value.get(),
                min(64, size), type);
            printf(" %-30ws %-12ws (%5u B) %ws\n", name.get(),
                (PCWSTR)display.first, size, (PCWSTR)display.second);
        }
    }
//...
```

The `GetValueAsString` helper function returns a `std::pair<CString, CString>` with the type and value as text for the most common types:

```
std::pair<CString, CString>
GetValueAsString(const BYTE* data, DWORD size, DWORD type) {
    CString value, stype;
    switch (type) {
        case REG_DWORD:
            stype = L"REG_DWORD";
            value.Format(L"%u (0x%X)",
                *(DWORD*)data, *(DWORD*)data);
            break;

        case REG_QWORD:
            stype = L"REG_QWORD";
            value.Format(L"%llu (0x%llX)",
                *(DWORD64*)data, *(DWORD64*)data);
```

```
            break;

        case REG_SZ:
            stype = L"REG_SZ";
            value = (PCWSTR)data;
            break;

        case REG_EXPAND_SZ:
            stype = L"REG_EXPAND_SZ";
            value = (PCWSTR)data;
            break;

        case REG_BINARY:
            stype = L"REG_BINARY";
            for (DWORD i = 0; i < size; i++)
                value.Format(L"%s%02X ", value, data[i]);
            break;

        default:
            stype.Format(L"%u", type);
            value = L"(Unsupported)";
            break;
    }

    return { stype, value };
}
```

Here is a truncated output from a call to `DumpKey` with the key *HKEY_CURRENT_USER\Control Panel* with all arguments set to `true`:

```
Subkeys: 15 Values: 1
values:
 SettingsExtensionAppSnapshot   REG_BINARY    (     8 B)
    00 00 00 00 00 00 00 00
Keys:
 Accessibility                  Modified: Tue Mar 10 12:47:14 2020
--------
Subkey: Accessibility
Subkeys: 13 Values: 4
values:
 MessageDuration                REG_DWORD     (     4 B) 5 (0x5)
 MinimumHitRadius               REG_DWORD     (     4 B) 0 (0x0)
 Sound on Activation            REG_DWORD     (     4 B) 1 (0x1)
```

```
 Warning Sounds                 REG_DWORD    (    4 B) 1 (0x1)
Keys:
 AudioDescription               Modified: Tue Mar 10 12:47:14 2020
--------
Subkey: AudioDescription
Subkeys: 0 Values: 2
values:
 On                             REG_SZ       (    4 B) 0
 Locale                         REG_SZ       (    4 B) 0
Keys:
 Blind Access                   Modified: Tue Mar 10 12:42:53 2020
--------
Subkey: Blind Access
Subkeys: 0 Values: 1
values:
 On                             REG_SZ       (    4 B) 0
Keys:
 HighContrast                   Modified: Tue Mar 10 12:47:14 2020
--------
Subkey: HighContrast
Subkeys: 0 Values: 3
values:
 Flags                          REG_SZ       (    8 B) 126
 High Contrast Scheme           REG_SZ       (    2 B)
 Previous High Contrast Scheme MUI Value REG_SZ       (    2 B)
Keys:
 Keyboard Preference            Modified: Tue Mar 10 12:42:53 2020
--------
Subkey: Keyboard Preference
Subkeys: 0 Values: 1
values:
 On                             REG_SZ       (    4 B) 0
Keys:
 Keyboard Response              Modified: Tue Mar 10 12:42:53 2020
--------
Subkey: Keyboard Response
Subkeys: 0 Values: 9
values:
 AutoRepeatDelay                REG_SZ       (   10 B) 1000
 AutoRepeatRate                 REG_SZ       (    8 B) 500
 BounceTime                     REG_SZ       (    4 B) 0
 DelayBeforeAcceptance          REG_SZ       (   10 B) 1000
 Flags                          REG_SZ       (    8 B) 126
```

```
 Last BounceKey Setting          REG_DWORD    (    4 B) 0 (0x0)
 Last Valid Delay                REG_DWORD    (    4 B) 0 (0x0)
 Last Valid Repeat               REG_DWORD    (    4 B) 0 (0x0)
 Last Valid Wait                 REG_DWORD    (    4 B) 1000 (0x3E8)
Keys:
 MouseKeys                       Modified: Tue Mar 10 12:42:53 2020
```

Registry Notifications

Some applications need to know when certain changes happen in the Registry, and update their behavior by reading again certain values of interest. The Registry API provides the `RegNotifyChangeKeyValue` for exactly this purpose:

```
LSTATUS RegNotifyChangeKeyValue(
    _In_ HKEY hKey,
    _In_ BOOL bWatchSubtree,
    _In_ DWORD dwNotifyFilter,
    _In_opt_ HANDLE hEvent,
    _In_ BOOL fAsynchronous);
```

`hKey` is the root key to watch for. It can be obtained by a normal call to `RegCreateKeyEx` or `RegOpenKeyEx` by specifying the `REG_NOTIFY` access mask, or one of the 5 main predefined keys can be used. `bWatchSubtree` indicates whether the specified key is only one watched ('FALSE') or that the entire tree of keys under `hKey` is watched for changes (`TRUE`).

The `dwNotifyFilter` indicates which operations should trigger a notification. Any combination of the flags shown in table 17-4 is valid.

Table 17-4: Flags to **`RegNotifyChangeKeyValue`**

Flag	Description
`REG_NOTIFY_CHANGE_NAME` (1)	subkey is added or deleted
`REG_NOTIFY_CHANGE_ATTRIBUTES` (2)	changes to any attribute of a key
`REG_NOTIFY_CHANGE_LAST_SET` (4)	change in modification time, indicating a value was added, changed or deleted
`REG_NOTIFY_CHANGE_SECURITY` (8)	change in the security descriptor of a key
`REG_NOTIFY_THREAD_AGNOSTIC` (0x10000000)	(Win 8+) the notification registration is not tied to the calling thread (see text for more details)

The `hEvent` parameter is an optional handle to an event kernel object, that becomes signaled when a notification arrives. This is required if the last argument (`fAsynchronous`) is set to `TRUE`. If `fAsynchronous` is FALSE, the call does not return until a change is detected. If `fAsynchronous` is `TRUE`, the call returns

immediately, and the event must be waited on to get notifications. The flag `REG_NOTIFY_THREAD_AGNOSTIC` indicates the calling thread is not associated with the registration, so that any thread can wait on the event handle. If this flag is not specified and the calling thread terminates, the registration is cancelled.

Using `RegNotifyChangeKeyValue` is fairly easy. Its main difficiency is the fact it does not specify exactly what change had occurred, and does not provide additional information as to the key and/or value where the change occurred. This makes it suitable to simple cases where a single key monitoring is needed (non-recursive), so when a change is detected, it's not too expensive to examining the changes in the key.

The *RegWatch* sample application shows how to use `RegNotifyChangeKeyValue` in synchronous mode. The interesting part is shown here:

```
// root is one of the standard hive keys
// path is the subkey (can be NULL)

HKEY hKey;
auto error = ::RegOpenKeyEx(root, path, 0, KEY_NOTIFY, &hKey);
if (error != ERROR_SUCCESS) {
    printf("Failed to open key (%u)\n", error);
    return 1;
}

// watch for adding/modifying keys/values
DWORD notifyFlags = REG_NOTIFY_CHANGE_NAME
    | REG_NOTIFY_CHANGE_LAST_SET;

printf("Watching...\n");
while (ERROR_SUCCESS == ::RegNotifyChangeKeyValue(hKey,
    recurse, notifyFlags, nullptr, FALSE)) {
    // no further info
    printf("Changed occurred.\n");
}
::RegCloseKey(hKey);
```

The alternative way to get more details on changes in the Registry is to use *Event Tracing For Windows* (ETW). This is the same mechanism used in chapters 7 and 9 for detecting DLL loads. The ETW kernel provider provides a list of Registry-related notifications that can be processed.

The *RegWatch2* sample application uses ETW to shows Registry activity. There is no built-in filter for getting notifications from certain keys, and it's up to the consumer to filter notifications it does not care about. The example shown does not filter anything, and shows the key name related to the operation (if any).

The class `TraceManager` used in chapters 7 and 9 has been copied as is. The only change is in the `Start` method (*TraceManager.cpp*) that changes the event flags to indicate Registry events (rather than image events in the original code):

```
_properties->EnableFlags = EVENT_TRACE_FLAG_REGISTRY;
```

The `EventParser` class (*EventParser.h/cpp*) has been copied as is (the only changes are the addition of header files because *RegWatch2* does not use precompiled headers). Here is the `main` function:

```
TraceManager* g_pMgr;
HANDLE g_hEvent;

int main() {
    TraceManager mgr;
    if (!mgr.Start(OnEvent)) {
        printf("Failed to start trace. Are you running elevated?\n");
        return 1;
    }

    g_pMgr = &mgr;

    g_hEvent = ::CreateEvent(nullptr, FALSE, FALSE, nullptr);

    ::SetConsoleCtrlHandler([](auto type) {
        if (type == CTRL_C_EVENT) {
            g_pMgr->Stop();
            ::SetEvent(g_hEvent);
            return TRUE;
        }
        return FALSE;
        }, TRUE);

    ::WaitForSingleObject(g_hEvent, INFINITE);
    ::CloseHandle(g_hEvent);

    return 0;
}
```

The `main` function creates a `TraceManager` object and calls `Start` (remember using ETW in this way requires admin rights). Stopping the session is done with a Ctrl+C key combination. The call to `SetConsoleCtrlHandler` is used to be notified when such key combination is detected. Unfortunately, the function pointer to `SetConsoleCtrlHandler` does not provide a way to pass a context argument, which is why the pointer to the `TraceManager` is also stored in a global variable.

In addition, an event object is created and held in a global variable to indicate `Stop` has been called, so that the wait is satisfied and the program can exit.

Each notification is sent to the `OnEvent` function, passed in the call to `TraceManager::Start`. Here is `OnEvent`:

```
void OnEvent(PEVENT_RECORD rec) {
    EventParser parser(rec);
    auto ts = parser.GetEventHeader().TimeStamp.QuadPart;
    printf("Time: %ws PID: %u: ",
        (PCWSTR)CTime(*(FILETIME*)&ts).Format(L"%c"),
        parser.GetProcessId());
    switch (parser.GetEventHeader().EventDescriptor.Opcode) {
        case EVENT_TRACE_TYPE_REGCREATE:
            printf("Create key"); break;
        case EVENT_TRACE_TYPE_REGOPEN:
            printf("Open key"); break;
        case EVENT_TRACE_TYPE_REGDELETE:
            printf("Delete key"); break;
        case EVENT_TRACE_TYPE_REGQUERY:
            printf("Query key"); break;
        case EVENT_TRACE_TYPE_REGSETVALUE:
            printf("Set value"); break;
        case EVENT_TRACE_TYPE_REGDELETEVALUE:
            printf("Delete value"); break;
        case EVENT_TRACE_TYPE_REGQUERYVALUE:
            printf("Query value"); break;
        case EVENT_TRACE_TYPE_REGENUMERATEKEY:
            printf("Enum key"); break;
        case EVENT_TRACE_TYPE_REGENUMERATEVALUEKEY:
            printf("Enum values"); break;
        case EVENT_TRACE_TYPE_REGSETINFORMATION:
            printf("Set key info"); break;
        case EVENT_TRACE_TYPE_REGCLOSE:
            printf("Close key"); break;
        default:
            printf("(Other)"); break;
    }
    auto prop = parser.GetProperty(L"KeyName");
    if (prop) {
        printf(" %ws", prop->GetUnicodeString());
    }
    printf("\n");
}
```

Only some of the possible notifications is specifically captured, all others are displayed as "(other)". if a key name property exists, it's displayed as well. If you run the application, you'll appreciate the sheer number of Registry operations that occur at any given time.

> There are other pieces of information for each event. Call `EventParser::GetProperties()` to get all the custom properties for an event record.

Both of the notification options we've seen do not allow any interception of changes. Such a powerful capability is only available when using kernel APIs. My book "Windows Kernel Programming" shows how to achieve this.

Transactional Registry

In chapter 11 (part 1), we've seen that file operations can be part of a transaction by using functions such as `CreateFileTransacted`, associating it with an open transaction. Such a transaction can be created with `CreateTransaction` (refer to chapter 11 for the details). Registry operations can be transacted as well, operating as an atomic set of operations, which may also be combined with file transacted operations.

Performing Registry operations as part of a transaction must be done by using a key created with `RegCreateKeyTransacted` or opened with `RegOpenKeyTransacted`:

```
LSTATUS RegCreateKeyTransacted (
    _In_ HKEY hKey,
    _In_ LPCTSTR lpSubKey,
    _Reserved_ DWORD Reserved,
    _In_opt_ LPTSTR lpClass,
    _In_ DWORD dwOptions,
    _In_ REGSAM samDesired,
    _In_opt_ CONST LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    _Out_ PHKEY phkResult,
    _Out_opt_ LPDWORD lpdwDisposition,
    _In_        HANDLE hTransaction,
    _Reserved_ PVOID  pExtendedParemeter);
LSTATUS RegOpenKeyTransacted (
    _In_ HKEY hKey,
    _In_opt_ LPCTSTR lpSubKey,
    _In_opt_ DWORD ulOptions,
    _In_ REGSAM samDesired,
    _Out_ PHKEY phkResult,
    _In_        HANDLE hTransaction,
    _Reserved_ PVOID  pExtendedParemeter);
```

These extended functions have the same parameters as the non-transactional versions, except the last two parameters that indicate the transaction to work against (`hTransaction`). All subsequent operations

using the returned `hKey` are part of the transaction, and will eventually all succeed, or all will fail as one atomic operation.

> See the discussion on transactions in chapter 11.

Registry and Impersonation

The `HKEY_CURRENT_USER` hive normally refers to the user's process token. If a thread is impersonating (see chapter 16 for more on impersonation), it might be more appropriate to get the impersonation token's user-related hives. Several functions are available to make this easier.

The `RegOpenCurrentUser` API opens a key under `HKEY_CURRENT_USER` of the impersonation token's user:

```
LSTATUS RegOpenCurrentUser(
    _In_ REGSAM samDesired,     // access mask
    _Out_ PHKEY phkResult);     // reulting key
```

if the current thread is not impersonating, the function is equivalent to opening `HKEY_CURRENT_USER` for the current process' token user.

A similar function exists for `HKEY_CLASSES_ROOT`:

```
LSTATUS RegOpenUserClassesRoot(
    _In_ HANDLE hToken,
    _Reserved_ DWORD dwOptions,     // must be zero
    _In_ REGSAM samDesired,
    _Out_ PHKEY phkResult);
```

This function requires the token to use, which makes it more flexible, not requiring impersonation. `HKEY_CLASSES_ROOT` is naturally more complex, as it merges information from the user's hive (`HKCU\SOFTWARE\Classes`) and the machine (`HKLM\SOFTWARE\Classes`).

Remote Registry

A Registry key can be opened or created on a remote machine by first calling `RegConnectRegistry` to connect to the Registry on another machine. To make it work, the *Remote Registry* service must be running on the remote computer. By default, this service is configured to start manually, so it's unlikely to be running. There are other security restrictions when connecting to a remote Registry even if the service is running. Check the documentation for details. Here is `RegConnectRegistry`:

```
LSTATUS RegConnectRegistry (
    _In_opt_ LPCTSTR lpMachineName,
    _In_ HKEY hKey,
    _Out_ PHKEY phkResult);
```

`lpMachineName` is the machine to connect to, which must have the format *computername*. `hKey` must be one of the following predefined keys: `HKEY_USERS`, `HKEY_LOCAL_MACHINE` or `HKEY_PERFORMANCE_DATA`. The last parameter (`phkResult`) is the returned handle to use locally.

Given the new handle, normal Registry operations can be performed, including reading values, opening subkeys, writing values, etc., all subject to security checks. Once finished, the handle should be closed normally with `RegCloseKey`.

An extended version of `RegConnectRegistry` exists as well, `RegConnectRegistryEx`:

```
LSTATUS RegConnectRegistryEx (
    _In_opt_ LPCTSTR lpMachineName,
    _In_ HKEY hKey,
    _In_ ULONG Flags,
    _Out_ PHKEY phkResult);
```

It's identical to `RegConnectRegistry`, but allows specifying some flags. The only currently supported flag is `REG_SECURE_CONNECTION` (1), which indicates the caller wants to establish a secure connection to the remote Registry. This causes the RPC calls sent to the remote computer to be encrypted.

Miscellaneous Registry Functions

The Registry API includes other miscellaneous functions, some of which are described briefly in this section.

The `RegGetKeySecurity` and `RegSetKeySecurity` functions allow retrieving and manipulating the security descriptor of a given key. Although the more generic function `GetSecurityInfo` and `SetSecurityInfo` functions (described in chapter 16) can be used with registry keys, the specific ones are easier to use.

```
LSTATUS RegGetKeySecurity(
    _In_ HKEY hKey,
    _In_    SECURITY_INFORMATION SecurityInformation,
    _Out_   PSECURITY_DESCRIPTOR pSecurityDescriptor,
    _Inout_ LPDWORD lpcbSecurityDescriptor);
LSTATUS RegSetKeySecurity(
    _In_ HKEY hKey,
    _In_ SECURITY_INFORMATION SecurityInformation,
    _In_ PSECURITY_DESCRIPTOR pSecurityDescriptor);
```

A Registry key (and all its values and subkeys) can be saved to a file by calling `RegSaveKey`:

```
LSTATUS RegSaveKey (
    _In_ HKEY hKey,
    _In_ LPCTSTR lpFile,
    _In_opt_ CONST LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

These functions save all the information for `hKey` and its descendants to a file given by `lpFile`. `hKey` can be a key opened with the standard functions or one of the following predefined keys only: `HKEY_CLASSES_ROOT` or `HKEY_CURRENT_USER`. `lpFile` must not exist before the call, otherwise the call fails. The optional provided security attributes can provide the security descriptor on the new file; if `NULL`, a default security descriptor is used, typically inherited from the file's parent folder.

The format the data is saved in is called "standard format", supported since Windows 2000. This format is proprietary and generally undocumented. Specifically, this is **not** the .REG file format used by *RegEdit.exe* when the *Export* menu item is selected to back up a Registry key. The REG file format is specific to *RegEdit* and is not generally known to the Registry functions.

The current (latest) format for saving keys is available by calling the extended function `RegSaveKeyEx`:

```
LSTATUS RegSaveKeyEx(
    _In_ HKEY hKey,
    _In_ LPCTSTR lpFile,
    _In_opt_ CONST LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    _In_ DWORD Flags);
```

The function adds a `Flags` parameter, that must be one (and only one) of the following values:

- `REG_STANDARD_FORMAT` (1) - the original format, the same one used by `RegSaveKey`.
- `REG_LATEST_FORMAT` (2) - the latest (better) format.
- `REG_NO_COMPRESSION` (4) - saves the data uncompressed (as is). Only works on true hives (marked with a special icon in *RegExp*). The full hive list can be viewed within the Registry itself, at *HKLM\System\CurrentControlSet\Control\hivelist*.

One downside of `RegSaveKeyEx`, is that the predefined key `HKEY_CLASSES_ROOT` is not supported.

> Both `RegSaveKey` and `RegSaveKeyEx` require the caller to have the *SeBackupPrivilege* in its token. This privilege is normally given to members of the administrators group, but not to users with standard user rights.

With a saved file in hand, the information can be restored into the Registry with `RegRestoreKey`:

```
LSTATUS RegRestoreKey(
    _In_ HKEY hKey,
    _In_ LPCTSTR lpFile,
    _In_ DWORD dwFlags);
```

`hKey` specifies the key to restore information to. It can be any open key or one of the standard 5 hive keys. `lpFile` is the file where the data is stored. The restore operation preserves the name of root key identified by `hKey`, but replaces all other attributes of that key, and replaces all subkeys/values, as stored in the file.

`dwFlags` provides several options, from which only two are officially documented:

- `REG_FORCE_RESTORE` (8) - forces the restore operation even if there are open key handles to subkeys that will be overwritten.
- `REG_WHOLE_HIVE_VOLATILE` (1) - creates a new hive in memory only. In this case, `hKey` must be `HKEY_USERS` or `HKEY_LOCAL_MACHINE`. The hive is removed in the next system boot.

> Similarly to `RegSaveKey(Ex)`, the caller must have the *SeRestorePrivilege* privilege in its token, normally given to administrators.

An alternative to loading a hive with `RegRestoreKey` is to use `RegLoadAppKey`:

```
LSTATUS RegLoadAppKey(
    _In_ LPCTSTR lpFile,
    _Out_ PHKEY phkResult,
    _In_ REGSAM samDesired,
    _In_ DWORD dwOptions,
    _Reserved_ DWORD Reserved);
```

`RegLoadAppkey` loads the hive specified by `lpFile` into an invisible root that cannot be enumerated, and so is not a part of the standard Registry. If `lpFile` does not exist, it's created to holf the new application hive. The only way to access anything in the hive is through the root `phkResult` key returned on success. The advantage over `RegRestoreKey` is that the caller does not need the *SeRestorePrivilege* and so can be used by non-admin callers.

The only flag available in `dwOptions` (`REG_PROCESS_APPKEY`), if used, prevents other callers from loading the same hive file while the key handle is open.

> The `RegLoadAppKey` API creates an *application hive*, which is only accessible to the process creating it. The hive itself is stored in the real Registry under the key "\REGISTRY\A". You can do a simple search in *Process Explorer* for this name. You'll typically find UWP processes using application hives.

> The application hive is destroyed automatically when all the handles to keys in the hive are closed.

A somewhat similar operation to `RegRestoreKey` with `REG_WHOLE_HIVE_VOLATILE` is available with `RegLoadKey`, where a hive can be loaded from a file and stored as a new hive under either *HKEY_LOCAL_MACHINE* or *HKEY_USERS*. This is useful when looking at a Registry file offline, such as a file brought from another system. *RegEdit* provides this functionality by calling `RegLoadKey` in its menu option *File / Load Hive...*. You'll notice this option is only available when the selected key in the tree view is either *HKEY_LOCAL_MACHINE* or *HKEY_USERS* (figure 17-6).

A more powerful Registry editor/viewer is my own *Registry Explorer*, available at https://github.com/zodiacon/RegExp.

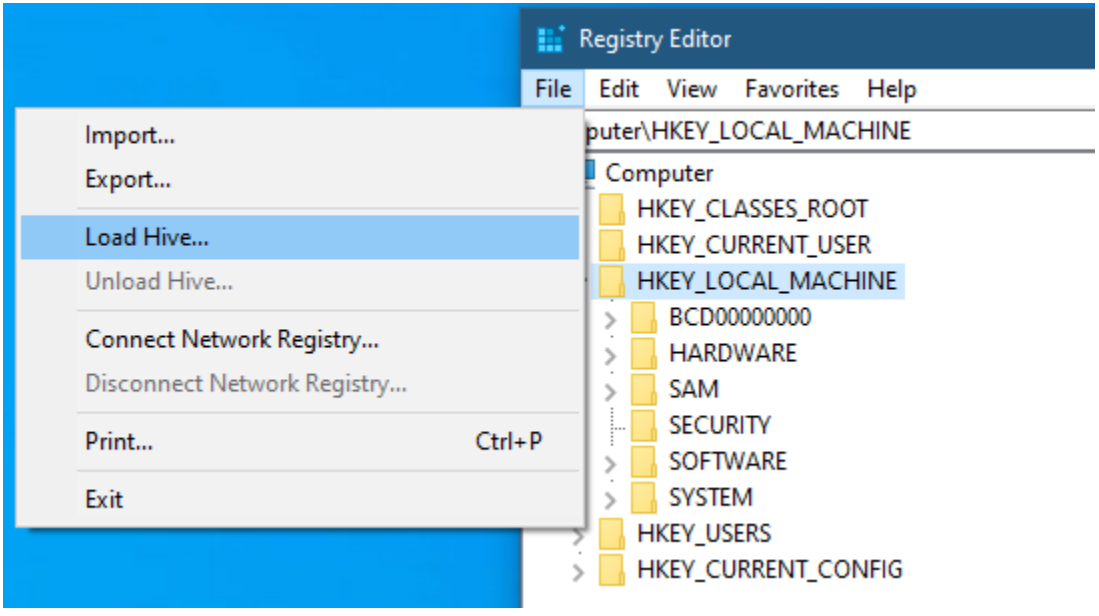


Figure 17-6: *Load Hive* option in *RegEdit*

Here is `RegLoadKey`'s definition:

```
LSTATUS RegLoadKey(
    _In_ HKEY hKey,
    _In_opt_ LPCTSTR lpSubKey,
    _In_ LPCTSTR lpFile);
```

`hKey` can be *HKEY_LOCAL_MACHINE* or *HKEY_USERS* (because these are true keys rather than various links). `lpSubKey` is the name of the subkey under which the hive is loaded from the file specified by `lpFile`. Any changes made to the loaded key (or any of its subkeys) is persisted eventually in the file.

Unloading the hive can be done with `RegUnloadKey`:

```
LSTATUS RegUnLoadKey(
    _In_ HKEY hKey,
    _In_opt_ LPCTSTR lpSubKey);
```

The parameters mirror their counterparts in `RegLoadKey`.

Sometimes it may be desirable to redirect one of the predefined keys to an alternative key. This feat is possible with `RegOverridePredefKey`:

```
LSTATUS RegOverridePredefKey (
    _In_ HKEY hKey,
    _In_opt_ HKEY hNewHKey);
```

`hKey` must be one of the standard predefined keys, and `hNewKey` must be a different key. Specify `NULL` for `hNewKey` to restore the overriden key to its default. The redirection works for the current process only.

Summary

The Registry is the primary database the Windows system uses, for system-wide and user-specific settings. Some parts of the Registry are especially important in the context of system security, kernel operation, and much more. In this chapter, we looked at the Registry's concepts and layout, as well as the most common API functions used to manipulate it.

Chapter 18: Pipes and Mailslots

Windows provides different *Inter-Process Communication* (IPC) mechanisms, many of which we met in earlier chapters. Kernel objects of all kinds (except perhaps I/O completion ports) are sharable by definition, so provide ways to transfer information between processes if so desired. In some cases, the information is very simple, such as thread synchronization states with mutexes, events, and semaphores. Others provide more in terms of "data" - most notably memory-mapped files (sections).

There are other mechanisms that are not using the standard kernel objects. For example, window messages can be used to send information from one process to another by sending or posting a message to a window created by another process. It's even possible to post a message directly to a thread with `PostThreadMessage`.

All the above mechanisms, as useful as they are, cannot span machines. They are bound to the local machine where they were created. Files for one, can be used across machines by sharing directories, but files are not "managed" in any special way, they are just storing data, and couldn't care less how that data gets there.

Two other mechanisms provided by Windows are pipes and mailslots, which can be used to transfer data between a client and server, sometimes in both directions. They can be used across machines as well. In this chapter, we'll examine the fundamentals of mailslots and pipes (both anonymous and named).

Pipes and mailslots are implemented internally as file systems.

There is another mechanism that can be used for client/server communication, possibly between machines - the *Component Object Model* (COM), which we'll examine in chapter 21. Yet another mechanism for inter-machine communication is Windows sockets, very similar to "classic" sockets used in other operating systems. Sockets are beyond the scope of this book.

In this chapter:

Mailslots

Mailslots are similar to standard real-world mailboxes. You can send a letter that would arrive to a mailbox based on the receiver's address, but the receiver cannot return a letter to the sender unless the receiver sends a letter of its own and knows the sender's address. In short, mailslots are one-way communication devices. Figure 18-1 shows a mailbox server that receives messages from clients.

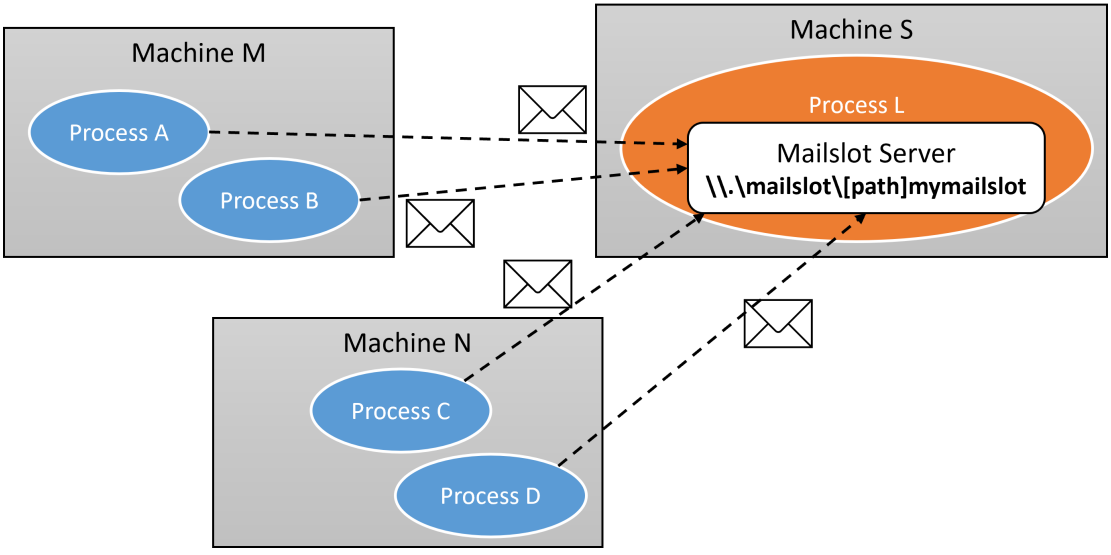


Figure 18-1: Mailslots

A mailslot server creates a mailslot by calling `CreateMailslot`:

```
HANDLE CreateMailslot(
    _In_     LPCTSTR lpName,
    _In_     DWORD nMaxMessageSize,
    _In_     DWORD lReadTimeout,
    _In_opt_ LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

The mailslot name must have this format: \\.*mailslot**[path]name* where *path* is an optional virtual path separated with backslashes. The full mailslot name must be unique, and the dot representing the local machine is required - you cannot create a mailslot server on another machine (but clients can connect from other machines).

`nMaxMessageSize` is the maximum message size (in bytes) that the mailslot accepts by default. Specify zero to indicate no specific limit. You can provide a higher limit if you wish when the mailslot is queried for messages with `GetMailslotInfo` we'll see soon. `lReadTimeout` indicates how long the caller is willing to wait when using `ReadFile` to read the next message (in milliseconds). It can be zero to indicate no wait, or `MAILSLOT_WAIT_FOREVER` (same as `INFINITE` or `(DWORD)-1`) to wait until a message arrives. Finally, `lpSecurityAttributes` allows configuring a custom security descriptor for the newly created

mailslot. Specifying `NULL` provides a default security descriptor (based on the default stored in the caller's access token).

The return value is a valid handle if successful, or `INVALID_HANDLE_VALUE` otherwise. The returned handle is a file object handle, because mailslots are implemented as file systems. You can convince yourself of that by examining a successful call handle in *Process Explorer* (figure 18-2), where the mailslot name is "loggerbox".

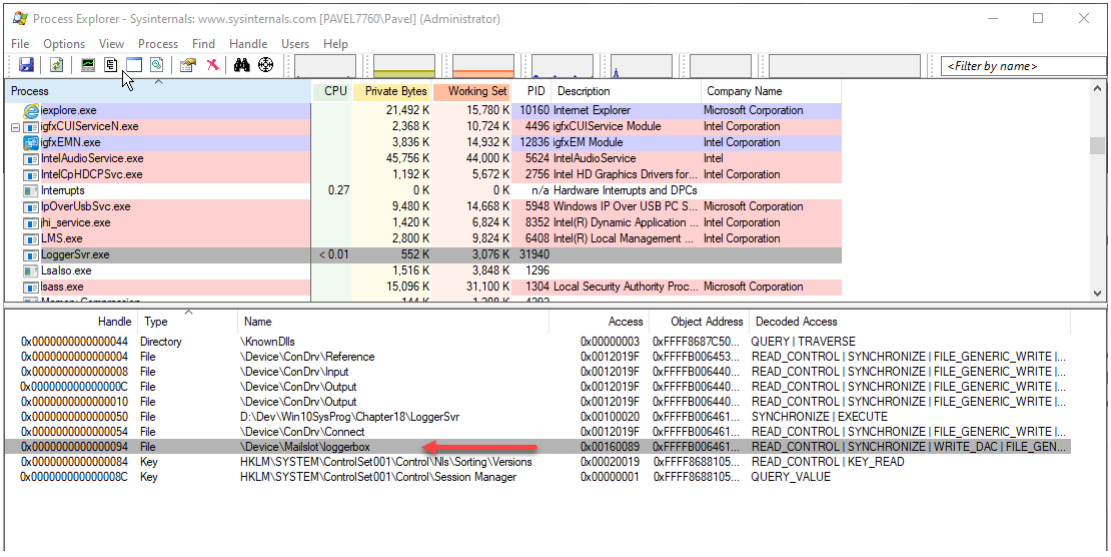


Figure 18-2: Mailslot handle in *Process Explorer*

The mailslot in figure 18-2 was obtained by the following code:

```
HANDLE hMailSlot = ::CreateMailslot(L"\\\\.\\mailslot\\loggerbox",
    1024, MAILSLOT_WAIT_FOREVER, nullptr);
```

With a mailslot handle, the server process reads messages and processes them. Querying about message availability and size is done with `GetMailslotInfo`:

```
BOOL GetMailslotInfo(
    _In_      HANDLE hMailslot,
    _Out_opt_ LPDWORD lpMaxMessageSize,
    _Out_opt_ LPDWORD lpNextSize,
    _Out_opt_ LPDWORD lpMessageCount,
    _Out_opt_ LPDWORD lpReadTimeout);
```

The only required argument is the mailslot handle. `lpMaxMessageSize` returns the maximum message size, which can be higher than the set with `CreateMailslot`. In most cases, this value is not interesting, thus specifying `NULL` is common. `lpNextSize` returns the size of the next message, if any. If no message

is waiting, `MAILSLOT_NO_MESSAGE` is returned. `lpMessageCount` returns the number of messages waiting to be retrieved. Finally, `lpReadTimeout` returns the maximum time a `ReadFile` call can wait before returning if no message is waiting. `lpNextSize` is the most useful parameter to specify, as it allows the code to prepare for retrieving the next message by (perhaps) allocating an appropriate buffer.

To actually read the next message, a standard `ReadFile` call can be used. The mailslot handle is an asynchronous one, so `ReadFile` can use any of the available Windows mechanisms to be notified when the result is available, or use a synchronous call by specifying `NULL` for the overlapped structure (last parameter to `ReadFile`). See chapter 9 in part 1 for more details on asynchronous I/O processing.

A typical mailslot server will wait for a message in a loop and read each one as they arrive until some exit condition. If all handles to the mailslot server (one by default) are closed, the mailslot file object is destroyed, as it is normally with any kernel object.

Given the mailslot created above, the following code is used to read messages from the mailslot, assuming these are `NULL`-terminated ANSI strings and displays them as they come in:

```
DWORD nextSize;
char message[1024];
DWORD read;
SYSTEMTIME st;

while (::GetMailslotInfo(hMailSlot, nullptr, &nextSize, nullptr,
    nullptr)) {
    if (nextSize == MAILSLOT_NO_MESSAGE) {
        ::Sleep(100);
        continue;
    }
    if (!::ReadFile(hMailSlot, message, nextSize, &read, nullptr))
        continue;

    ::GetLocalTime(&st);
    printf("%02d:%02d:%02d.%03d: %s\n",
        st.wHour, st.wMinute, st.wSecond, st.wMilliseconds, message);
}

::CloseHandle(hMailSlot);
```

> This code is in the *LoggerSvr* project, part of this chapter's samples.

If you later decide to change the mailslot read time out, call `SetMaislotInfo`:

```
BOOL SetMailslotInfo(
    _In_ HANDLE hMailslot,
    _In_ DWORD lReadTimeout);
```

Mailslot Clients

A mailslot client connects to a mailslot by calling the normal `CreateFile` or `CreateFile2` APIs, specifying the "filename" to be the full mailslot name as created by the mailslot server. If the mailslot cannot be found, the standard `ERROR_FILE_NOT_FOUND` (2) is returned. Other errors are possible as well, such as `ERROR_ACCESS_DENIED` (5).

The following example shows a client that reads input text from the keyboard, and writes it to the mailslot with the normal `WriteFile` API. The client in this example is on the same machine as the mailslot:

```
HANDLE hFile = ::CreateFile(L"\\\\.\\mailslot\\loggerbox",
    GENERIC_WRITE, FILE_SHARE_WRITE, nullptr, OPEN_EXISTING,
    0, nullptr);
if (hFile == INVALID_HANDLE_VALUE)
    return Error("Failed to open mailslot");    // handle error

DWORD written;
char text[256];
do {
    printf("Enter message (q to quit): ");
    gets_s(text);
    if (_stricmp(text, "q") == 0)
        break;
} while (::WriteFile(hFile, text,
    (DWORD)strlen(text) + 1, &written, nullptr));

::CloseHandle(hFile);
```

> This code is in the *TestLog* project, part of the samples for this chapter.

if the client is on another machine, the code to connect to the mailslot is almost identical. The only change is the machine name that must be specified so that the mailslot can be located:

```
HANDLE hFile = ::CreateFile(L"\\\\myserver\\mailslot\\loggerbox",
    GENERIC_WRITE, FILE_SHARE_WRITE, nullptr, OPEN_EXISTING,
    0, nullptr);
```

Mailslots are fairly simple creatures, and easy to use. A logger service, as demonstrated by the above code snippets is a good example - all the information logged has a single time line, a central location for storing the information. In many cases such a mailslot would be hosted in a service (see chapter 19). The log data can be written to anywhere of course.

As with any communication, some protocol is usually defined. The above example is a simplistic one, that assumes everything is just a simple string. In realistic scenarios, messages might be data structures that contain some common header to distinguish between different message types. Then the task of the mailbos server is to parse messages and process them appropriately.

You can use more than one thread to read and process messages from a mailslot. Any mechanism we've seen in part 1, such as using explicit threads or the thread pool would work.

Multi-Mailslot Communication

Clients can communicate with multiple mailslots with a certain name at the same time by using any of the following formats for the "filename" in the call to `CreateFile`:

- *domain**mailslot**name* - receives a handle to all mailslots named *name* in the domain *domain*.
- **mailslot**name* - receives a handle to all mailslots named *name* in the primary domain.

If any of these options are used, the client can send messages that are no more than 424 bytes at a time (single call to `WriteFile`). A larger message results in an error. The message itself is sent to all the mailslots as described by the path.

Anonymous Pipes

As the name suggests, a pipe is a two-ended communication mechanism. There are two types of pipes - anonymous and named. Anonymous pipes are easier to use, but they are limited in several ways, most notably the fact that they cannot be used to communicate between processes on different machines.

We'll start with anonymous pipes and deal with named pipes in the next section.

An anonymous pipe is created by calling `CreatePipe`:

```
BOOL CreatePipe(
    _Out_ PHANDLE hReadPipe,
    _Out_ PHANDLE hWritePipe,
    _In_opt_ LPSECURITY_ATTRIBUTES lpPipeAttributes,
    _In_ DWORD nSize);
```

`CreatePipe` returns on success two handles to a newly created anonymous pipe in `hReadPipe` and `hWritePipe`. Both handles are local to the current process, just like any other created handle. `lpPipeAttributes` can specify an optional security descriptor to protect the pipe object. If `NULL` is specified, the pipe is not protected at all. A more common case is using the `SECURITY_ATTRIBUTES` to indicate the handles should be marked inheritable, as sharing the pipe with another process is commonly done via handle inheritance, as will soon be demonstrated. Finally, `nSize` specifies the internal buffer size that should be used by the pipe driver, but it is merely a suggestion. A value of zero is usually specified, letting the pipe driver to handle this decision. The result of a successful call to `CreatePipe` is depicted in figure 18-3.

Figure 18-3: Anonymous Pipe

The write handle can only be used with `WriteFile` to write to one end of the pipe, whereas the read handle can only be used with `ReadFile` to read the written data. Both handles are for file objects (just like mailslot handles), but the underlying driver is the pipes driver (figure 18-4).

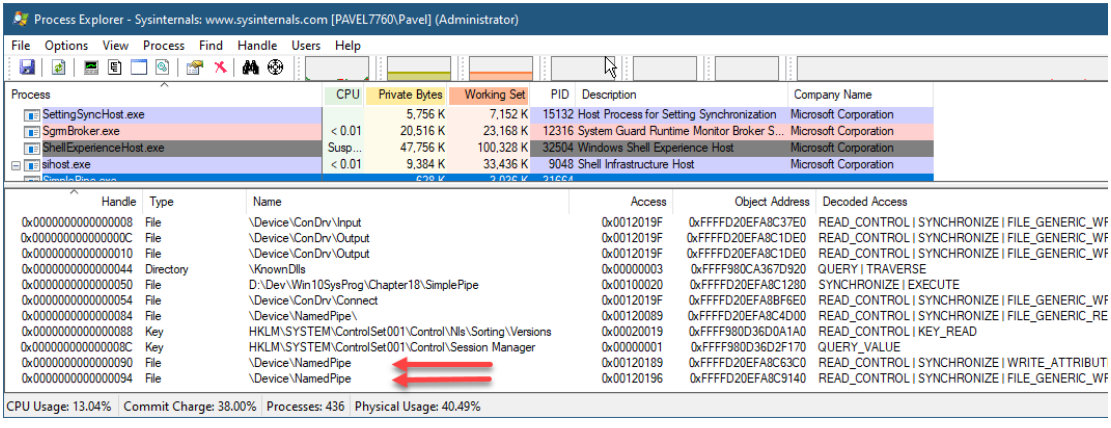


Figure 18-4: Pipe handles in *Process Explorer*

Anonymous pipes are unidirectional (sometimes called *simplex*) - one side calls `WriteFile` with the write handle and the other side uses `ReadFile` with the read handle. If bi-directional (*duplex*) communication is needed, a second pipe can be created, with the read handle "handed out" to the side holding the write handle and vice versa. A more complete option is to use named pipes for duplex communication, as we'll see in the next section.

The read and write handles can be shared with other threads in the process, if that makes sense for the application. However, the more common case is sharing one end of the anonymous pipe with another process, so that one side writes and the other side reads. Figure 18-5 shows such an example, where process A holds the read handle and process B holds the write handle.

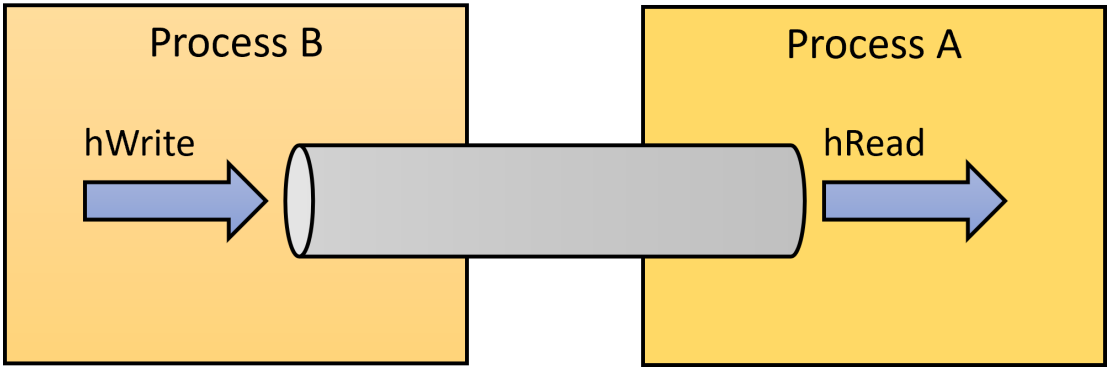


Figure 18-5: Anonymous pipe shared between processes

The question is how to pass one of the handles to another process. There is nothing special about this question, as it's the normal case of how to share a kernel object between processes, as discussed in chapter 2 (part 1). In that chapter, three ways of sharing kernel objects were discussed:

- Sharing by name.
- Sharing by inheriting a handle.
- Sharing by duplicating a handle.

The first option (sharing by name) is not available for anonymous pipe handles, as the underlying objects are unnamed. The remaining options are valid. Using handle inheritance is usually more convenient, and this is the approach we'll take in the following example.

The Command Redirect Application

Processes that write (and/or read) to the console use handles to the console input and/or output. This is true if the application uses `WriteFile` or `WriteConsole` and similar APIs, or uses C/C++ runtime functions such as `printf` and `scanf` - console handles are used internally. We can replace the default handles created at process startup time, effectively "redirecting" input and/or output.

The *SimplePipe* console application from this chapter's samples shows one way this can be achieved. The application created a *cmd.exe* process and redirects its output to an anonymous pipe. The *cmd.exe* process writes to its console handle, but that one has been replaced with a write pipe handle.

The `main` function starts by creating the anonymous pipe:

```
int main() {
    HANDLE hRead, hWrite;
    if (!::CreatePipe(&hRead, &hWrite, nullptr, 0))
        return Error("Failed to create pipe");  // handle the error
```

We want to share the write handle with a new process (soon to be created), so we need to mark it as inheritable. We can do that by setting the `bInheritHandle` field to `TRUE` in the passed-in `SECURITY_ATTRIBUTES` to `CreatePipe` (that would also set the read handle's inheritance flag), or just use `SetHandleInformation`:

```
::SetHandleInformation(hWrite, HANDLE_FLAG_INHERIT, HANDLE_FLAG_INHERIT);
```

Now we're ready to create the *cmd.exe* process while enabling handle inheritance. Setting the inheritted handle as the standard output handle for the new process is achieved by setting the relevant field in the `STARTUPINFO` structure (refer to chapter 3 in part 1 if you need a refresher):

```
PROCESS_INFORMATION pi;
STARTUPINFO si = { sizeof(si) };
si.hStdOutput = hWrite;
si.dwFlags = STARTF_USESTDHANDLES;

WCHAR name[] = L"cmd.exe";
if (!::CreateProcess(nullptr, name, nullptr, nullptr, TRUE,
    CREATE_NEW_CONSOLE, nullptr, nullptr, &si, &pi))
    return Error("Failed to create cmd process");
```

The `CREATE_NEW_CONSOLE` flag is important because otherwise the *cmd.exe* process will share the same console that *SimplePipe* has (being a console app itself).

At this point, the returned thread handle and the local pipe write handle are not needed anymore (the write handle has been duplicated into the new process):

```
::CloseHandle(pi.hThread);  // thread handle not needed
::CloseHandle(hWrite);      // write pipe handle not needed
```

All that's left to do is to read data from the read end of the pipe and do something with the data. In this case, just print it into the *SimplePipe* console:

```
char text[512];
DWORD read;
while (::WaitForSingleObject(pi.hProcess, 0) == WAIT_TIMEOUT) {
    if (!::ReadFile(hRead, text, sizeof(text) - 1, &read, nullptr))
        break;

    // NULL-terminate the string
    text[read] = 0;
    printf("%s", text);
}

::CloseHandle(hRead);
::CloseHandle(pi.hProcess);
```

The code loops around until the *cmd.exe* process it terminated. It reads whatever data is available and displays it. Figure 18-6 shows a few examples from the *cmd.exe* process (right) and the resulting output in the *SimplePipe* console.

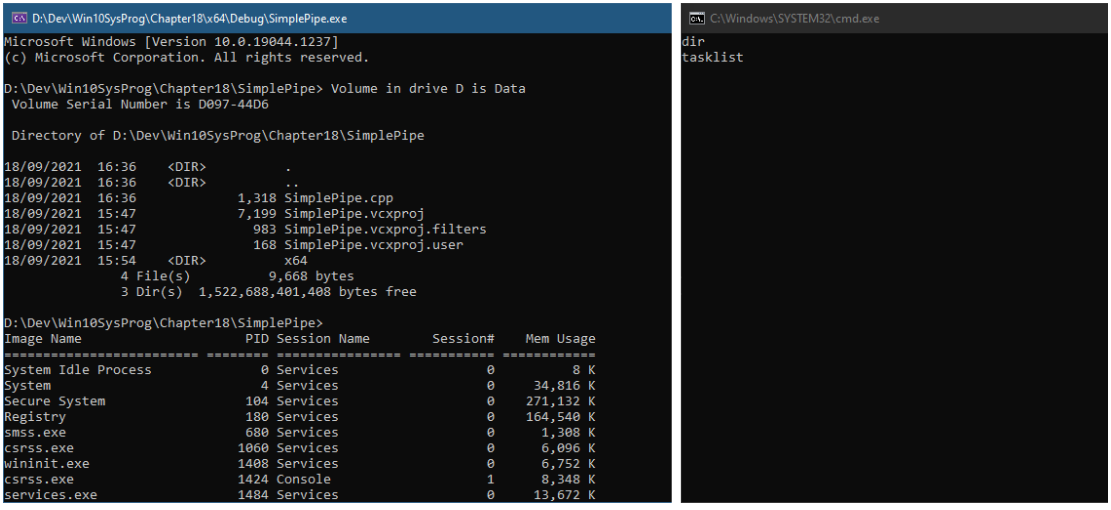


Figure 18-6: *SimplePipe* redirection in action

Create a GUI application that allows creating any process with a console, and displays all their output in a dedicated window or edit box.

Named Pipes

Named pipes are significantly more complex than anonymous pipes, but have several useful features beyond anonymous pipes:

- Client and server may be in different machines.
- Complete messages are supported, not just a stream of bytes.
- Multiple instances of the pipe may be created. Each pipe instance can communicate with its own client.
- Pipes support duplex communication in addition to only input or output.

In this section, we'll introduce the fundamentals of named pipes, but a comprehensive treatment is beyond the scope of this chapter.

A named pipe server uses `CreateNamedPipe` to create the first instance of a named pipe:

```
HANDLE CreateNamedPipe(
    _In_ LPCTSTR lpName,
    _In_ DWORD dwOpenMode,
    _In_ DWORD dwPipeMode,
    _In_ DWORD nMaxInstances,
    _In_ DWORD nOutBufferSize,
    _In_ DWORD nInBufferSize,
    _In_ DWORD nDefaultTimeOut,
    _In_opt_ LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

The parameters to `CreateNamedPipe` are as follows:

- `lpName` is the pipe's name, which must be in the format \\.*pipe\[path]pipename*. The named pipe must be on the local machine, there is no way to create a pipe server on a different machine. Clients can connect to the pipe with `CreateFile` from any machine on the network.
- `dwOpenMode` consists of several flags. One of the following three values must be specified, indicating the direction of the pipe: `PIPE_ACCESS_DUPLEX` (bi-directional), `PIPE_ACCESS_INBOUND` (from client to server), `PIPE_ACCESS_OUTBOUND` (from server to client). Additionally, `FILE_FLAG_OVERLAPPED` can be specified for asynchronous I/O. The `FILE_FLAG_WRITE_THROUGH` flag indicates that write operations across machines for byte-type pipes do not return until the entire buffer is transmitted across the network. If the `FILE_FLAG_FIRST_PIPE_INSTANCE` flag is used, only the first instance of the named pipe is created successfully. For subsequent calls using this flag, the creation fails. Finally, three security-related flags are supported, indicating the caller's ability to manipulate security attributes of the resulting pipe: `WRITE_DAC`, `WRITE_OWNER`, and `ACCESS_SYSTEM_SECURITY`. You can find details for these flags in the documentation or chapter 16.
- `dwPipeMode` is yet another flags parameter. The following pairs of flags are each mutually exclusive:
  - `PIPE_TYPE_BYTE` (default) vs. `PIPE_TYPE_MESSAGE`: a byte-type pipe does not distinguish between bytes written in multiple write operations, whereas a message-type pipe treats each write as a message unit. If `GetLastError` returns `ERROR_MORE_DATA` from a `ReadFile` call, it means a message has not been read completely.
  - `PIPE_READMODE_BYTE` (default) vs. `PIPE_READMODE_MESSAGE`: specify the pipe's read mode, either a stream of bytes or a stream of messages (only applicable to `PIPE_TYPE_MESSAGE`).

  - `PIPE_WAIT` (default) vs. `PIPE_NOWAIT`: `PIPE_WAIT` specifies blocking mode, where calls to `ReadFile`, `WriteFile` and `ConnectNamedPipe` (described later) block until the operation completes. This might mean waiting indefinitely if the client gets stuck for some reason. `PIPE_NOWAIT` causes such calls to return immediately. This latter mode should not be used if possible. Using `FILE_FLAG_OVERLAPPED` flag in the `dwOpenMode` is the preferred way to achieve asynchronous I/O with pipes.
  - `PIPE_ACCEPT_REMOTE_CLIENTS` (default) vs. `PIPE_REJECT_REMOTE_CLIENTS`: specify if the pipe supports connections from other machines or not.
- `nMaxInstances` specifies the maximum number of pipe instances that are allowed to be created simultanously, where `PIPE_UNLIMITED_INSTANCES` (255) indicates unlimited number of instances, constrained by the available system resources. Note that you cannot set an explicit number that is higher than 254 or lower than 1.
- `nOutBufferSize` and `nInBufferSize` specify the amount of memory to commit by the pipes kernel driver so that reads and writes are efficient. If expected buffer sizes are relatively small, the numbers don't matter that much, but if large chunks are expected this might have adverse effect on the system, as this memory is committed from the kernel's non-paged pool (memory region that is never paged out). In any case, the pipe driver is free to use these number as suggestions only.
- `nDefaultTimeOut` is the default timeout (in milliseconds) to wait when calling

`WaitNamedPipe` by a client (described later), where zero means the default of 50 msec.

- `lpSecurityAttributes` is the standard security attributes structure that can be used to modify the security descriptor of the pipe. With `NULL`, the pipe gets default security as follows: The following accounts get full access: *LocalSystem*, Administrators and Owner (stored in the token of the caller thread/process). If the pipe is created from an AppContainer (typically a UWP process), then that AppContainer also has full access. The following accounts get read access: Everyone and annonymous.

A successful call to `CreateNamedPipe` returns a normal handle (a file object), otherwise it returns `INVALID_HANDLE_VALUE`.

With a valid pipe handle, the server can wait for client connections by calling `ConnectNamedPipe`:

```
BOOL ConnectNamedPipe(
    _In_ HANDLE hNamedPipe,
    _Inout_opt_ LPOVERLAPPED lpOverlapped);
```

The first parameter is the pipe handle. The second is an `OVERLAPPED` structure that must be specified if the pipe was open with `FILE_FLAG_OVERLAPPED`. For blocking calls, the function does not return until a client connects. `ConnectNamedPipe` returns `TRUE` on a successful client connection. However, if it returns `FALSE`, it does not necessarily means the function fails. For asyncronous calls, `GetLastError` returns `ERROR_IO_PENDING` (as with all asynchronous calls). Even with a synchronous call, you must call `GetLastError`. If the retunred value is `ERROR_PIPE_CONNECTED` all is well. This just means the client connected between the call to `CreateNamedPipe` and `ConnectNamedPipe`. Any other value returned from `GetLastError` indicates a real error.

Now that a client is connected, the server should call `ReadFile` and/or `WriteFile` as appropriate to deal with the client's request. However, this might take time depending on the actual work involved, so it's better to perform this on a separate thread, while the current thread can go back and create another pipe instance and wait for the next client to connect. This is where the maximum number of instances comes into play. A concrete example is shown in the next sample application.

Once the server is done with a client, it can connect to the next one using the same pipe instance if it calls `DisconnectNamedPipe` first, or just close the handle, ditching this instance:

```
BOOL DisconnectNamedPipe(_In_ HANDLE hNamedPipe);
```

Pipe Client

Similarly to mailslots, named pipe clients use the normal `CreateFile` or `CreateFile2` functions to open a handle to the pipe based on its full name, that must include a machine name if the client and server on different machines.
Next, the client calls `ReadFile` and/or `WriteFile` to read/write data from/to the pipe.

Sometimes a pipe client might attempt to call `CreateFile` before the pipe server is able to initialize, so the call may fail as the pipe name would not be found. Instead, a client can opt to call `WaitNamedPipe` until an instance of the pipe is ready or a timeout elapses:

```
BOOL WaitNamedPipe(
    _In_ LPCTSTR lpNamedPipeName,
    _In_ DWORD nTimeOut);
```

`lpNamedPipeName` is the full pipe name. `nTimeOut` is the interval to wait before returning (in milliseconds), where two special values can be used instead: `NMPWAIT_USE_DEFAULT_WAIT` (0) uses the default specified in `CreateNamedPipe`, while `NMPWAIT_WAIT_FOREVER` waits as long as it takes. After the call completes successfully, the client should call `CreateFile` immediately. The call to `CreateFile` can still fail if another client manages to "steal" that instance of the pipe. In that case, the caller should go back to waiting on `WaitNamedPipe`.

For reading/writing to the pipe, Windows has few shortcuts that can be used instead. One is `CallNamedPipe` that can be used for message-type pipes only:

```
BOOL CallNamedPipe(
    _In_ LPCTSTR lpNamedPipeName,
    _In_reads_bytes_opt_(nInBufferSize) LPVOID lpInBuffer,
    _In_ DWORD nInBufferSize,
    _Out_ LPVOID lpOutBuffer,
    _In_ DWORD nOutBufferSize,
    _Out_ LPDWORD lpBytesRead,
    _In_ DWORD nTimeOut);
```

The function encapsulates opening a pipe, write a message, waiting for a response and closing the pipe handle, effectively a combination of `CreateFile` (with `WaitNamedPipe` if the pipe is not available), `WriteFile`, `ReadFile`, `CloseHandle` and some logic inbetween. Technically, `TransactNamedPipe` (described later) is used instead of `WriteFile`/`ReadFile` combination, but the net effect is similar. The `nTimeOut` parameter has the same meaning as with `WaitNamedPipe` (as that's used internally).

The Pipe Calculator Application

This application consists of a named pipe server, that provides simple calculation services to clients. For demonstration purposes, a message structure is defined with a few message types as shown here:

```
enum class MessageType {
    Sum,
    Product,
    Average,
};

struct InputMessage {
    MessageType Type;
    ULONG Count;
    double Values[ANYSIZE_ARRAY];
};
```

A message contains a type (`MessageType` enum), and an array of `double` values, their count specified as part of the message. Let's assume we are willing to accept no more than than 100 `double`s. Given these constraints, we can calculate the maximum message size:

```
const int MaxElements = 100;
const DWORD MaxInputMessageSize = sizeof(InputMessage)
    + (MaxElements - 1) * sizeof(double);
```

`ANYSIZE_ARRAY` just equals one - it's a typical way to indicate an array of unknown size at compile time.

Here is the beginning of the `main` function of the server. An infinite loop creates named pipe instances and waits for clients to connect:

```
int main() {
    for (;;) {
        HANDLE hNamedPipe = ::CreateNamedPipe(
            L"\\\\.\\pipe\\calculator",
            PIPE_ACCESS_DUPLEX,
            PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE
                | PIPE_WAIT | PIPE_ACCEPT_REMOTE_CLIENTS,
            PIPE_UNLIMITED_INSTANCES,
            sizeof(double), MaxInputMessageSize, 0, nullptr);

        if (hNamedPipe == INVALID_HANDLE_VALUE)
            return Error("Failed to create named pipe");

        printf("Calculator pipe created successfully. Listening...\n");
```

The pipe is configured to work synchronously, as a message-type, with unlimited instances. The sizes of the output and input buffers can be specified as we know what we expect for valid messages.

Now we can wait for a client to connect:

```
if (!::ConnectNamedPipe(hNamedPipe, nullptr) &&
    ::GetLastError() != ERROR_PIPE_CONNECTED)
    return Error("Failed in ConnectNamedPipe");
```

Once connected, we want to deal with this client in a different thread. One option is to use `CreateThread` to create a thread-per-client, which is a popular approach. It's not the best in terms of scalability, though, as there is a cost associated with thread creation, and the number of threads cannot be easily limited in this way. A better (and sometimes simpler) approach is to use the thread pool. Here is the simple version without any special thread pool configurations:

```
    ::TrySubmitThreadpoolCallback(HandleConnection, hNamedPipe,
        nullptr);
}
```

Refer to chapter 9 (part 1) for more details on thread pools. The pipe handle is passed as an argument to the thread pool callback, here named `HandleConnection`. Its first task is to retrieve the handle and allocate a buffer enough to contain the maximum message size:

```
void CALLBACK
HandleConnection(PTP_CALLBACK_INSTANCE, PVOID context) {
    BYTE buffer[MaxInputMessageSize];
    auto hPipe = static_cast<HANDLE>(context);
```

Before making the actual calls to `ReadFile` and `WriteFile`, I have set a map of message type to lambda functions that handle the message type correctly:

```
using namespace std;
static const
unordered_map<MessageType, function<double(double*, ULONG)>> ops{
    { MessageType::Sum, [](auto numbers, auto count) {
        double result = 0;
        for (ULONG i = 0; i < count; i++)
            result += numbers[i];
        return result;
        }
    },
    { MessageType::Average, [](auto numbers, auto count) {
        double result = 0;
        for (ULONG i = 0; i < count; i++)
            result += numbers[i];
        return result / count;
        }
    },
    { MessageType::Product, [](auto numbers, auto count) {
        double result = 1;
        for (ULONG i = 0; i < count; i++)
            result *= numbers[i];
        return result;
        }
    },
};
```

The above code requires the following headers: `<unordered_map>` and `<functional>`.

> Don't be scared of the code above. Each lambda function is just like a normal function that accepts an array of doubles along with the count of items, performs the required operation, and returns the result. Feel free to rewrite this part in any way you see fit.

Now we can get the data from the client, do some checks that message is valid, do the work, and finally return the result. First reading the message and aborting if the client disconnects:

```
DWORD read;
for (;;) {
    if (!::ReadFile(hPipe, buffer, sizeof(buffer), &read, nullptr)
        || read == 0) {
        if (::GetLastError() == ERROR_BROKEN_PIPE) {
            printf("Client disconnected\n");
        }
        else {
            printf("Error reading from pipe (%u)\n", ::GetLastError());
        }
        break;
    }
```

ERROR_BROKEN_PIPE indicates the client closed its end of the pipe, so it's a good time to abort. Next, examine the message and make sure is valid:

```
auto msg = reinterpret_cast<InputMessage*>(buffer);
if (read < sizeof(InputMessage) ||
    read < (msg->Count - 1) * sizeof(double) + sizeof(InputMessage)) {
    printf("Message too short\n");
    break;
}
auto count = msg->Count;
if (count > MaxElements) {
    printf("Too many elements\n");
    break;
}
auto it = ops.find(msg->Type);
if (it == ops.end()) {      // invalid operation
    printf("Unknown math operation\n");
    continue;
}
```

These checks can include anything that is required to make sure the expected message format and data is received. All that's left is to perform the work and return a result:

```
    auto result = (it->second)(msg->Values, msg->Count);
    DWORD written;
    if (!::WriteFile(hPipe, &result, sizeof(result), &written,
        nullptr)) {
        printf("Failed to send result!\n");
        continue;
    }
    printf("Result: %f sent!\n", result);
}
```

Finally, when the client disconnects, just disconnect the pipe (optional) and close the handle:

```
    ::DisconnectNamedPipe(hPipe);
    ::CloseHandle(hPipe);
}
```

The client has the same definitions for the message (usually put in a common header file). The `main` function gets a pipe handle using a combination of `WaitNamedPipe` and `CreateFile` like so:

```
int main() {
    WCHAR pipeName[] = L"\\\\.\\pipe\\calculator";
    HANDLE hPipe;

    int count = 3;  // retry 3 times
    for (;;) {
        ::WaitNamedPipe(pipeName, NMPWAIT_WAIT_FOREVER);
        hPipe = ::CreateFile(pipeName, GENERIC_READ | GENERIC_WRITE,
            0, nullptr, OPEN_EXISTING, 0, nullptr);
        if (hPipe != INVALID_HANDLE_VALUE)
            break;
        if (--count == 0)
            return Error("Cannot establish connection",
                ERROR_NOT_CONNECTED);
    }
```

The client can now send messages and get back results. `CallNamedPipe` can be used, but in this example the normal `WriteFile` and `ReadFile` are used in a helper function:

```
bool SendRequest(HANDLE hPipe, MessageType type,
    const double* values, ULONG count) {
    DWORD messageSize = sizeof(InputMessage) + sizeof(double) * (count -\
 1);
    auto buffer = std::make_unique<BYTE[]>(messageSize);
    auto msg = reinterpret_cast<InputMessage*>(buffer.get());
    msg->Type = type;
    msg->Count = count;
    ::memcpy(msg->Values, values, sizeof(double) * count);

    DWORD written;
    if (!::WriteFile(hPipe, msg, messageSize, &written, nullptr)) {
        printf("Failed to write to pipe (%u)\n", ::GetLastError());
        return false;
    }
    ::FlushFileBuffers(hPipe);

    double result;
    DWORD read;
    if (!::ReadFile(hPipe, &result, sizeof(result), &read, nullptr)) {
        printf("Failed to read result (%u)\n", ::GetLastError());
        return false;
    }

    printf("Result: %f\n", result);
    return true;
}
```

The `main` function just invokes `SendRequest` with various parameters before closing the pipe handle:

```
double values[] = { 12, 3.5, 9.3, 22, .2, 11, 37.8, -10, -6.3 };
SendRequest(hPipe, MessageType::Sum, values, _countof(values));
::Sleep(1000);
SendRequest(hPipe, MessageType::Average, values, _countof(values));
SendRequest(hPipe, MessageType::Product, values, _countof(values));

::CloseHandle(hPipe);
```

Here is the output on the server's console window:

```
Calculator pipe created successfully. Listening...
Calculator pipe created successfully. Listening...
Result: 79.500000 sent!
Result: 8.833333 sent!
Result: 45020462.256000 sent!
Client disconnected
```

And the client's console:

```
Result: 79.500000
Result: 8.833333
Result: 45020462.256000
```

The server project is *CalculatorSvr*, and the client project is *CalcClient*, part of the samples for this chapter.

Replace the `ReadFile` / `WriteFile` calls in `SendRequest` with `CallNamedPipe`.

Other Pipe Functions

The `TransactNamedPipe` can be used either by a server or a client to perform a read/write combination of operations:

```
BOOL TransactNamedPipe(
    _In_  HANDLE hNamedPipe,
    _In_  LPVOID lpInBuffer,
    _In_  DWORD nInBufferSize,
    _Out_ LPVOID lpOutBuffer,
    _In_  DWORD nOutBufferSize,
    _Out_ LPDWORD lpBytesRead,
    _Inout_opt_ LPOVERLAPPED lpOverlapped);
```

The parameters are similar to `ConnectNamedPipe`, but the first parameter is a pipe handle (client or server), which is why `TransactNamedPipe` is more flexible (and in fact used internally by `CallNamedPipe`).

There are quite a few pipe-related functions that allow retrieving information about a pipe: `GetNamedPipeClientComputerName`, `GetNamedPipeClientProcessId`,

`GetNamedPipeClientSessionId`, `GetNamedPipeHandleState`, `GetNamedPipeInfo`, `GetNamedPipeServerProcessId`, and `GetNamedPipeServerSessionId`. Refer to the documentation for their descriptions.

`PeekNamedPipe` can be used to read data from a pipe without removing it. Finally, `ImpersonateNamedPipeClient` can be used by a pipe server to impersonate the client (changing the token of the the current server's thread), so that the operation that needs to be performed on the behalf of the client use the client's security context instead of the server's.

Replace the `ReadFile` / `WriteFile` calls in the server project in the previous section with `TransactNamedPipe`.

Summary

Pipes and mailslots provide flexible client/server communication. Mailslots are simple, unidirectional communication channels, while named pipes are much more flexible. In many scenarios a mailslot server and named pipe servers are implemented within Windows Services, which is the topic of the next chapter.

Chapter 19: Services

The term "Service" is highly overloaded in the software industry. In this chapter, we examine *Windows Services* - standard Windows applications that have some special features:

- Controlled by the *Service Control Manager* (SCM) that can start and stop services (among other things).
- Can run with one of the three special built-in Windows user accounts: *Local System*, *Network Service*, and *Local Service*.
- Can start automatically when the system starts up, without requiring an interactive user log on.

In this chapter:

Services Overview

Services are one of the hallmarks of Windows, having many of these running on a typical Windows system. The *Services* MMC snap-in shows all the services installed on the system, with some important properties (see figure 19-1). We'll discuss all these columns (and more) in later sections in this chapter.

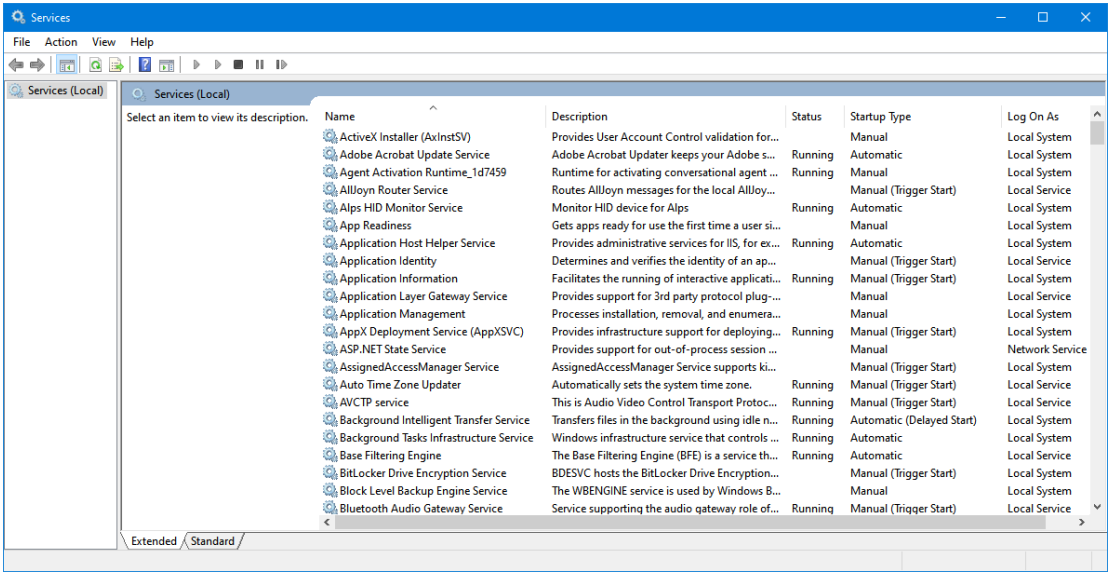


Figure 19-1: The *Services* MMC snapin

Another view of services is available as part of *Task Manager* in the *Services* tab (figure 19-2). The column set is limited, but it does show the process ID where a running service is hosted.

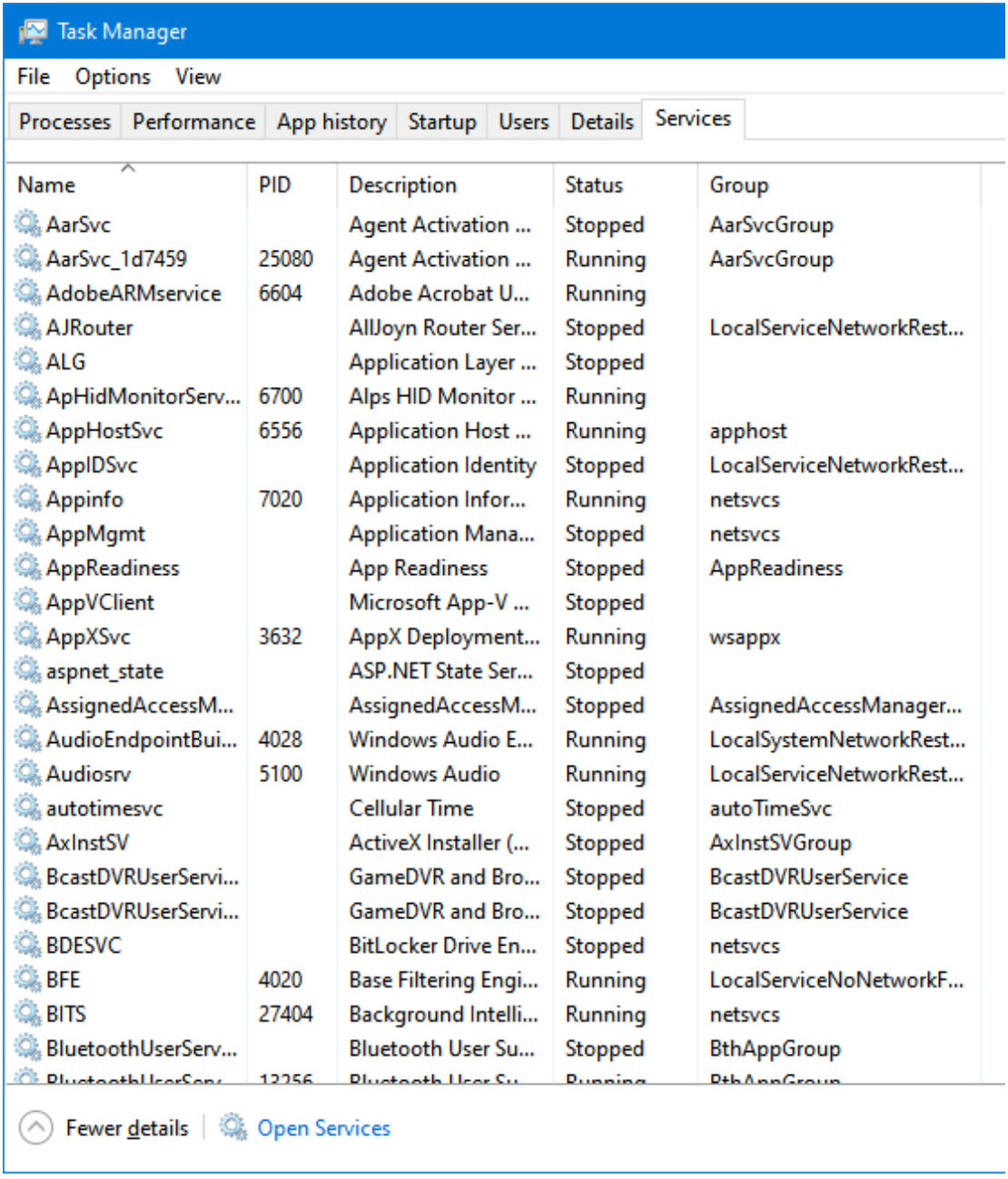


Figure 19-2: Services in *Task Manager*

The user interface provided by the *Services* applet and *Task Manager* allow starting and stopping services, if the logged-on user has administrator rights. All service manipulation requires admin rights, so that users with standard user rights cannot tamper with services in any way.

Manipulating services can be done with command-line tools or code. The most well-known tool is *sc.exe* (short for *Service Control*). This tool provides commands for all service-related manipulations, from creation, to starting and stopping, to configuring and deleting. The other common tool to use is *Powershell*, which has several cmdlets used for interacting with services (such as `Get-Service` and `Start-Service`).

Services-related information is stored in the Registry, under *HKLM\System\CurrentControlSet\Services* (figure 19-3). Technically, this list contains keys for services and device drivers. The entries referring to services are the ones being read by the *Services* applet. The value *Type* in each key specifies whether the key describes a service or a device driver. A small value (typically 1 or 2) indicates a device driver, while larger values (0x10 and higher) indicate a service. The exact values will be discussed later in this chapter.

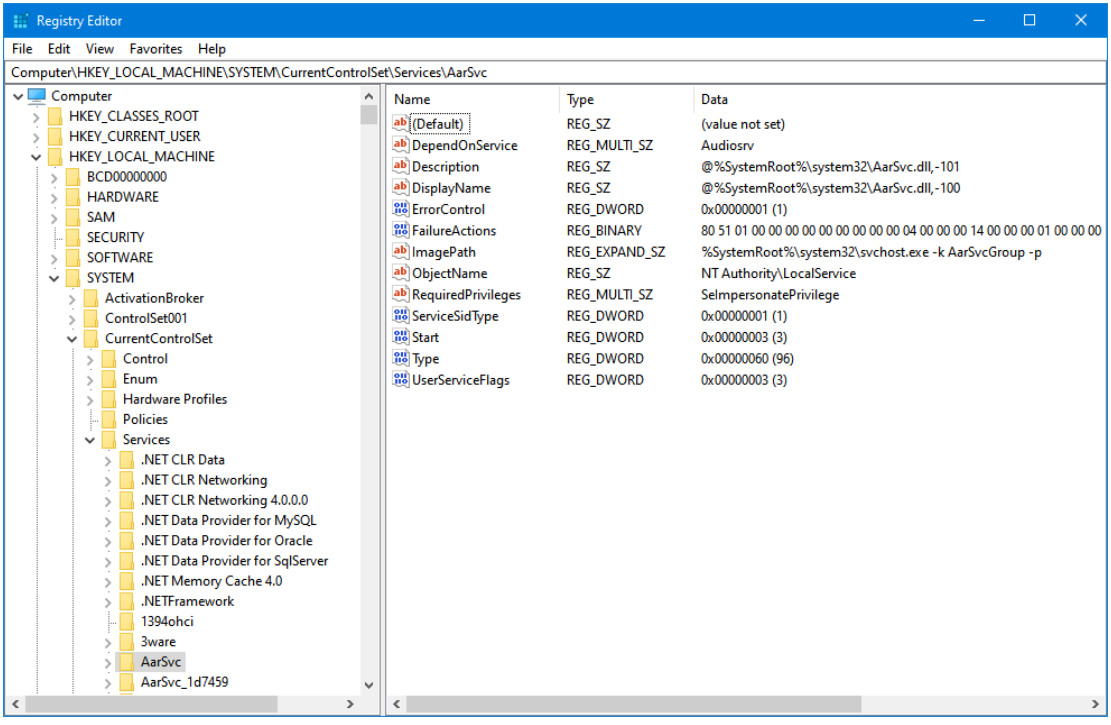


Figure 19-3: Services installed in the Registry

Most of Windows built-in services are implemented as DLLs and hosted in generic processes running the image *Svchost.exe*. This allows hosting multiple services in a single process, thus conserving resources. There are some downsides to this approach, however:

- Stability - if one service crashes, it brings down the entire process (the other services crash as well).
- Security - all the services must be running under the same user account.
- Isolation - somewhat related to stability. Resource consumption is not easily isolated to the relevant service. For example, if a memory leak is detected, or even just large memory consumption in the process, it's difficult to pinpoint the problematic service because all share the same address space.

Up until Windows 10 version 1703, Windows combined multiple services in a single *SvcHost.exe* instance, to reduce memory consumption. Starting from version 1703, if the system has at least 3.5 GB of RAM, most services are hosted in their own processes. 3.5 GB of RAM may seem very little for today's laptop/desktop systems, and that's because it mostly applies to smaller devices, such as phones, small tablets, IoT devices, etc., that might not have that much RAM.

Process Explorer lists the services running in a process by adding a *Services* tab. Figure 19-4 shows an example for the Plug and Play service, hosted in its own process, while figure 19-5 shows a bunch of services hosted in the same process, one of which is the all-important *DCOM Launch* service (responsible, among other things, for launching COM servers - see chapter 21).

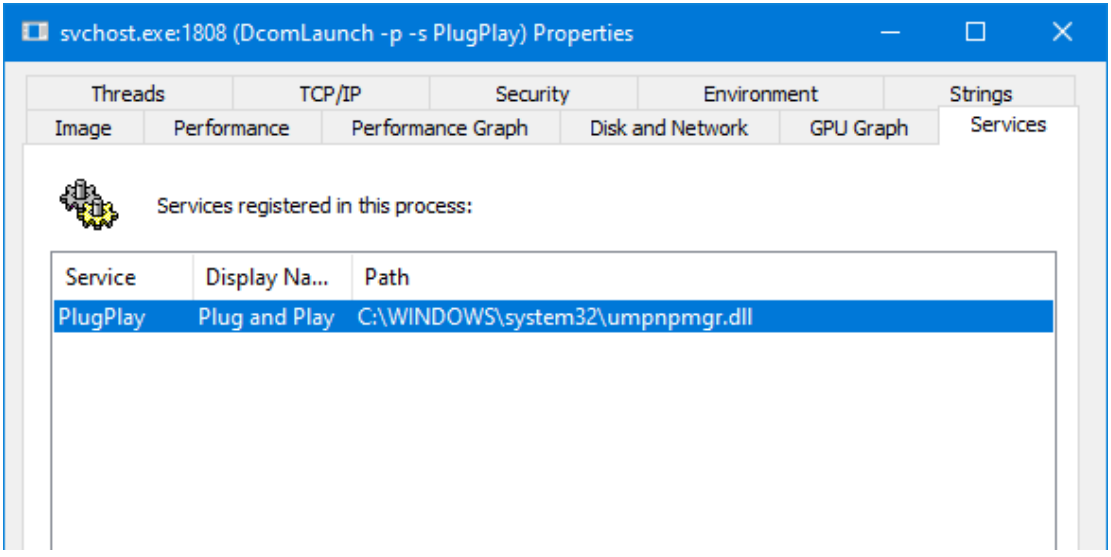


Figure 19-4: A single service in *SvcHost.exe*

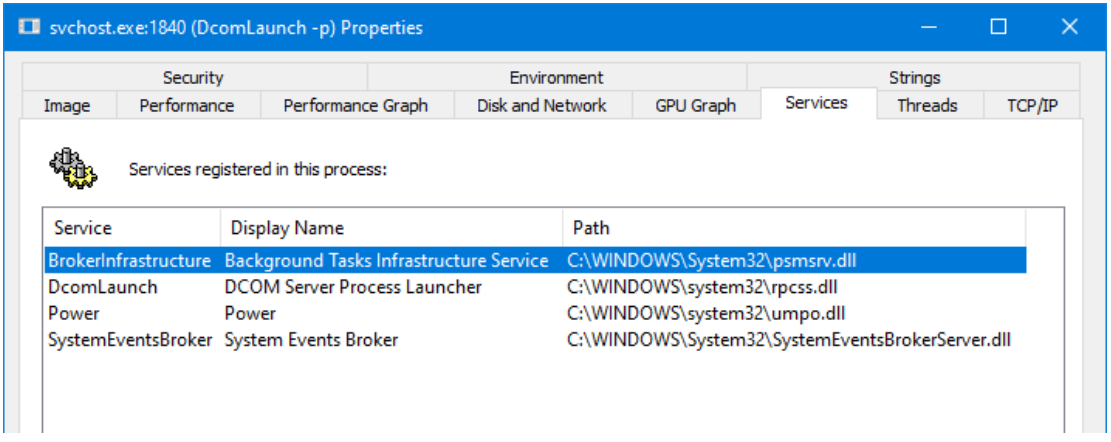


Figure 19-5: Multiple services in *SvcHost.exe*

Service Process Architecture

To allow a service to execute, it must first be installed on the system, as we'll see in the section *Controlling Services*, later in this chapter. Once it's installed, and the service is started, a set of operations commences, depicted in figure 19-6.

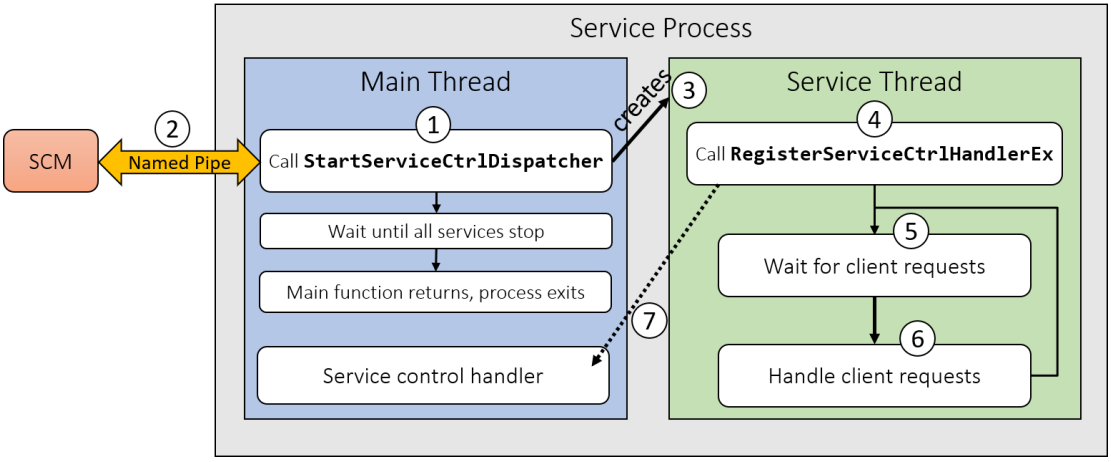


Figure 19-6: Service process architecture

The first thing a `main` function of service needs to do is call `StartServiceCtrlDispatcher` to connect to the SCM and declare the services hosted in this process (1). Once the SCM connects with the service (using a named pipe) (2), `StartServiceCtrlDispatcher` creates a thread for each service hosted in the process (3). This allows each service to have its own execution path, unrelated to the first (main) thread with explicitly creating additional threads.

The service thread that was created calls `RegisterServiceCtrlHandler(Ex)` to register its handler for service-related commands (stop, pause, continue, and others) (4). This handler is always called in the context of the main thread, and so does not interfere with the service's work (7). Then, the service thread initializes itself as appropriate and reports it has started to the SCM by calling `SetServiceStatus` as needed (see the next section for the details). Now the service is ready to receive requests in whatever manner was set up by the service (5). Any request coming in is handled (6).

Back at the main thread, `StartServiceCtrlDispatcher` does not return until all services in the process have stopped. Then the process can exit gracefully.

A Simple Service

To put the previous section into practice, we'll build a simple service called *AlarmSvc*, which provides a very simple alarm notification. The first step is to create a normal C++ console application with *Visual Studio*.

The `main` function must call `StartServiceCtrlDispatcher` to connect with the SCM, declaring the services hosted in this process:

```
typedef struct _SERVICE_TABLE_ENTRY {
    LPTSTR                   lpServiceName;
    LPSERVICE_MAIN_FUNCTION  lpServiceProc;
} SERVICE_TABLE_ENTRY, *LPSERVICE_TABLE_ENTRY;

typedef VOID (WINAPI *LPSERVICE_MAIN_FUNCTION)(
    DWORD   dwNumServicesArgs,
    LPSTR   *lpServiceArgVectors);

BOOL StartServiceCtrlDispatcher(
    _In_ CONST  SERVICE_TABLE_ENTRY* lpServiceStartTable);
```

The function accepts an array of structures, each consisting of a service name as it appears in the Registry and a "main" function for that service, running on a new thread. The last entry must contain two `NULL` values to indicate the end of the list. Here is the `main` function for the *AlarmSvc* service:

```
void WINAPI AlarmMain(DWORD dwNumServicesArgs, LPTSTR* args);

int main() {
    WCHAR name[] = L"alarm";
    const SERVICE_TABLE_ENTRY table[] = {
        { name, AlarmMain },
        { nullptr, nullptr }
    };
    if (!::StartServiceCtrlDispatcher(table))
        return 1;

    return 0;
}
```

In the case where a single service is hosted in the process (single "real" entry in the table), the service name provided is ignored, but must not be `NULL`.

Note that the service name is typed as `LPTSTR`, meaning it should not be part of read-only memory, and that's why the name is first set to a variable on the stack (which is always read/write), before passing it to the structure.

The `main` function does not have much else to do. The `AlarmMain` function is the next interesting part, called by a new thread. Before moving forward, we'll define a few global variables (for convenience and simplicity) that will be used in the upcoming code snippets:

```
SERVICE_STATUS_HANDLE g_hService;
SERVICE_STATUS g_Status;
HANDLE g_hStopEvent;
HANDLE g_hMailslot;
PTP_TIMER g_Timer;
```

The service's main function (referred to as *ServiceMain* in the documentation) starts by initializing some members of `g_Status`:

```
void WINAPI AlarmMain(DWORD dwNumServicesArgs, LPTSTR* args) {
    g_Status.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
    g_Status.dwWaitHint = 500;
```

The arguments to the *ServiceMain* function come from a call to `StartService` that caused the SCM to launch the process in the first place. `StartService` is allowed to pass arguments to the service, and these end up in *ServiceMain*. The first argument is always the name of the service, so `dwNumServicesArgs` is always greater or equal to one. (`StartService` is discussed in the section *Controlling Services*, later in this chapter.)

The service must call `RegisterServiceCtrlHandlerEx` to set up a callback that is called by the SCM when some message is sent to the service:

```
SERVICE_STATUS_HANDLE WINAPI RegisterServiceCtrlHandlerEx(
  _In_     LPCTSTR               lpServiceName,
  _In_     LPHANDLER_FUNCTION_EX lpHandlerProc,
  _In_opt_ LPVOID                lpContext);
```

`lpServiceName` must be the name of the service as registered when the service was installed. `lpHandlerProc` is the callback that must have the following prototype:

```
DWORD WINAPI HandlerEx(
  _In_ DWORD  dwControl,    // control code
  _In_ DWORD  dwEventType,  // event type (for certain codes)
  _In_ LPVOID lpEventData,  // more information (certain events)
  _In_ LPVOID lpContext);   // context passed to
                            // RegisterServiceCtrlHandlerEx
```

We'll see how to implement the handler later in this section. The last parameter to `RegisterServiceCtrlHandlerEx` is a service-defined value that is passed as is to the handler function. The return value is a handle (`SERVICE_STATUS_HANDLE`) used in other functions that need a representation for the service, most notably `SetServiceStatus`, discussed next. Here is the call to `RegisterServiceCtrlHandlerEx` for the alarm service:

```
bool error = true;
do {        // non-loop for easy breaking
    g_hService = ::RegisterServiceCtrlHandlerEx(L"alarmsvc",
        AlarmHandler, nullptr);
    if(!g_hService)
        break;
```

The service is going to be running until told to stop. To that end, the service creates an event object used to signal it to shutdown:

```
g_hStopEvent = ::CreateEvent(nullptr, FALSE, FALSE, nullptr);
if (!g_hStopEvent)
    break;
```

The service is going to expose a mailslot (discussed in chapter 18), accepting messages to set or remove an alarm. For this purpose, the mailslot must be created and configured to be write-accessible by all users. Before starting on this (potentially lengthy) operation, the service should report its status to the SCM, as the SCM is fairly impatient, and might conclude the service is "stuck" in some way if it doesn't report its status frequently enough. The state reported at this stage should be "start pending":

```
SetStatus(SERVICE_START_PENDING);
```

`SetStatus` is a helper function that calls `SetServiceStatus` defined like so:

```
typedef struct _SERVICE_STATUS {
    DWORD   dwServiceType;      // service type
    DWORD   dwCurrentState;     // state
    DWORD   dwControlsAccepted; // accepted control codes
    DWORD   dwWin32ExitCode;
    DWORD   dwServiceSpecificExitCode;
    DWORD   dwCheckPoint;
    DWORD   dwWaitHint;
} SERVICE_STATUS, *LPSERVICE_STATUS;

BOOL SetServiceStatus(
    _In_    SERVICE_STATUS_HANDLE hServiceStatus,
    _In_    LPSERVICE_STATUS      lpServiceStatus);
```

The `SERVICE_STATUS` structure must be filled according to the status to report. `dwServiceType` should be a constant, as it reflects the type of service. We'll discuss all the possible values in the section "Controlling Services" later in this chapter, but the most common ones are `SERVICE_WIN32_OWN_PROCESS` (single service in the process) and `SERVICE_WIN32_SHARED_PROCESS` (the service is sharing the process with other services).

`dwCurrentState` is the state to report. Possible values are shown in table 19-1.

Table 19-1: Service state values

Value	Description
`SERVICE_STOPPED` (1)	Service is not running
`SERVICE_START_PENDING` (2)	Service is initializing
`SERVICE_STOP_PENDING` (3)	Service is shutting down
`SERVICE_RUNNING` (4)	Service is running (active)
`SERVICE_CONTINUE_PENDING` (5)	Service is preparing to continue
`SERVICE_PAUSE_PENDING` (6)	Service is preparing to pause
`SERVICE_PAUSED` (7)	Service has paused its operation

`dwControlsAccepted` indicates to the SCM which commands the service supports. This is a combination of flags shown in tables 19-2 and 19-3. Table 19-2 lists flags that allow sending controls to the service with `ControlService(Ex)`, while table 19-3 lists flags that relate to commands that are sent by the system only. Furthermore, these commands can only be sent to services registering their handler with `RegisterServiceCtrlHandlerEx`, rather than `RegisterServiceCtrlHandler`.

Table 19-2: Accepted by services

Value (`SERVICE_ACCEPT_`)	Description	Controls sent (`SERVICE_CONTROL_`)
`STOP` (1)	Service can be stopped	`STOP`
`PAUSE_CONTINUE` (2)	Service can pause and continue	`PAUSE`, `CONTINUE`
`SHUTDOWN` (4)	Service is notified when the system is shutting down	`SHUTDOWN`
`PARAM_CHANGE` (8)	Service can re-read its startup parameters without stop/start	`PARAMCHANGE`
`NETBINDCGANGE` (0x10)	Service accepts network-related changes without stop/start	`NETBINDADD`, `NETBINDREMOVE`, `NETBINDENABLE`, `NETBINDDISABLE`
`PRESHUTDOWN` (0x100)	Service accepts pre-shutdown control	`PRESHUTDOWN`

Table 19-3: Accepted by services (`RegisterServiceCtrlHandlerEx`)

Value (`SERVICE_ACCEPT_`)	Description	Controls sent (`SERVICE_CONTROL_`)
`HARDWAREPROFILECHANGE` (0x20)	Service is notified when computer's hardware profile changes	`HARDWAREPROFILECHANGE`
`POWEREVENT` (0x40)	Service is notified when the system's power state changes	`POWEREVENT`

Table 19-3: Accepted by services (`RegisterServiceCtrlHandlerEx`)

Value (`SERVICE_ACCEPT_`)	Description	Controls sent (`SERVICE_CONTROL_`)
`SESSIONCHANGE` (0x80)	Service is notified when the session's status has changed	`SESSIONCHANGE`
`TIMECHANGE` (0x200)	Service is notified when the system's time has changed	`TIMECHANGE`
`TRIGGEREVENT` (0x400)	Service is notified when one of its triggers has fired	`TRIGGEREVENT`
`USER_LOGOFF` (0x800)	Service is notified when the user logs off	`USER_LOGOFF`

The *Alarm* service is going to start simple - just accept the stop command. Here is the helper `SetStatus` function:

```
void SetStatus(DWORD status) {
  g_Status.dwCurrentState = status;
  g_Status.dwControlsAccepted =
    status == SERVICE_RUNNING ? SERVICE_ACCEPT_STOP : 0;
  ::SetServiceStatus(g_hService, &g_Status);
}
```

Now that the service reports its "start pending" status, it must complete its initialization before reporting it's in the running state. The *Alarm* service needs to create a mailslot and configure its security descriptor to allow full control to the user running the service, with all other users having write access only. The following code fragment performs all this and creates the mailslot. The security-related code is similar to code encountered in chapter 16 (*Security*):

```
    //
    // create the Everyone SID
    //
    BYTE worldSid[SECURITY_MAX_SID_SIZE];
    DWORD len = sizeof(worldSid);
    auto pWorldSid = (PSID)worldSid;

    if (!::CreateWellKnownSid(WinWorldSid, nullptr, pWorldSid, &len))
        break;

    //
    // get SID of the user running this process
    //
    HANDLE hToken;
```

```
if (!::OpenProcessToken(::GetCurrentProcess(),
    TOKEN_QUERY, &hToken))
    break;

BYTE userBuffer[SECURITY_MAX_SID_SIZE + sizeof(TOKEN_USER) +
    sizeof(SID_AND_ATTRIBUTES)];
auto user = (TOKEN_USER*)userBuffer;
BOOL ok = ::GetTokenInformation(hToken, TokenUser, userBuffer,
    sizeof(userBuffer), &len);
::CloseHandle(hToken);
if (!ok)
    break;
auto ownerSid = user->User.Sid;

//
// allocate and initialize a new security descriptor
//
PSECURITY_DESCRIPTOR sd = ::HeapAlloc(::GetProcessHeap(), 0,
    SECURITY_DESCRIPTOR_MIN_LENGTH);
if (!sd)
    break;
if (!::InitializeSecurityDescriptor(sd,
    SECURITY_DESCRIPTOR_REVISION))
    break;

//
// build ACEs
//
EXPLICIT_ACCESS ea[2] = { 0 };
ea[0].grfAccessPermissions = FILE_ALL_ACCESS;
ea[0].grfAccessMode = SET_ACCESS;
ea[0].grfInheritance = NO_INHERITANCE;
ea[0].Trustee.TrusteeForm = TRUSTEE_IS_SID;
ea[0].Trustee.TrusteeType = TRUSTEE_IS_WELL_KNOWN_GROUP;
ea[0].Trustee.ptstrName = (PWSTR)ownerSid;

ea[1].grfAccessPermissions =
    FILE_GENERIC_WRITE | FILE_GENERIC_READ;
ea[1].grfAccessMode = SET_ACCESS;
ea[1].grfInheritance = NO_INHERITANCE;
ea[1].Trustee.TrusteeForm = TRUSTEE_IS_SID;
ea[1].Trustee.TrusteeType = TRUSTEE_IS_WELL_KNOWN_GROUP;
ea[1].Trustee.ptstrName = (PWSTR)pWorldSid;
```

```
    //
    // create DACL from ACEs
    //
    PACL dacl;
    if (ERROR_SUCCESS != ::SetEntriesInAcl(_countof(ea), ea,
        nullptr, &dacl))
        break;

    //
    // apply owner and DACL
    //
    if (!::SetSecurityDescriptorOwner(sd, ownerSid, FALSE))
        break;

    if (!::SetSecurityDescriptorDacl(sd, TRUE, dacl, FALSE))
        break;

    //
    // create mailslot
    //
    SECURITY_ATTRIBUTES sa = { sizeof(sa) };
    sa.lpSecurityDescriptor = sd;
    g_hMailslot = ::CreateMailslot(L"\\\\.\\mailslot\\Alarm", 1024,
        MAILSLOT_WAIT_FOREVER, &sa);
    if (g_hMailslot == INVALID_HANDLE_VALUE)
        break;

    //
    // cleanup
    //
    ::HeapFree(::GetProcessHeap(), 0, sd);
    ::LocalFree(dacl);
    error = false;
} while (false);
```

If any error is encountered, we need to report the service as stopped and return. Otherwise, the service is ready to do its work and should report itself as running:

```
if (error) {
    SetStatus(SERVICE_STOPPED);
    return;
}

SetStatus(SERVICE_RUNNING);
```

What about notifying someone about the error encountered? A typical service runs in session 0, so cannot show any UI, except for a very limited message possible with `WTSSendMessage` (that we will use soon to implement the alarm notification). The most common error reporting mechanism used by services is writing to the Windows event log. We'll examine the event log in the next chapter (*Debugging and Diagnostics*). For now, we'll do nothing, or just use `OutputDebugString` calls that can be captured by a tool such as *DbgView* from *Sysinternals*.

The service now needs to listen to the mailslot for incoming messages. First, we'll add a common header file that describes the accepted messages:

```
// AlarmCommon.h

enum class MessageType {
    SetAlarm,
    CancelAlarm,
};

struct AlarmMessage {
    MessageType Type;
    FILETIME Time;
};
```

To set the alarm, a `FILETIME` structure representing the target time is provided. If the request is to cancel the alarm, the `Time` member will be ignored.

The following code examines the mailslot once per second, and handles the request(s) that come in. A `while` loop is used until the stop event is signaled, which causes the loop to exit and report to the SCM that the service is stopped.

```
    DWORD msgSize, count;
    AlarmMessage msg;

    while (::WaitForSingleObject(g_hStopEvent, 1000) == WAIT_TIMEOUT) {
        do {
            if (!::GetMailslotInfo(g_hMailslot, nullptr, &msgSize,
                &count, nullptr))
                break;
            if (msgSize == MAILSLOT_NO_MESSAGE)
```

```
                break;

            DWORD bytes;
            if (msgSize == sizeof(msg) &&
                ::ReadFile(g_hMailslot, &msg, sizeof(msg), &bytes, nullp\
tr)) {
                HandleMessage(msg);
            }
            count--;
        } while (count > 0);

    }
    SetStatus(SERVICE_STOPPED);

    ::CloseHandle(g_hStopEvent);
    ::CloseHandle(g_hMailslot);
}
```

The missing function is `HandleMessage`:

```
void HandleMessage(const AlarmMessage& msg) {
    switch (msg.Type) {
        case MessageType::SetAlarm:
            if (g_Timer == nullptr)
                g_Timer = ::CreateThreadpoolTimer(OnTimerExpired,
                    nullptr, nullptr);
            ::SetThreadpoolTimer(g_Timer,
                (PFILETIME)&msg.Time, 0, 1000);
            break;

        case MessageType::CancelAlarm:
            if (g_Timer) {
                ::CloseThreadpoolTimer(g_Timer);
                g_Timer = nullptr;
            }
            break;
    }
}
```

The function uses a thread pool timer we encountered in chapter 9 (in part 1) if the alarm is to be set. `OnTimerExpired` is called by a thread pool thread when the timer expires. This is where the wakeup message is shown to the user.

```
void NTAPI OnTimerExpired(PTP_CALLBACK_INSTANCE, PVOID, PTP_TIMER) {
    WCHAR title[] = L"!!! ALARM !!!";
    WCHAR msg[] = L"It's time to wake up!";
    DWORD response;

    ::WTSSendMessage(nullptr, ::WTSGetActiveConsoleSessionId(),
        title, sizeof(title), msg, sizeof(msg),
        MB_OK | MB_ICONEXCLAMATION, 0, &response, FALSE);
}
```

The `WTSSendMessage` function is part of the *Windows Terminal Services* API, which allows sending a message box-like message to a session. `WTSGetActiveConsoleSessionId` returns the active console session ID (where the current user is logged in), so that the message is shown in the correct session. Here are the definitions for both APIs:

```
BOOL WTSSendMessage(
    _In_ HANDLE hServer,
    _In_ DWORD SessionId,
    _In_reads_bytes_(TitleLength) LPTSTR pTitle,
    _In_ DWORD TitleLength,
    _In_reads_bytes_(MessageLength) LPTSTR pMessage,
    _In_ DWORD MessageLength,
    _In_ DWORD Style,
    _In_ DWORD Timeout,
    _Out_ DWORD * pResponse,
    _In_ BOOL bWait);

DWORD WTSGetActiveConsoleSessionId();
```

Notice that the message text and title length are specified in bytes, rather than characters, which is very unusual for Windows APIs. This is why the code above uses the `sizeof` operator rather than using something like the `_countof` macro or the `wcslen` function. `WTSSendMessage` has most of the functionality from the `MessageBox` API, like the ability to get a response back from the user and show multiple buttons (Ok, Cancel, etc.).

Add support for multiple alarms.

Installing the Service

Now that we have a service executable ready to go, it needs to be installed. The classic tool to use is *sc.exe*, built-in in Windows. (sc=*Service Control*). A more modern tool to use is a set of cmdlets in PowerShell. Either works, internally calling APIs we'll examine in the section *Controlling Services*, later in this chapter.

A service must be registered in the SCM's database using the *sc create* command. Here is the minimum required to install the *Alarm* service, assuming it's in *c:\temp* (run from an elevated command window):

```
c:\>sc create alarmsvc binPath= c:\Temp\AlarmSvc.exe
```

Notice the equal sign next to *binPath* and space right after it (*sc.exe* is quite picky about spacing). You'll have to use the correct path to the service executable. If the operation succeeds, you'll see the following displayed:

```
[SC] CreateService SUCCESS
```

The installation can be verified by querying the SCM with *sc.exe*:

```
c:/>sc query alarmsvc

SERVICE_NAME: alarmsvc
        TYPE               : 10  WIN32_OWN_PROCESS
        STATE              : 1  STOPPED
        WIN32_EXIT_CODE    : 1077  (0x435)
        SERVICE_EXIT_CODE  : 0  (0x0)
        CHECKPOINT         : 0x0
        WAIT_HINT          : 0x0
```

You can also open the *Registry* editor and navigate to *HKLM\System\CurrentControlSet\Services\alarmsvc*. You should see something similar to figure 19-7.

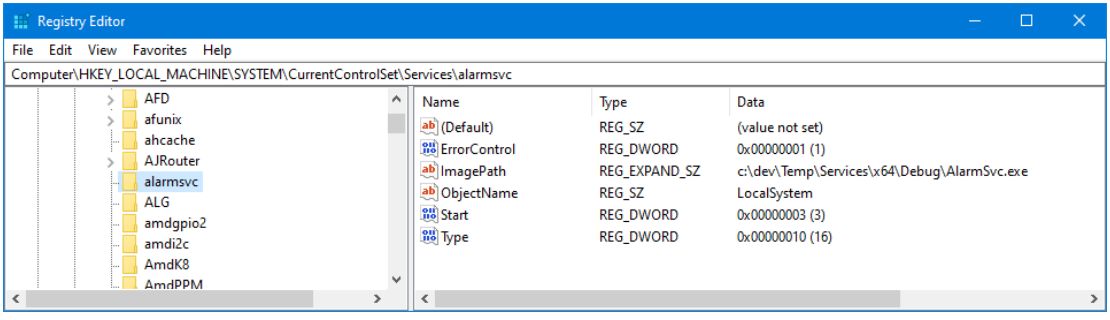


Figure 19-7: The alarm service in the Registry

Now that the service is installed, it's time to start it, using *sc.exe* again:

```
c:\>sc start alarmsvc

SERVICE_NAME: alarmsvc
        TYPE               : 10  WIN32_OWN_PROCESS
        STATE              : 2  START_PENDING
                (NOT_STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE    : 0  (0x0)
        SERVICE_EXIT_CODE  : 0  (0x0)
        CHECKPOINT         : 0x0
        WAIT_HINT          : 0x1f4
        PID                : 46724
        FLAGS              :

c:\>sc query alarmsvc

SERVICE_NAME: alarmsvc
        TYPE               : 10  WIN32_OWN_PROCESS
        STATE              : 4  RUNNING
                (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE    : 0  (0x0)
        SERVICE_EXIT_CODE  : 0  (0x0)
        CHECKPOINT         : 0x0
        WAIT_HINT          : 0x0
```

Notice the PID running the new service, which you can verify in *Task Manager*. You'll see the process is running under the local system account and in session 0 (figure 19-8).

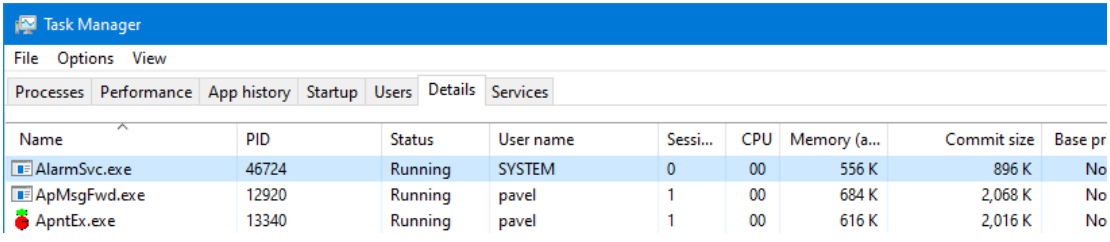


Figure 19-8: The alarm service process

Stopping the service is easy enough with *sc.exe*:

```
c:/>sc stop alarmsvc

SERVICE_NAME: alarmsvc
        TYPE               : 10  WIN32_OWN_PROCESS
        STATE              : 1  STOPPED
        WIN32_EXIT_CODE    : 0  (0x0)
        SERVICE_EXIT_CODE  : 0  (0x0)
        CHECKPOINT         : 0x0
        WAIT_HINT          : 0x0
```

Finally, if deleting (uninstalling) the service is required, *sc.exe* is up to the task:

```
c:/>sc delete alarmsvc

[SC] DeleteService SUCCESS
```

> You can also use Powershell cmdlets to perform the previously shown operations. `Get-Service` returns information about services, `New-Service` installs a new service, `Start-Service` starts a service, `Stop-Service` stops a service, and there are some more related cmdlets: `Set-Service`, `Suspend-Service`, `Resume-Service` and `Restart-Service`.

A Service Client

Now that we have a working service, we need to write a client that sends appropriate messages to the mailslot. The client is going to accept command-line arguments to set or cancel the alarm:

```
// alarm.cpp

#include <Windows.h>
#include <stdio.h>
#include <string>
#include "..\AlarmSvc\AlarmCommon.h"

int main(int argc, const char* argv[]) {
    if (argc < 2) {
        printf("Usage: alarm <set hh:mm:ss | cancel>\n");
        return 0;
    }
```

The client only allows setting the time (without date), meaning the alarm must be set off "today". This is done just for simplicity's sake. The service itself doesn't care, as it works with a `FILETIME` value, whatever it may be.

Next, we open a handle to the mailslot. If that fails, the service is probably down:

```
HANDLE hFile = ::CreateFile(L"\\\\.\\mailslot\\Alarm",
    GENERIC_WRITE, 0, nullptr, OPEN_EXISTING, 0, nullptr);
if (hFile == INVALID_HANDLE_VALUE)
    return Error("Failed to open mailslot");
```

The `Error` helper function is the usual one we used many times before to print an error message and quit. The next part is sending the correct command and arguments to the mailslot as defined by the `AlarmMessage` structure.

```
    DWORD bytes;
    if (_stricmp(argv[1], "set") == 0 && argc > 2) {
        AlarmMessage msg;
        msg.Type = MessageType::SetAlarm;
        std::string time(argv[2]);
        if (time.size() != 8) {
            printf("Time format is illegal.\n");
            return 1;
        }
        //
        // get local time (for obtaining the date)
        //
        SYSTEMTIME st;
        ::GetLocalTime(&st);
        //
        // change the time based on user input
        //
        st.wHour = atoi(time.substr(0, 2).c_str());
        st.wMinute = atoi(time.substr(3, 2).c_str());
        st.wSecond = atoi(time.substr(6, 2).c_str());
        ::SystemTimeToFileTime(&st, &msg.Time);

        //
        // convert to UTC time
        //
        ::LocalFileTimeToFileTime(&msg.Time, &msg.Time);

        //
        // write to the mailslot
```

```
        //
        if (!::WriteFile(hFile, &msg, sizeof(msg), &bytes, nullptr))
            return Error("failed in WriteFile");
    }
    else if (_stricmp(argv[1], "cancel") == 0) {
        AlarmMessage msg;
        msg.Type = MessageType::CancelAlarm;
        if (!::WriteFile(hFile, &msg, sizeof(msg), &bytes, nullptr))
            return Error("failed in WriteFile");
    }
    else {
        printf("Error in command line arguments.\n");
    }
    ::CloseHandle(hFile);

    return 0;
}
```

While the service is running, we can run the client like so:

```
alarm set 13:33:00
```

When the set time arrives, a message box will pop up (figure 19-9). Note that if the set time is in the past, the message box will pop up immediately.

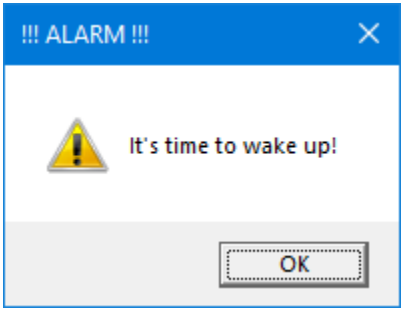


Figure 19-9: Alarm service notification

Controlling Services

The previous section used the *sc.exe* tool to install, start, stop, and uninstall the service. As you might expect, the tool is merely using Windows APIs to perform these (and other) operations. In this section, we'll examine the APIs used for such activities.

Installing a Service

Service installation is achieved with the `CreateService` API:

```
SC_HANDLE CreateService(
    _In_       SC_HANDLE hSCManager,
    _In_       LPCTSTR   lpServiceName,
    _In_opt_   LPCTSTR   lpDisplayName,
    _In_       DWORD     dwDesiredAccess,
    _In_       DWORD     dwServiceType,
    _In_       DWORD     dwStartType,
    _In_       DWORD     dwErrorControl,
    _In_opt_   LPCTSTR   lpBinaryPathName,
    _In_opt_   LPCTSTR   lpLoadOrderGroup,
    _Out_opt_  LPDWORD   lpdwTagId,
    _In_opt_   LPCTSTR   lpDependencies,
    _In_opt_   LPCTSTR   lpServiceStartName,
    _In_opt_   LPCTSTR   lpPassword);
```

Quite a few parameters! Before `CreateService` (or any other service-related function for that matter), can do anything, it must open a handle to the *Service Control Manager* with `OpenSCManager`, later passed in as the first argument to other functions:

```
SC_HANDLE OpenSCManager(
    _In_opt_  LPCTSTR  lpMachineName,
    _In_opt_  LPCTSTR  lpDatabaseName,
    _In_      DWORD    dwDesiredAccess);
```

`lpMachineName` is the computer name to which to connect. Typically this value is set to `NULL` or an empty string, which indicates the local machine. `lpDatabaseName` must be `SERVICES_ACTIVE_DATABASE` or `NULL`, which implies the former. Finally, `dwDesiredAccess` indicates the access required to the SCM database. `SC_MANAGER_CONNECT` is always implied, but other values may be needed for certain operations. To make a successful call to `CreateService`, the `SC_MANAGER_CREATE_SERVICE` access right is required, which normally is granted to administrators. In fact, local administrators are granted `SC_MANAGER_ALL_ACCESS`, meaning they can do anything.

`OpenSCManager` returns a handle to the SCM, or `NULL` on failure. Note that the handle has a special type, and is not like handles we met before, since the SCM is not a kernel object. The returned handle is just used as an opaque value representing the connection to SCM. When the handle is no longer needed, call `CloseServiceHandle` to close it.

```
BOOL CloseServiceHandle(_In_ SC_HANDLE hSCObject);
```

Don't use `CloseHandle` to close a SCM or a service handle!

The *Windows Implementation Library* (WIL) has a handle type that calls `CloseServiceHandle` in its destructor called `wil::unique_schandle`. This is similar to the `wil::unique_handle` uses many times in part 1.

Now we can turn our attention to `CreateService`. `lpServiceName` is the service name to create, which must be unique, as can be viewed under the Registry key *HKLM\System\CurrentControlSet\Services*. `lpDisplayName` is a friendly display name normally visible in the *Services* applet. It can be `NULL`, in which case the service name serves as a display name as well.

`dwDesiredAccess` is the access mask required for the returned service handle. For example, if after a successful call to `CreateService`, the service should be started, then asking for `SERVICE_START` will return a powerful-enough handle to call the `StartService` API without the need to call `OpenService` again (these functions are discussed later in subsequent sections).

`dwServiceType` indicates what type of service to create. `CreateService` is also capable of installing kernel device drivers, not just user-mode services. Table 19-4 shows the documented values for the service type.

Table 19-4: Service types

Type (`SERVICE_`)	Description
`KERNEL_DRIVER` (1)	Generic driver
`FILE_SYSTEM_DRIVER` (2)	File system or file system filter driver
`WIN32_OWN_PROCESS` (0x10)	Service hosted in its own process
`WIN32_SHARE_PROCESS` (0x20)	Service that can share a process with other services
`USER_OWN_PROCESS` (0x50)	Per-user service hosted in its own process (Windows 10+)
`USER_SHARE_PROCESS` (0x60)	Per-user service that shares a process with other services (Windows 10+)

The first two values are for installing drivers, which is no different than installing services, except some service parameters are irrelevant for drivers.

An extra flag can be added to the values in table 19-4 for services only, `SERVICE_INTERACTIVE_PROCESS` (0x100), which is related to *Interactive Services*, discussed later in this chapter. Most services created by third parties (not built-in with Windows) specify `WIN32_OWN_PROCESS`, meaning the service is the only one hosted in a process.

The `WIN32_SHARE_PROCESS` value is used by many built-in Windows services, and used as such in Windows versions prior to Windows 10 version 1703, where multiple services are bundled together in a single process to conserve resources. The downside is the potential effect one service has on others. If one

service causes an exception that is unhandled, the entire process crashes, bringing down all services in the process.

> Starting with Windows 10 version 1703 ("RS2"), if the system has at least 3.5 GB of RAM, most Windows built-in services are hosted in their own process to increase robustness. It's also easier to debug and diagnose (from Microsoft's point of view). 3.5 GB of RAM might seem a very low value, easily obtainable in today's laptop/desktop systems. The value becomes more relevant when smaller devices are considered, such as the Hololens, and other *Internet of Things* (IoT) running Windows 10.

The last two values in table 19-4 are related to per-user services, discussed in their own section, later in this chapter.

The `dwStartType` parameter indicates when (and if) the service/driver should start. Table 19-5 lists the possible values.

Table 19-5: Service start type

Type (`SERVICE_`)	Description
`BOOT_START` (0)	(valid for kernel drivers only) Driver is loaded early in the boot process
`SYSTEM_START` (1)	(valid for kernel drivers only) Driver is loaded at kernel initialization completion
`AUTO_START` (2)	Service/driver is loaded automatically by the SCM when the Windows subsystem is available
`DEMAND_START` (3)	Service/driver is not loaded automatically
`DISABLED` (4)	Service/driver cannot be loaded

The first two values in table 19-5 are applicable for kernel drivers only, so will not be discussed here. `SERVICE_AUTO_START` is the first value valid for services. Such services start automatically, without requiring any logged on user to be present. Many built-in services have this setting. An augmentation of auto-start is delayed auto start, that must be explicitly configured with the `ChangeServiceConfig2` API (or a comparable tool), indicates the service should start automatically, but approximately two minutes after the SCM initializes, so as to interfere less with a quick-logging user's processes. `ChangeServiceConfig2` is discussed in the section *Configuring Services*, later in this chapter.

`SERVICE_DEMAND_START` indicates the service is not started automatically. To start it, an explicit call must be made to `StartService` (or a comparable tool used). There is yet another variation for demand-start services, called trigger-start, discussed in the *Configuring Service* section as well.

Lastly, `SERVICE_DISABLED` indicates the service cannot be started at all. To start such a service, its start type must be changed to something other than disabled, and then `StartService` must be called. Such a change can only be made by an administrator anyway.

The next parameter, `dwErrorControl`, indicates what should be done if service initialization fails for any reason. Table 19-6 lists the possible values.

Table 19-6: Error control

Type (`SERVICE_ERROR_`)	Description
`IGNORE` (0)	Do nothing
`NORMAL` (1)	The error is logged to the event log service
`SEVERE` (2)	The error is logged to the event log. If the current configuration is the *Last Known Good*, the system continues normally. Otherwise, Windows is restarted in the *Last Known Good* configuration.
`CRITICAL` (3)	Similar to `SEVERE` above, but if the configuration is already *Last Known Good*, crashes the system ("Blue Screen of Death").

The values of `SERVICE_ERROR_SEVERE` and `SERVICE_ERROR_CRITICAL` may be needed for critical drivers and services only. Most services are not critical to a system, so these values should not normally be used. The most common (and recommended) value is `SERVICE_ERROR_NORMAL`, which logs an error to the event log, but otherwise the system continues running normally.

> The term *Last Known Good* indicates a system configuration that resulted in a successful boot. The exact details are beyond the scope of this chapter, since this is mostly relevant to kernel boot drivers.

Next, the `lpBinaryPathName` is the full path to the executable to run for the service, which can include command line arguments that are passed to the standard `main` function. If the path contains spaces, it should be surrounded by quatation marks to make sure it's interpreted correctly.

`lpLoadOrderGroup` is an optional string specifying the load order group this service belongs to. With a load order group, the order of service loading can be controlled to some extent. Specify `NULL` or an empty string so that the service is not part of any group (the common value). Without a group, the service starts after all services with groups start (with the same start type value). The list of group names is located in the Registry key *HKLM\System\CurrentControlSet\Control\ServiceGroupOrder* in the *List* multiple-string value.

The `lpdwTagId` optional parameter applies to kernel drivers only. Briefly, the return value provides a "tag", uniquely identifying the driver within its group. It can be used to further order driver loading within a single group by adding the tag to the group's name in the Registry key *HKLM\System\CurrentControlSet\Control\GroupOrderList*.

`lpDependencies` is an optional multiple string of service names and/or group names that the service being created depends on. If the value is `NULL`, the service is independent. Each service/group in the string must be separated by `\0` and another `\0` must end the list (same format as a `REG_MULTI_SZ` Registry value). For groups, each group name must be preceded with a `+` character, to avoid conflicts with service names. Additionally, being dependent on a group means that the service can load if at least one service from the group started succesfully.

`lpServiceStartName` is the account under which the service process should execute. Here are the possible options:

- `NULL` causes the service to execute under the *LocalSystem* built-in account.
- `NT AUTHORITY\NetworkService` indicates the process should run under the *Network Service* built-in account.
- `NT AUTHORITY\LocalService` indicates the process should run under the *Local Service* built-in account.
- User name in the format *Domain\Name* causes the process to run under the indicated user. The user is signed in (with the password specified in the last parameter) if the user is allowed to log-on as a service. This user right can be set with administrative tools, such as the built-in *Local Security Policy* tool for local users. Figure 19-10 shows the user right. If the user is on the local machine, *Domain* can be set to ".".

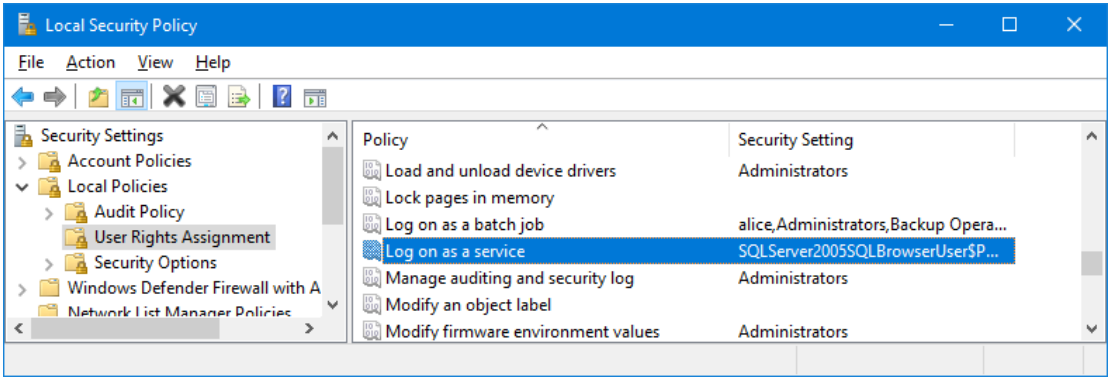


Figure 19-10: Log in as Service use right

A service can also be configured to run under a *virtual* or *managed* account (see the section *Service Security* for more information, later in this chapter). In that case, this parameter is the managed/virtual account name.

The final parameter, `lpPassword`, is the password to use in case of a *domain\user* type of login. In all other cases, this should be set to `NULL`.

The password is not stored as plain text in the service key in the registry (or at all in that key).

The return value of `CreateService` is a service handle, typed as `SC_HANDLE`, just like a SCM handle. The permissions associated with the handle are those specified by the `dwDesiredAccess` parameter.

In the *Alarm* service, we may want to provide command line options to install, uninstall, start and stop the service using the executable itself, rather than forcing a user to use a tool (such as *sc.exe*) with the correct parameters.

To that end, we'll add an `InstallService` function, that would install the service in its own process to run under the *Local Service* account:

```
int InstallService() {
    SC_HANDLE hScm = ::OpenSCManager(nullptr, nullptr,
        SC_MANAGER_CREATE_SERVICE);
    if (!hScm)
        return Error("Failed to open SCM database");

    WCHAR path[MAX_PATH];
    ::GetModuleFileName(nullptr, path, _countof(path));

    SC_HANDLE hService = ::CreateService(hScm,
        L"alarmsvc",
        L"Alarm Demo Service",
        SERVICE_ALL_ACCESS,
        SERVICE_WIN32_OWN_PROCESS,
        SERVICE_DEMAND_START,
        SERVICE_ERROR_NORMAL,
        path,
        nullptr, nullptr, nullptr,
        L"NT AUTHORITY\\LocalService", nullptr);
    if (!hService)
        return Error("Failed to install service");

    printf("Service installed successfully.\n");

    ::CloseServiceHandle(hService);
    ::CloseServiceHandle(hScm);

    return 0;
}
```

We'll tie the 'InstallService' function to the command line arguments later in the upcoming sections.

Starting a Service

Once a service (or driver) is installed, starting it is a matter of calling `StartService`:

```
BOOL StartService(
    _In_  SC_HANDLE hService,
    _In_  DWORD     dwNumServiceArgs,
    _In_  LPCWSTR   *lpServiceArgVectors);
```

The function requires a service handle (`hService`) with at least `SERVICE_START` access mask. The other parameters are optional "startup" arguments, passed to the *ServiceMain* callback registered with

StartServiceControlDispatcher. Similarly to `InstallService`, it would be nice to have our own service starting function within the *Alarm* service:

```
int StartService() {
    SC_HANDLE hScm = ::OpenSCManager(nullptr, nullptr,
        SC_MANAGER_CONNECT);
    if (!hScm)
        return Error("Failed to open SCM database");

    SC_HANDLE hService = ::OpenService(hScm, L"alarmsvc",
        SERVICE_START);
    if (!hService)
        return Error("Failed to open service");

    if (::StartService(hService, 0, nullptr))
        printf("Service started successfully.\n");
    else
        return Error("Failed to start service");
    ::CloseServiceHandle(hService);
    ::CloseServiceHandle(hScm);
    return 0;
}
```

Stopping a Service

All commands to the service except *StartService* are delivered via one of the generic functions *ControlService* or *ControlServiceEx*:

```
BOOL ControlService(
    _In_  SC_HANDLE         hService,
    _In_  DWORD             dwControl,
    _Out_ LPSERVICE_STATUS  lpServiceStatus);

BOOL ControlServiceEx(
    _In_    SC_HANDLE  hService,
    _In_    DWORD      dwControl,
    _In_    DWORD      dwInfoLevel,
    _Inout_ PVOID      pControlParams);
```

We'll start with `ControlService` and then discuss the differences of `ControlServiceEx`. `hService` is a handle to the service to which a control code is to be sent. The access mask for this handle must match the type of control code sent (`dwControl` parameter). Table 19-7 lists the possible control codes.

Table 19-7: Control codes with `ControlService(Ex)`

Type (`SERVICE_CONTROL_`)	Access mask	Description
`STOP` (1)	`SERVICE_STOP`	Stop the service
`PAUSE` (2)	`SERVICE_PAUSE_CONTINUE`	Pause the service
`CONTINUE` (3)	`SERVICE_PAUSE_CONTINUE`	Resume the service
`INTERROGATE` (4)	`SERVICE_INTERROGATE`	Asks the service to report its status
`PARAMCHANGE` (6)	`SERVICE_PAUSE_CONTINUE`	Notifies the service that the startup parameters have changed

The control codes `SERVICE_CONTROL_NETBINDADD` (7), `SERVICE_CONTROL_NETBINDREMOVE` (8) `SERVICE_CONTROL_NETBINDENABLE` (9), and `SERVICE_CONTROL_NETBINDDISABLE` (10) have been deprecated. The recommendation is to use Plug & Play notifications (by calling `RegisterDeviceNotifications` - see the docs for details). In addition to the control codes mentioned, the service can accept custom control codes in the range 128 to 255.

Each request to the service is handled serially, so a control code is not sent until a previous control code has been handled by the service. The SCM will allow 30 seconds for the service to handle the control code. If the timeout elapses, `ControlService` fails and returns `ERROR_SERVICE_REQUEST_TIMEOUT`. Other error values are possible (check the documentation).

Any control code arrives to the service handler callback, registered with `RegisterServiceCtrlHandler(Ex)`. If the service didn't indicate it accepts the control code in question, the function fails immediately. Consult the documentation for the details of how a service should respond to the various control codes.

The last parameter to `ControlService` is the latest status reported by the service to the SCM. If the function fails because the service is not running, the control code is invalid or the service is not accepting such control code, the structure is not filled in.

`ControlServiceEx` drops the service status return structure (you can always call `QueryServiceConfig` and/or `QueryServiceStatus(Ex)` to get the same and other information about the service - see later in this chapter), and adds the ability to provide extra information with the `pControlParams` parameter. This parameter must be non-`NULL` if `dwInfoLevel` is set to `SERVICE_CONTROL_STATUS_REASON_INFO` (currently the only supported value). This is only valid for `SERVICE_CONTROL_STOP` request. In that case, the following structure is to be passed to the function:

```
typedef struct _SERVICE_CONTROL_STATUS_REASON_PARAMS {
    DWORD                   dwReason;
    LPTSR                   pszComment;
    SERVICE_STATUS_PROCESS  ServiceStatus;
} SERVICE_CONTROL_STATUS_REASON_PARAMS;
```

The `dwReason` member should be set to one of several possible reasons listed in the documentation for why the service is being sent a stop control code. `pszComment` is an optional comment for the stop request. Finally, `ServiceStatus` is filled upon return from `ControlServiceEx` with the status of the service.

`SERVICE_STATUS_PROCESS` can be thought of as an extended version of `SERVICE_STATUS`, which includes information on the service process:

```
typedef struct _SERVICE_STATUS_PROCESS {
    DWORD   dwServiceType;
    DWORD   dwCurrentState;
    DWORD   dwControlsAccepted;
    DWORD   dwWin32ExitCode;
    DWORD   dwServiceSpecificExitCode;
    DWORD   dwCheckPoint;
    DWORD   dwWaitHint;
    DWORD   dwProcessId;    // PID of service (if running)
    DWORD   dwServiceFlags;
} SERVICE_STATUS_PROCESS, *LPSERVICE_STATUS_PROCESS;
```

The *Alarm* service defines a simple `StopService` function to stop the service by calling `ControlService`, if so requested:

```
int StopService() {
    auto hScm = ::OpenSCManager(nullptr, nullptr, SC_MANAGER_CONNECT);
    if (!hScm)
        return Error("Failed to open SCM database");

    auto hService = ::OpenService(hScm, L"alarmsvc", SERVICE_STOP);
    if (!hService)
        return Error("Failed to open service");

    if (::ControlService(hService, SERVICE_CONTROL_STOP, &g_Status))
        printf("Service stopped successfully.\n");
    else
        return Error("Failed to stop service");
    ::CloseServiceHandle(hService);
    ::CloseServiceHandle(hScm);

    return 0;
}
```

Uninstalling the Service

Uninstalling a service is accomplished with `DeleteService`:

```
BOOL DeleteService(_In_ SC_HANDLE hService);
```

What could be simpler? The only requirement is that `hService` must have the `DELETE` and `SERVICE_QUERY_STATUS` access masks. Here is the `UninstallService` function in the *Alarm* service:

```
int UninstallService() {
    auto hScm = ::OpenSCManager(nullptr, nullptr,
        SC_MANAGER_CONNECT);
    if (!hScm)
        return Error("Failed to open SCM database");

    auto hService = ::OpenService(hScm, L"alarmsvc",
        DELETE | SERVICE_QUERY_STATUS);
    if (!hService)
        return Error("Failed to open service");

    SERVICE_STATUS status;
    if (::QueryServiceStatus(hService, &status) &&
        status.dwCurrentState == SERVICE_RUNNING)
        StopService();

    if (::DeleteService(hService))
        printf("Service uninstalled successfully.\n");
    else
        Error("Failed to uninstall service");
    ::CloseServiceHandle(hService);
    ::CloseServiceHandle(hScm);
    return 0;
}
```

The function does some extra work, stopping the service if it's running by calling `QueryServiceStatus`, discussed later in this chapter.

To tie the various functions, the initial code in `main` changes to invoke a helper function if there are parameters on the command line:

```
int main(int argc, const char* argv[]) {
    if (argc > 1)
        return HandleCommandLine(argc, argv);
```

Here is the final piece:

```
int HandleCommandLine(int argc, const char* argv[]) {
    if (_stricmp(argv[1], "install") == 0) {
        return InstallService();
    }
    if (_stricmp(argv[1], "uninstall") == 0) {
        return UninstallService();
    }
    if (_stricmp(argv[1], "start") == 0) {
        return StartService();
    }
    if (_stricmp(argv[1], "stop") == 0) {
        return StopService();
    }
    return 0;
}
```

Service Status and Enumeration

The status of a service can be queried with the `QueryServiceStatus` and `QueryServiceStatusEx` functions:

```
BOOL QueryServiceStatus(
   _In_   SC_HANDLE         hService,
   _Out_  LPSERVICE_STATUS  lpServiceStatus);

BOOL QueryServiceStatusEx(
   _In_   SC_HANDLE         hService,
   _In_   SC_STATUS_TYPE    InfoLevel,
   _Out_  LPBYTE            lpBuffer,
   _In_   DWORD             cbBufSize,
   _Out_  LPDWORD           pcbBytesNeeded);
```

Both functions require a service handle with the `SERVICE_QUERY_STATUS` access mask. The older function retrieves a `SERVICE_STATUS` structure we've seen a few times in this chapter. The extended function is more generic, but currently only supports a structure of type `SERVICE_STATUS_PROCESS` we met as well.

`InfoLevel` must be `SC_STATUS_PROCESS_INFO` (the only value currently supported), with the aforementioned `SERVICE_STATUS_PROCESS` structure expected in the `lpBuffer` parameter. `cbBufSize` is the size of the buffer and `pcbBytesNeeded` returns the number of bytes used, or, if `NULL` is specified (and `cbBufSize` is zero), returns the number of bytes needed.

The *sc.exe* tool with the *query* command-line argument returns information about a service.

The service and drivers on the system can be enumerated by calling `EnumServicesStatus` or `EnumServicesStatusEx` where the status of services is returned with the same structures used for querying a service status, with two extra members for the service name and display name:

```
typedef struct _ENUM_SERVICE_STATUS {
    LPTSTR          lpServiceName;
    LPTSTR          lpDisplayName;
    SERVICE_STATUS  ServiceStatus;
} ENUM_SERVICE_STATUS, *LPENUM_SERVICE_STATUS;

BOOL EnumServicesStatus(
    _In_          SC_HANDLE              hSCManager,
    _In_          DWORD                  dwServiceType,
    _In_          DWORD                  dwServiceState,
    _Out_         LPENUM_SERVICE_STATUS  lpServices,
    _In_          DWORD                  cbBufSize,
    _Out_         LPDWORD                pcbBytesNeeded,
    _Out_         LPDWORD                lpServicesReturned,
    _Inout_opt_   LPDWORD                lpResumeHandle);

typedef struct _ENUM_SERVICE_STATUS_PROCESS {
    LPTSTR                  lpServiceName;
    LPTSTR                  lpDisplayName;
    SERVICE_STATUS_PROCESS  ServiceStatusProcess;
} ENUM_SERVICE_STATUS_PROCESS, *LPENUM_SERVICE_STATUS_PROCESS;

BOOL EnumServicesStatusEx(
    _In_          SC_HANDLE     hSCManager,
    _In_          SC_ENUM_TYPE  InfoLevel,
    _In_          DWORD         dwServiceType,
    _In_          DWORD         dwServiceState,
    _Out_         LPBYTE        lpServices,
    _In_          DWORD         cbBufSize,
    _Out_         LPDWORD       pcbBytesNeeded,
    _Out_         LPDWORD       lpServicesReturned,
    _Inout_opt_   LPDWORD       lpResumeHandle,
    _In_opt_      LPCTSTR       pszGroupName);
```

The enumeration functions bear a close resemblance to the status querying functions. The extended version is more generic, but only an array of `ENUM_SERVICE_STATUS_PROCESS` structures are accepted, identified by `InfoLevel` being set to `SC_ENUM_PROCESS_INFO`. The SCM handle provided must have the `SC_MANAGER_ENUMERATE_SERVICE` for the call to have a chance of succeeding.

`dwServiceType` specifies a "filter" used to return only certain of processes, if desired. These have roughly

the same values as a service type, but include two extra values to indicate all service (`SERVICE_WIN32`) and all drivers (`SERVICE_DRIVER`).

`dwServiceState` allows for further filtering by indicating if all services/drivers should be enumerated (`SERVICE_STATE_ALL`), or just those that are running (`SERVICE_ACTIVE`) or stopped (`SERVICE_INACTIVE`).

The buffer to return information in is specified by the `lpServices` parameter, and its size by `cbBufSize`. Since the size might be too small, `pcbBytesNeeded` returns the needed size. One way to deal with this is to call the function with a size of zero and get back the required size. In any case, the buffer cannot be larger than 256 KB.

The optional `lpResumeHandle` can be used to make multiple calls to the enumeration function in case not all service information has been returned by a single call. Finally, `EnumServicesStatusEx` allows further filtering with the last parameter specifying a group name of interest. If specified, only services/drivers in that group are enumerated.

The *enumsvc* Application

The *enumsvc* project demonstrates using the enumeration and status querying functions (using the extended versions only). Its purpose is to enumerate all services or show the state of a single service of choice.

The `main` function checks for a service name at the command-line, and if so diverts the code to `DisplayServiceStatus` to show the detials of a specific service:

```
int wmain(int argc, const wchar_t* argv[]) {
    if(argc > 1)
        return DisplayServiceStatus(argv[1]);
```

if there are no command-line arguments, `main` enumerates all services (but not drivers), running or not. First, a handle to the SCM must be opened:

```
auto hScm = ::OpenSCManager(nullptr, nullptr,
    SC_MANAGER_ENUMERATE_SERVICE);
if (!hScm)
    return Error("Failed to open handle to SCM");
```

The `Error` helper function is the usual one present in many of the demo applications.

Next, the enumeration itself:

```
auto size = 256 << 10;
auto buffer = std::make_unique<BYTE[]>(size);
DWORD returnedSize;
DWORD count;
if (!::EnumServicesStatusEx(hScm, SC_ENUM_PROCESS_INFO, SERVICE_WIN32,
    SERVICE_STATE_ALL, buffer.get(), size, &returnedSize, &count,
    nullptr, nullptr))
    return Error("Failed in enumeration");
```

The code simply allocates a 256 KB block dynamically using `std::make_unique<>` and then calls `EnumServicesStatusEx`, requesting all services, regardless of state.

All that's left to do is to cast the buffer to an array of `ENUM_SERVICE_STATUS_PROCESS` and display the information:

```
    // display title
    int len = printf("%-30s %-35s %-18s %-9s %6s %s\n",
        "Name", "Display Name", "Type", "State", "PID", "Accepted");
    printf("%s\n", std::string(len, '-').c_str());

    // cast to the correct type
    auto status = (ENUM_SERVICE_STATUS_PROCESS*)buffer.get();

    for (DWORD i = 0; i < count; i++) {
        auto& s = status[i];
        printf("%-30ws %-30ws %-18ws %-9ws %6d %ws\n",
            std::wstring(s.lpServiceName).substr(0, 30).c_str(),
            std::wstring(s.lpDisplayName).substr(0, 35).c_str(),
            ServiceTypeToString(
                s.ServiceStatusProcess.dwServiceType).c_str(),
            ServiceStateToString(
                s.ServiceStatusProcess.dwCurrentState),
            s.ServiceStatusProcess.dwProcessId,
            ServiceControlsAcceptedToString(
                s.ServiceStatusProcess.dwControlsAccepted).c_str());

    }

    ::CloseServiceHandle(hScm);

    return 0;
}
```

Most of the above code is about formatting and converting numbers to human-readable strings. The functions `ServiceTypeToString`, `ServiceStateToString`, and

`ServiceControlsAcceptedToString` are simple number-to-human string translations (see the details in the source code repository). Here is one of those functions:

```
std::wstring ServiceTypeToString(DWORD type) {
    static struct {
        DWORD type;
        PCWSTR text;
    } types[] = {
        { SERVICE_KERNEL_DRIVER, L"Kernel Driver" },
        { SERVICE_FILE_SYSTEM_DRIVER, L"FS/Filter Driver" },
        { SERVICE_WIN32_OWN_PROCESS, L"Own" },
        { SERVICE_WIN32_SHARE_PROCESS, L"Shared" },
        { SERVICE_INTERACTIVE_PROCESS, L"Interactive" },
        { SERVICE_USER_SERVICE, L"User" },
        { SERVICE_USERSERVICE_INSTANCE, L"Instance" },
    };

    std::wstring text;
    for (auto& item : types)
        if ((item.type & type) == item.type)
            text += std::wstring(item.text) + L", ";

    return text.substr(0, text.size() - 2);
}
```

Displaying the state of a single service is done by `DisplayServiceStatus` with the help of `GetServiceStatus`:

```
int DisplayServiceStatus(const wchar_t* name) {
    SERVICE_STATUS_PROCESS status;
    if (GetServiceStatus(name, status))
        return 1;

    printf("Service type: %ws\n",
        ServiceTypeToString(status.dwServiceType).c_str());
    printf("Service state: %ws\n",
        ServiceStateToString(status.dwCurrentState));
    if (status.dwCurrentState != SERVICE_STOPPED) {
        printf("Process ID: %u\n", status.dwProcessId);
    }
    printf("Controls accepted: %ws\n",
        ServiceControlsAcceptedToString(
            status.dwControlsAccepted).c_str());
```

```
    return 0;
}

int GetServiceStatus(PCWSTR name, SERVICE_STATUS_PROCESS& status) {
    auto hScm = ::OpenSCManager(nullptr, nullptr,
        SC_MANAGER_CONNECT);
    if (!hScm)
        return Error("Failed to open handle to SCM");

    auto hService = ::OpenService(hScm, name, SERVICE_QUERY_STATUS);
    if (!hService)
        return Error("Failed to open handle to service");

    DWORD size = sizeof(status);
    if (!::QueryServiceStatusEx(hService, SC_STATUS_PROCESS_INFO,
        (PBYTE)&status, sizeof(status), &size))
        return Error("Failed to query service status");

    ::CloseServiceHandle(hService);
    ::CloseServiceHandle(hScm);

    return 0;
}
```

`GetServiceStatus` is the one calling `QueryServiceStatusEx` to get a single `SERVICE_STATUS_PROCESS` back to the caller. `DisplayServiceStatus` uses that status and the helper translation functions to show the information. Note that only if the service is active (not in the stopped state), its process ID is shown (otherwise, zero is reported in `dwProcessId`).

Here is an example run with some services:

```
C:\>enumsvc alarmsvc
Service type: Own
Service state: Running
Process ID: 31764
Controls accepted: Stop

C:\>enumsvc bits
Service type: Own, Shared
Service state: Running
Process ID: 16252
Controls accepted: Stop, Pre Shutdown, Time Change
```

```
C:\>enumsvc appinfo
Service type: Own, Shared
Service state: Running
Process ID: 12744
Controls accepted: Stop, Session Change
```

Here is the initial output in the full enumeration case (broken up in a book page output):

```
C:\>enumsvc
Name                Display Name                    Type        State    \
 PID
Accepted
------------------------------------------------------------------------\
----
AdobeARMservice     Adobe Acrobat Update Service   Own         Running  \
6132
  Stop
AJRouter            AllJoyn Router Service         Shared      Stopped  \
   0
ALG                 Application Layer Gateway Serv Own         Stopped  \
   0
ApHidMonitorService Alps HID Monitor Service       Own         Running  \
6116
  Stop, Shutdown, Session Change
AppHostSvc          Application Host Helper Servic Own, Shared Running  \
6084
  Stop, Pause, Continue, Shutdown
AppIDSvc            Application Identity           Shared      Stopped  \
   0
Appinfo             Application Information        Own, Shared Running 1\
2744
  Stop, Session Change
```

Service Configuration

Installing a service or driver with `CreateService` configures various properties of the service, such as its display name, start type, user name, and more. However, not all possible configuration options are exposed via `CreateService`. Furthermore, most of the values in `CreateService` can be changed later on with `ChangeServiceConfig`:

```
BOOL ChangeServiceConfig(
    _In_        SC_HANDLE  hService,
    _In_        DWORD      dwServiceType,
    _In_        DWORD      dwStartType,
    _In_        DWORD      dwErrorControl,
    _In_opt_    LPCTSTR    lpBinaryPathName,
    _In_opt_    LPCTSTR    lpLoadOrderGroup,
    _Out_opt_   LPDWORD    lpdwTagId,
    _In_opt_    LPCTSTR    lpDependencies,
    _In_opt_    LPCTSTR    lpServiceStartName,
    _In_opt_    LPCTSTR    lpPassword,
    _In_opt_    LPCTSTR    lpDisplayName);
```

`ChangeServiceConfig` allows changing the parameters initially provided to `CreateService` (except the service name). The service handle must have the `SERVICE_CHANGE_CONFIG` access mask. The first three parameters (`dwServiceType`, `dwStartType` and `dwErrorControl`) can be set to the special value `SERVICE_NO_CHANGE` to indicate a configuration that should not change. Similarly, specifying `NULL` for `lpBinaryPathName` does not change the path to the service/driver binary. Specifying `NULL` for `lpLoadOrderGroup` does not change the group; specify and empty string to remove the group association. The same rule applies to the `lpDependencies` parameter.

The user account under which the service runs can be changed with the `lpServiceStartName` parameter and `lpPassword` (if applicable). Specifying `NULL` for `lpServiceStartName` does not make a change in the user account. Note that if the account is changed, it will take effect the next time the service is started. Finally, `lpDisplayName` is the new display name for the service (specify `NULL` to leave it unchanged). The name can also extract a localized resource by using the format *@path\DllName,-id* where *id* is a string resource ID in the DLL (the *path* part is optional). The display name is retrieved with `RegLoadMUIString` when needed (refer to the documentation for that function).

The configuration can be read with the similarly named `QueryServiceConfig` API:

```
typedef struct _QUERY_SERVICE_CONFIG {
    DWORD   dwServiceType;
    DWORD   dwStartType;
    DWORD   dwErrorControl;
    LPTSTR  lpBinaryPathName;
    LPTSTR  lpLoadOrderGroup;
    DWORD   dwTagId;
    LPTSTR  lpDependencies;
    LPTSTR  lpServiceStartName;
    LPTSTR  lpDisplayName;
} QUERY_SERVICE_CONFIG, *LPQUERY_SERVICE_CONFIG;

BOOL QueryServiceConfig(
    _In_  SC_HANDLE              hService,
```

```
    _Out_ LPQUERY_SERVICE_CONFIG lpServiceConfig,
    _In_  DWORD                  cbBufSize,
    _Out_ LPDWORD                pcbBytesNeeded);
```

> There is some overlap between "configuration" and "status" with services. For example, the service type is part of `QUERY_SERVICE_CONFIG` and `SERVICE_STATUS`.

The service handle must have the `SERVICE_QUERY_CONFIG` access mask. The function expects a `QUERY_SERVICE_CONFIG` that packs all the details (except the password and the service name) normally provided to `CreateService` and `ChangeServiceConfig`. Since the structure contains pointers to strings, a large enough buffer must be provided which is not a simple `sizeof` of the structure. As is the usual case, it's possible to specify zero for `cbBufSize`, and the function will return the count of needed bytes in `*pcbBytesNeeded`. In any case, the buffer cannot be larger than 8 KB.

We can extend the *svcenum* application from the previous section by listing some of the configuration data of the service when querying a specific service. Here is a helper function to return a service's configuration as a generic buffer:

```
std::unique_ptr<BYTE[]> GetServiceConfig(const wchar_t* name) {
    auto hScm = ::OpenSCManager(nullptr, nullptr,
        SC_MANAGER_CONNECT);
    if (!hScm)
        return nullptr;

    auto hService = ::OpenService(hScm, name, SERVICE_QUERY_CONFIG);
    if (!hService)
        return nullptr;

    auto size = 8 << 10;    // 8 KB
    auto buffer = std::make_unique<BYTE[]>(size);
    assert(buffer);

    DWORD needed;
    BOOL ok = ::QueryServiceConfig(hScm,
        reinterpret_cast<QUERY_SERVICE_CONFIG*>(buffer.get()),
        size, &needed);
    ::CloseServiceHandle(hService);
    ::CloseServiceHandle(hScm);

    return ok ? std::move(buffer) : nullptr;
}
```

Since the configuration size is unknown in advance, I decided to allocate a buffer with `std::make_unique<>` and pass it to to `QueryServiceConfig`. Then I just return it, freeing the caller from the responsibility of freeing it explicitly. Note the call to `std::move` to pass the ownership of the buffer to the caller. This is because `unique_ptr<>` maintains a single owner, and does not have a copy constructor (or copy assignment), having move constructor and move assignment only.

It's possible of course to use other mechanisms to pass a C-style buffer, similar to mechanisms used by the Windows API. Using C++ techniques can make the code safer and less error prone, however.

Here are the additions to `DisplayServiceStatus`:

```
auto buffer = GetServiceConfig(name);
if (buffer) {
    auto config = reinterpret_cast<QUERY_SERVICE_CONFIG*>(
            buffer.get());
    printf("Display name: %ws\n", config->lpDisplayName);
    printf("Image path: %ws\n", config->lpBinaryPathName);
    printf("Error control: %ws\n",
        ErrorControlToString(config->dwErrorControl));
}
```

Don't let `reinterpret_cast` scare you - it's the closest C++ cast to a C-style cast, when `static_cast` would fail compilation. A C-style cast would work just as well in the above case. Using `reinterpret_cast` means the developer paid special attention to the case in question and dimmed the cast correct, since the compiler has no way of verifying this. In most samples in this book I don't use `reinterpret_cast` to make things appear simpler.

`ChangeServiceConfig` and `QueryServiceConfig` cover only some of the configuration options of services. For all other configuration requirements, two other functions are used:

```
BOOL ChangeServiceConfig2(
    _In_      SC_HANDLE hService,
    _In_      DWORD     dwInfoLevel,
    _In_opt_  LPVOID    lpInfo);

BOOL QueryServiceConfig2(
    _In_   SC_HANDLE  hService,
    _In_   DWORD      dwInfoLevel,
    _Out_  LPBYTE     lpBuffer,
    _In_   DWORD      cbBufSize,
    _Out_  LPDWORD    pcbBytesNeeded);
```

`ChangeServiceConfig2` is **not** a superset of `ChangeServiceConfig`, covering different configuration aspects not covered by the latter. Similarly, `QueryServiceConfig2` is not a superset of `QueryServiceConfig`.

`dwInfoLevel` specifies the configuration property to change or query. `lpInfo` points to the relevant data structure to use to change the configuration. For query, the rest of the parameters are the usual bunch of a buffer, size, and returned size. Table 19-8 summarizes the values for `dwInfoLevel` and their meaning. Most are described in greater detail in subsequent subsections (`SERVICE_SID_INFO` and `REQUIRED_PRIVILEGES_INFO` are described in the section *Service Security*, later in this chapter).

Table 19-8: Confiuration options with **`ChangeServiceConfig2`/`QueryServiceConfig2`**

`dwInfoLevel` (`SERVICE_CONFIG_`)	Associated structure	Description
`DESCRIPTION` (1)	`SERVICE_DESCRIPTION`	Service description
`FAILURE_ACTIONS` (2)	`SERVICE_FAILURE_ACTIONS`	Actions to perform if/when the service crashes
`DELAYED_AUTO_START_INFO` (3)	`SERVICE_DELAYED_AUTO_START_INFO`	Delayed auto start
`FAILURE_ACTIONS_FLAG` (4)	`SERVICE_FAILURE_ACTIONS_FLAG`	Failure actions flag
`SERVICE_SID_INFO` (5)	`SERVICE_SID_INFO`	Service SID type
`REQUIRED_PRIVILEGES_INFO` (6)	`SERVICE_REQUIRED_PRIVILEGES_INFO`	Required privileges to add to the service
`PRESHUTDOWN_INFO` (7)	`SERVICE_PRESHUTDOWN_INFO`	Pre-shutdown timeout
`TRIGGER_INFO` (8)	`SERVICE_TRIGGER_INFO`	Trigger(s) to start/stop the service
`PREFERRED_NODE` (9)	`SERVICE_PREFERRED_NODE_INFO`	Preferred NUMA node
`LAUNCH_PROTECTED` (12)	`SERVICE_LAUNCH_PROTECTED_INFO`	Launch service as PPL (Windows 8.1+)

Service Description

This is the simplest configuration to understand from table 19-1. It allows changing the service description, normally visible in the *Services* applet. The structure in question just wraps a string pointer:

```
typedef struct _SERVICE_DESCRIPTION {
    LPTSTR       lpDescription;
} SERVICE_DESCRIPTION, *LPSERVICE_DESCRIPTION;
```

Failure Actions

One of the useful features of the SCM is its ability to relaunch a service if it crashes. This capability alleviates the need for creating some sort of "watchdog" process to watch over the service, and restart it if

it crashes. The SCM goes beyond just simple restart, however, encapsulated by the `SERVICE_FAILURE_ACTIONS` structure and related definitions:

```
typedef enum _SC_ACTION_TYPE {
    SC_ACTION_NONE        = 0,
    SC_ACTION_RESTART     = 1, // restart the service
    SC_ACTION_REBOOT      = 2, // reboot the machine
    SC_ACTION_RUN_COMMAND = 3, // run a command
    SC_ACTION_OWN_RESTART = 4  // restart service in its own process
} SC_ACTION_TYPE;

typedef struct _SC_ACTION {
    SC_ACTION_TYPE  Type;
    DWORD           Delay;
} SC_ACTION, *LPSC_ACTION;

typedef struct _SERVICE_FAILURE_ACTIONS {
    DWORD       dwResetPeriod;
    LPTSTR      lpRebootMsg;
    LPTSTR      lpCommand;
    DWORD       cActions;
    SC_ACTION * lpsaActions;
} SERVICE_FAILURE_ACTIONS, *LPSERVICE_FAILURE_ACTIONS;
```

The *Services* applet does provide a GUI for viewing and changing failure actions on the *Recovery* tab (figure 19-11). These are stored in a binary blob in the value *FailureActions* in the service's Registry key.

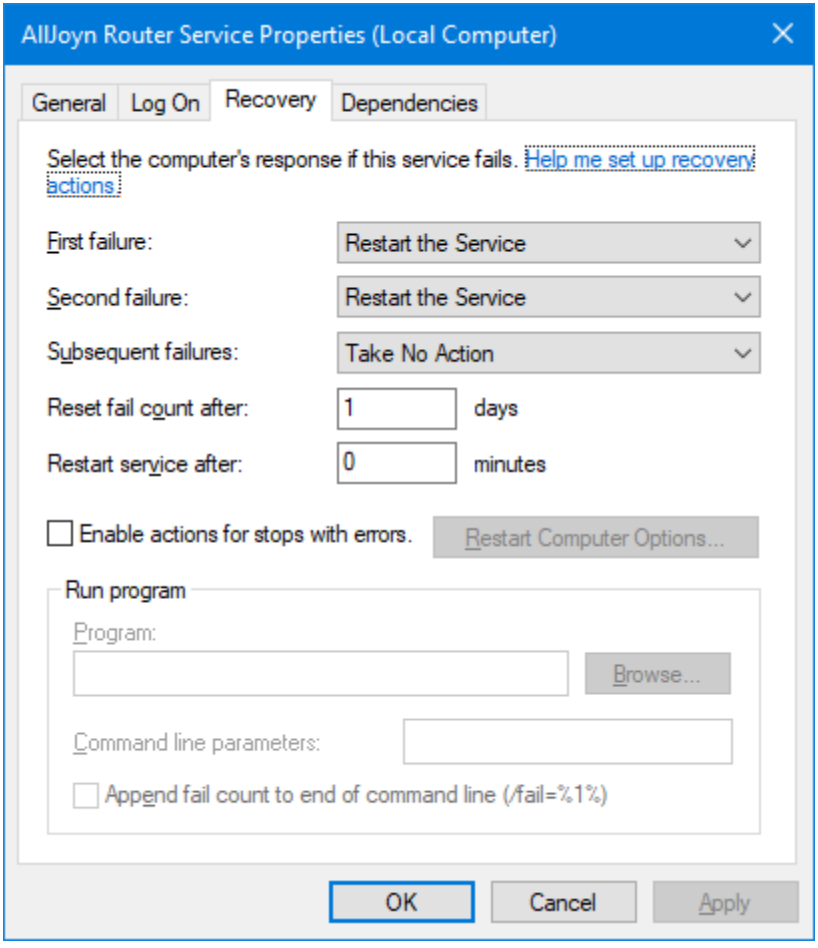


Figure 19-11: Failure actions in the *Services* applet

A service is considered failed when it terminates without reporting a `SERVICE_STOPPED` status to the SCM. Most of the time, this is due to an unhandled exception that causes the process to crash.

The SCM counts the number of times a service has failed since boot time. `dwResetPeriod` is the number of seconds of non-failure of the service, after which the failure count is reset to zero. A value of `INFINITE` indicates no reset should occur.

If the service fails, the actions specified by the array `lpsaActions` (whose count is `cActions`) is consulted for the action to take. The first entry indicates the first failure action, the second indicates the action to take on the second failure (assuming no count reset), and so on. If the failure count goes beyond `cActions`, the last action is repeated.

The *Services* applet *Recovery* tab shown in figure 19-11 only allows to set three actions, but you can set more programmatically if that makes sense to your service.

`lpRebootMsg` is a message to broadcast to other users logged on the machine in case the action to be carried out is a system reboot. Specifying `NULL` does not change thre previous message, and an empty string removes any message. The message can also specified in the same format as the display name in `ChangeServiceConfig` described in the previous section.

`lpCommand` is the process command line to launch (if the action type is `SC_ACTION_RUN_COMMAND`). The process is launched with the user account as the service. Just like `lpRebootMsg`, specifying `NULL` does not change the program, and an empty string removes it.

The `SC_ACTION` structure encapsulates the action to take. The action is specified with the `SC_ACTION_TYPE` enumeration. Most values are self-explanatory, except `SC_ACTION_OWN_RESTART`. This is undocumented for some reason, but it means the service should start in its own process if it was sharing a process with other services. This could be useful for an important service that may have failed because another service in the process caused a crash.

The `Delay` member of `SC_ACTION` indicates the delay in milliseconds before the specified action is taken.

A related setting to failure actions, is the service action flag, specified by the following structure, which is just a glorified Boolean value:

```
typedef struct _SERVICE_FAILURE_ACTIONS_FLAG {
    BOOL  fFailureActionsOnNonCrashFailures;
} SERVICE_FAILURE_ACTIONS_FLAG, *LPSERVICE_FAILURE_ACTIONS_FLAG;
```

The Boolean indicates when failure actions apply. "Non-crash failures" refers to the case where the service did report a stopped state, but the the `dwWin32ExitCode` member of `SERVICE_STATUS` indicates a Win32 error code (non-zero). The default value is `FALSE`, so that only true crashes trigger failure actions. Specifying `TRUE` activates failure actions even if the service reports stopping but `dwWin32ExitCode` indicates an error.

Pre-Shutdown Information

If a service accepts a pre-shutdown notification (`SERVICE_ACCEPT_PRESHUTDOWN` bit set as part of `dwControlsAccepted` in `SERVICE_STATUS`), it is notifed in its control code handler with a `SERVICE_CONTROL_PRESHUTDOWN` code. Normally, such a service has 3 minutes to complete its shutdown procedures before indicating it has stopped (by calling the normal `SetServiceStatus` with `SERVICE_STOPPED` state).

This timeout can be changed with the following pre-shutdown-related structure, another glorified wrapper, this time around a `DWORD`:

```
typedef struct _SERVICE_PRESHUTDOWN_INFO {
    DWORD  dwPreshutdownTimeout;    // Timeout in msecs
} SERVICE_PRESHUTDOWN_INFO, *LPSERVICE_PRESHUTDOWN_INFO;
```

`dwPreshutdownTimeout` indicates the timeout in milliseconds to use (180000 being the default of 3 minutes).

Delayed Auto-Start

Delayed auto-start allows an auto-start service to be "delayed" for about 2 minutes before actually starting, so that there is less contension for CPU or disk resources when the system boots. Enabling delayed auto start requires the following glorified Boolean wrapping structure:

```
typedef struct _SERVICE_DELAYED_AUTO_START_INFO {
    BOOL  fDelayedAutostart;      // Delayed autostart flag
} SERVICE_DELAYED_AUTO_START_INFO;
```

Trigger Information

There is yet another way for a service to start or stop - by receiving certain trigger(s). A trigger-start service is an efficient way to manage a service lifetime. For example, a service that needs to work with Active Directory does need to start if the computer is not connected to a domain controller. One way to handle this is to load the service automatically, and have it sit idley by until such a connection occurrs (if at all). Clearly, such a service still consumes memory and at least one thread. A better solution is for the SCM to load the service only if the machine is connected to a domain controller. This is the purpose of a trigger - connection to a domain is one example of a possible trigger.

The *Services* applet does not provide any GUI to show or modify triggers. The onlu hint of their existence is in the startup type show in the full list where some services have the value *Manual (Trigger Start)* in the *Startup Type* column. You can query a service for its triggers, if any, with *sc.exe* like so:

```
C:\>sc qtriggerinfo appinfo
[SC] QueryServiceConfig2 SUCCESS

SERVICE_NAME: appinfo

START SERVICE
    NETWORK EVENT  : bc90d167-9470-4139-a9ba-be0bbbf5b74d
        [RPC INTERFACE EVENT]
    DATA           : 201ef99a-7fa0-444c-9399-19ba84f12a1a
START SERVICE
    NETWORK EVENT  : bc90d167-9470-4139-a9ba-be0bbbf5b74d
        [RPC INTERFACE EVENT]
    DATA           : 5f54ce7d-5b79-4175-8584-cb65313a0e98
START SERVICE
    NETWORK EVENT  : bc90d167-9470-4139-a9ba-be0bbbf5b74d
        [RPC INTERFACE EVENT]
    DATA           : fd7a0523-dc70-43dd-9b2e-9c5ed48225b1
START SERVICE
    NETWORK EVENT  : bc90d167-9470-4139-a9ba-be0bbbf5b74d
        [RPC INTERFACE EVENT]
```

```
    DATA            : 58e604e8-9adb-4d2e-a464-3b0683fb1480
START SERVICE
    NETWORK EVENT   : bc90d167-9470-4139-a9ba-be0bbbf5b74d
        [RPC INTERFACE EVENT]
    DATA            : 0497B57D-2E66-424F-A0C6-157CD5D41700

C:\>sc qtriggerinfo alarmsvc
[SC] QueryServiceConfig2 SUCCESS

The service alarmsvc has not registered for any start or
    stop triggers.
```

You can get a holistic view of services and their triggers (if any) with my *System Explorer* tool. Select *System/Services* from the main menu, click *Columns* and check the *Triggers* item, and click *OK*. You should see something like figure 19-12 (sorted by the *Triggers* column with some other columns hidden).

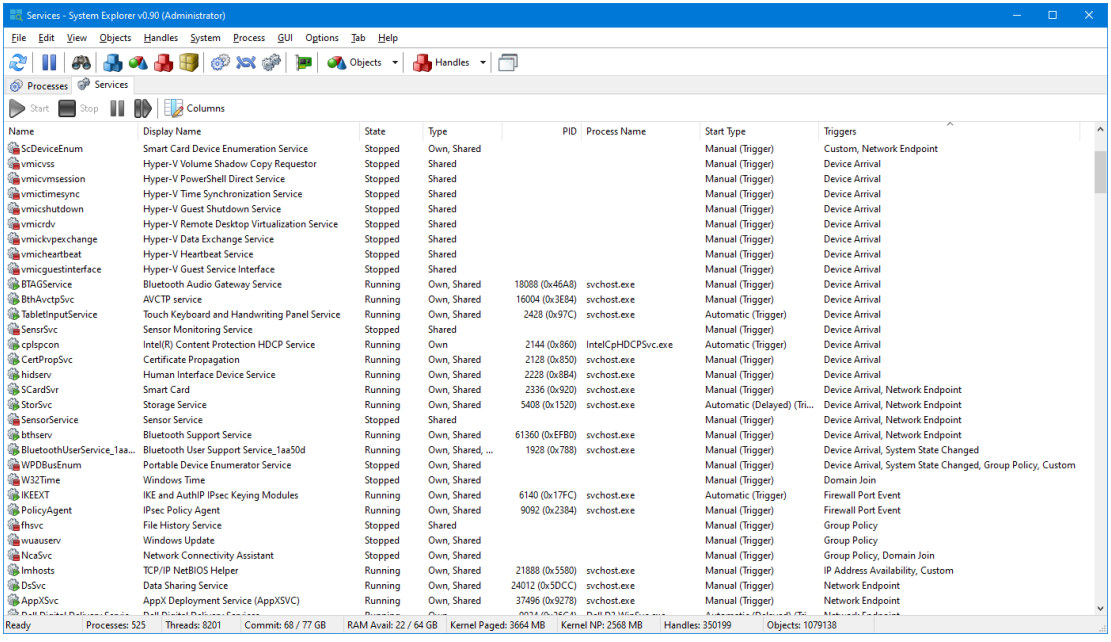


Figure 19-12: Triggers in *System Explorer*

Setting or querying for triggers requires the following structures:

```
typedef struct _SERVICE_TRIGGER_SPECIFIC_DATA_ITEM {
    DWORD   dwDataType; // Data type (SERVICE_TRIGGER_DATA_TYPE_*)
    DWORD   cbData;     // Size of trigger specific data
    PBYTE   pData;      // Trigger specific data
} SERVICE_TRIGGER_SPECIFIC_DATA_ITEM;

//  Trigger-specific information
typedef struct _SERVICE_TRIGGER {
    DWORD dwTriggerType;   // SERVICE_TRIGGER_TYPE_* constants
    DWORD dwAction;        // SERVICE_TRIGGER_ACTION_* constants
    GUID* pTriggerSubtype; // Provider GUID if the trigger type
                    // is SERVICE_TRIGGER_TYPE_CUSTOM
                    // Device class interface GUID
                    // if the trigger type is
                    // SERVICE_TRIGGER_TYPE_DEVICE_INTERFACE_ARRIVAL
                    // Aggregate identifier GUID if type is aggregate
    DWORD cDataItems; // Number of data items in pDataItems array
     // Trigger specific data
    PSERVICE_TRIGGER_SPECIFIC_DATA_ITEM  pDataItems;
} SERVICE_TRIGGER, *PSERVICE_TRIGGER;

// Service trigger information
typedef struct _SERVICE_TRIGGER_INFO {
    DWORD              cTriggers;  // Number of triggers
    PSERVICE_TRIGGER   pTriggers;  // Array of triggers
    PBYTE              pReserved;  // Reserved, must be NULL
} SERVICE_TRIGGER_INFO, *PSERVICE_TRIGGER_INFO;
```

The structures are annotated nicely. `SERVICE_TRIGGER_INFO` is the top level structure provided to `ChangeServiceConfig2` (and expected by `QueryServiceConfig2`). The number of triggers is indicated by the `cTriggers` member, followed by an array of `SERVICE_TRIGGER` instances, each describing a single trigger. Each trigger has a trigger type spcified in the `dwTriggerType` member. Table 19-9 lists the trigger types, and their meaning.

Table 19-9: Trigger types

Type (`SERVICE_TRIGGER_TYPE_`)	Description
`DEVICE_INTERFACE_ARRIVAL` (1)	Device arrival into the system
`IP_ADDRESS_AVAILABILITY` (2)	TCP/IP IP address available
`DOMAIN_JOIN` (3)	Machine joined a domain
`FIREWALL_PORT_EVENT` (4)	Firewall port open or closed
`GROUP_POLICY` (5)	Machine policy or user policy changed

Table 19-9: Trigger types

Type (SERVICE_TRIGGER_TYPE_)	Description
NETWORK_ENDPOINT (6)	Packet arrives on a particular network interface (Windows 8+)
CUSTOM (20)	Custom event base on *Event Tracing for Windows* (ETW)

The dwAction member indicates what to do if the trigger is satisfied - start the service (SERVICE_TRIGGER_ACTION_SERVICE_START) or stop it (SERVICE_TRIGGER_ACTION_SERVICE_STOP). pTriggerSubStype is a GUID whose meaning depends on the type of trigger. For example, SERVICE_TRIGGER_TYPE_NETWORK_ENDPOINT requires one of two GUIDs indicating whether to loop for IP availability or lack thereof:

- NETWORK_MANAGER_FIRST_IP_ADDRESS_ARRIVAL_GUID for availability.
- NETWORK_MANAGER_FIRST_IP_ADDRESS_REMOVAL_GUID for unavailability (triggered after one minute of unavailability).

The documentation lists other GUIDs relevant to the various trigger types. For SERVICE_TRIGGER_TYPE_CUSTOM, any ETW provider GUID can be used (should be with a single event). See chapter 20 for more on ETW.

Certain types of triggers require even more information, provided by the cDataItems and pDataItems members of SERVICE_TRIGGER. These point to an array of SERVICE_TRIGGER_SPECIFIC_DATA_ITEM, each holding a single value, that can be one of the following types: binary blob, string, byte (8-bit) value, or a 64-bit number. Refer to the documentation for the full details.

The following code snippet shows how to get trigger information for a service.

```
bool DisplayTriggers(SC_HANDLE hService) {
    DWORD len;
    //
    // get required size
    //
    ::QueryServiceConfig2(hService, SERVICE_CONFIG_TRIGGER_INFO,
        nullptr, 0, &len);
    if (::GetLastError() != ERROR_INSUFFICIENT_BUFFER)
        return false;

    //
    // allocate required size
    //
    auto buffer = std::make_unique<BYTE[]>(needed);
    if (!::QueryServiceConfig2(hService, SERVICE_CONFIG_TRIGGER_INFO,
```

```
        buffer.get(), len, &len))
        return false;

    auto info = reinterpret_cast<SERVICE_TRIGGER_INFO*>(
        buffer.get());
    WCHAR sguid[64];
    for (DWORD i = 0; i < info->cTriggers; i++) {
        const auto& tinfo = info->pTriggers[i];
        printf("Trigger %u\n", i);
        printf("  Type: %s\n",
            TriggerTypeToString(tinfo.dwTriggerType));
        printf("  Action: %s\n", tinfo.dwAction ==
            SERVICE_TRIGGER_ACTION_SERVICE_START ? "Start" : "Stop");
        if(::StringFromGUID2(*tinfo.pTriggerSubtype,
            sguid, _countof(sguid)))
            printf("  GUID: %ws\n", sguid);
    }

    return true;
}
```

`TriggetTypeToString` is a helper function to convert the type to a human-readable string. The code has been added to the *svcenum* project.

Extend the above code to display full details of triggers.

Preferred NUMA Node

Recall from chapter 3 (in part 1), that the preferred NUMA node for a process can be specified using a process attribute when calling `CreateProcess`. Since a service process is launched by the SCM, there is no direct way to set any attribute. In the case of NUMA node, `ChangeServiceConfig2` can be used by supplying the following structure:

```
typedef struct _SERVICE_PREFERRED_NODE_INFO {
    USHORT   usPreferredNode;    // Preferred node
    BOOLEAN  fDelete;            // Delete the preferred node setting
} SERVICE_PREFERRED_NODE_INFO, *LPSERVICE_PREFERRED_NODE_INFO;
```

Launch as PPL

Recall from chapter 3 (in part 1), that processes that are signed appropriately can be launched as protected or *Protected Process Light* (PPL). For some built-in services and third party services, mostly in the anti-malware category, Microsoft allows running these as PPL so that the service cannot be stopped or terminated, even with admin privileges. The executable must be signed by Microsoft, however, to get that capability. Configuring the service to run as PPL requires the following structure:

```
typedef struct _SERVICE_LAUNCH_PROTECTED_INFO {
    DWORD  dwLaunchProtected;     // Service launch protected
} SERVICE_LAUNCH_PROTECTED_INFO, *PSERVICE_LAUNCH_PROTECTED_INFO;
```

`dwLaunchProtected` can be one of the following:

- `SERVICE_LAUNCH_PROTECTED_NONE` (0) - run unprotected.
- `SERVICE_LAUNCH_PROTECTED_WINDOWS` (1) - run protected in the Windows signer level (see chapter 3).
- `SERVICE_LAUNCH_PROTECTED_WINDOWS_LIGHT` (2) - run protected light in the Windows signer level.
- `SERVICE_LAUNCH_PROTECTED_ANTIMALWARE_LIGHT` (3) - run protected light in the Antimalware signer level.

The values 1 and 2 can only be used by Microsoft services. Third party services can only utilize the value 3, if signed appropriately.

Once a service runs protected, it's not just protected from being stopped or terminated, but the following APIs automatically fail: `ChangeServiceConfig(2)`, `ControlService(Ex)`, `DeleteDevice`, and `SetServiceObjectSecurity`.

Debugging Services

On the one hand, debugging services is no different than debugging any other user-mode process. There are a few potential issues, however. If the normal activity of a service is to be debugged, a debugger launched elevated can attach to the process. Running elevated is required in cases where the service is running under one of the special service accounts, which is the case for most services. Such attachment works as expected, and the service can be debugged normally.

There are a few points that need to be taken into consideration:

- The service process is not launched directly by the debugger - it's always launched by the SCM. If the initialization code of the service needs to be debugged, then it's a problem, since by the time the debugger attaches to the process, that code would already be done.
- Any time the SCM waits for a service status notification (`SetServiceStatus`), it won't wait forever - it waits for up to 30 seconds by default. When this times out (because the developer has a breakpoint set and he/she is actively debugging), the SCM may terminate the process, concluding that the service is unresponsive.

- Reporting errors with console functions (e.g. `printf`) does not work for most services, since these run in session 0 (against a non-interactive Window Station), rather than the user's session.

Here are some ideas to get around these issues:

- Sometimes it's possible to construct the code in such a way that the execution of the `main` function can be done just like a standard process so that normal breakpoints can be used. This is sometimes not good enough, because the service would run with the logged on user rather than the real user (e.g. *LocalSystem*), which means the conditions are not the same. Still, this works in some cases.
- You can set an artificial delay in the `main` function so that the developer has enough time to attach the debugger to the process. Here is one way to do that:

```
if(::IsDebuggerPresent())    // is running under a debugger?
    ::Sleep(20000);          // allow 20 seconds for attaching
```

- The timeout the SCM uses is configurable using the Registry. If you create a value named *ServicesPipeTimeout* in the key *HKLM\SYSTEM\CurrentControlSet\Control* and set it to the required timeout in milliseconds (`DWORD` value), that value will be used after the next system boot (unfortunately, the SCM reads the value when it launches only).
- Using `printf` or similar is problematic. Here are other ways a service can report (errors or other information):
  - Call `OutputDebugString` to dump information to the debugger's output if connected (or another capable tool such as *DbgView* from *Sysinternals*). (See chapter 20 for more on `OutputDebugString`).
  - Write information to some log, such as a file.
  - Write events to the Event Log service (see chapter 20 for more on the event log).

Interactive Services

In pre-Windows 10 versions of Windows, a service running under the *LocalSystem* account could run in the user's session by configuring it as an *Interfactive Service*, by adding the `SERVICE_INTERACTIVE_PROCESS` (0x100) bit to the service type. The *Services* applet provides this option in its UI as well (figure 19-13).

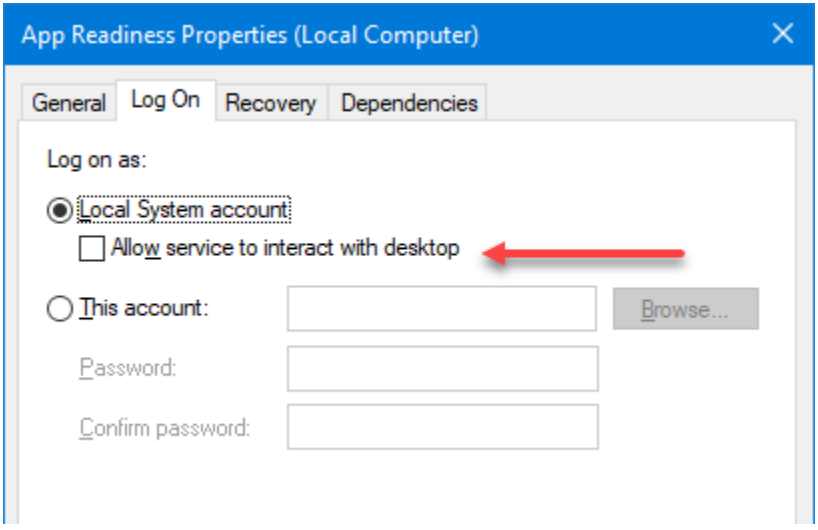


Figure 19-13: Intercative service setting in the *Services* applet

Such a service is theoretically easier to debug, as it executes in the same session as the logged on user. This features does not work anymore on Windows 10 and later systems. The original issue is related to security: a service running with the all-powerful *LocalSystem* account in a user's session is much more volnurable to attacks than if it runs in session 0.

Bottom line: forget about interactive services - it is not a viable option in modern Windows. If a service must run under a user's session, it can do so with *per-user services* (described later in this chapter).

Service Security

Most services run under the three service accounts (*LocalSystem*, *LocalService* and *NetworkService*), giving them certain predefined set of privileges. The privilege set of such services can be specified more explicitly, so that only the required services are available, rather than possibly a larger set that enlarges the attack surface of the service.

Calling `ChangeServiceConfig2` with `SERVICE_CONFIG_REQUIRED_PRIVILEGES_INFO` allows specifying a list of privileges as multiple null-terminated strings (just like a `REG_MULTI_SZ` Registry value). You can view configured privileges requested by a service using *sc.exe* or *System Explorer* (figure 19-14, with the *Privileges* column visible).

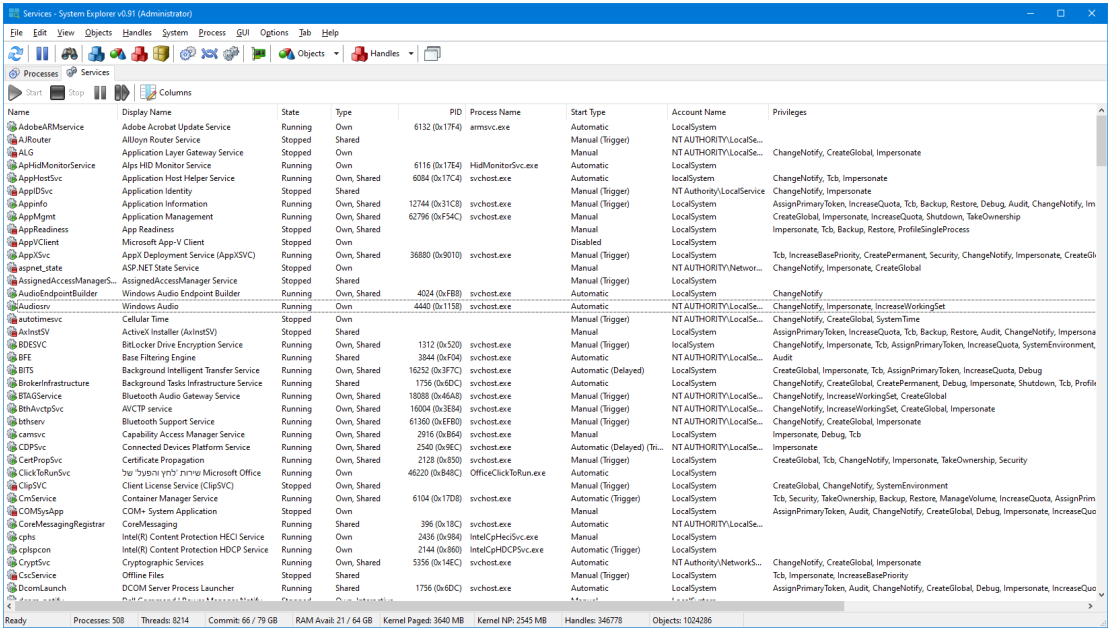


Figure 19-14: Requested privileges in *System Explorer*

The *svcpriv* application allows configuring a service with a set of privileges. Here is the `main` function:

```
int wmain(int argc, const wchar_t* argv[]) {
    if (argc < 3) {
        printf("Usage: svcpriv <servicename> <privilege1> [...]\n");
        return 0;
    }

    auto hScm = ::OpenSCManager(nullptr, nullptr,
        SC_MANAGER_ALL_ACCESS);
    if (!hScm)
        return Error("Failed to open handle to SCM");

    auto hService = ::OpenService(hScm, argv[1],
        SERVICE_CHANGE_CONFIG);
    if (!hService)
        return Error("Failed to open service handle");

    SERVICE_REQUIRED_PRIVILEGES_INFO info;
    WCHAR buffer[1024];
    size_t index = 0;
    for (int i = 2; i < argc; i++) {
        ::wcscpy_s(buffer + index, _countof(buffer) - index, L"Se");
```

```
            ::wcscat_s(buffer + index, _countof(buffer) - index,
                argv[i]);
            ::wcscat_s(buffer + index, _countof(buffer) - index,
                L"Privilege");
            index += ::wcslen(argv[i]) + 1 + ::wcslen(L"SePrivilege");
        }
        buffer[index] = 0;
        info.pmszRequiredPrivileges = buffer;
        if(!::ChangeServiceConfig2(hService,
            SERVICE_CONFIG_REQUIRED_PRIVILEGES_INFO, &info))
            return Error("Failed to change config");

        printf("Configuration changed successfully.\n");

        ::CloseServiceHandle(hScm);
        ::CloseServiceHandle(hService);

        return 0;
}
```

Most of the work done in the above code is building the multiple string value to put in the `SERVICE_REQUIRED_PRIVILEGES_INFO` structure before calling `ChangeServiceConfig2`. The privilege names expected in the command line should not include the "Se" prefix nor the "Privilege" suffix. These are added by the multi-string building code. Here is an example invocation:

```
c:\>svcpriv alarmsvc debug tcb createglobal
```

As with everything that has to do with service configuration, the value is written to the Registry service's key in the *RequiredPrivileges* value. Naturally, it only takes effect the next time the service is restarted.

The *SeChangeNotifyPrivilege* privilege is always granted, regardless of any configuration.

Service SID

Using one of the built-in service accounts is sometimes not fine-grained enough for setting permissions. For example, what if you wanted a specific service to have access to some file, but other services should not have it? If the service runs under one of the built-in service accounts (e.g. *LocalSystem*), then specifying that account's SID in an ACE would mean that any service running with that account would have access as well.

One way to mitigate that is to create a dedicated user account to serve as an identity for the service. However, this is troublesome and requires extra user management. Compound that by multiple services that may have similar requirements, and you get a management nightmare.

Fortunately, Windows 7 introduced the concept of service virtual accounts. Each service has its own account that can be leveraged. The account name is in the form *NT SERVICE\servicename*. This means that permissions can be set to target a specific service using the SID corresponding to this account. This SID is generated by using a SHA-1 hash on the service name, so that uniqueness is statistically guaranteed. You can get the SID of any service with the standard `LookupAccountName` API from chapter 16, where the "username" is the aforementioned account name. *System Explorer* shows these SIDs if you add the *Service SID* column (figure 19-15).

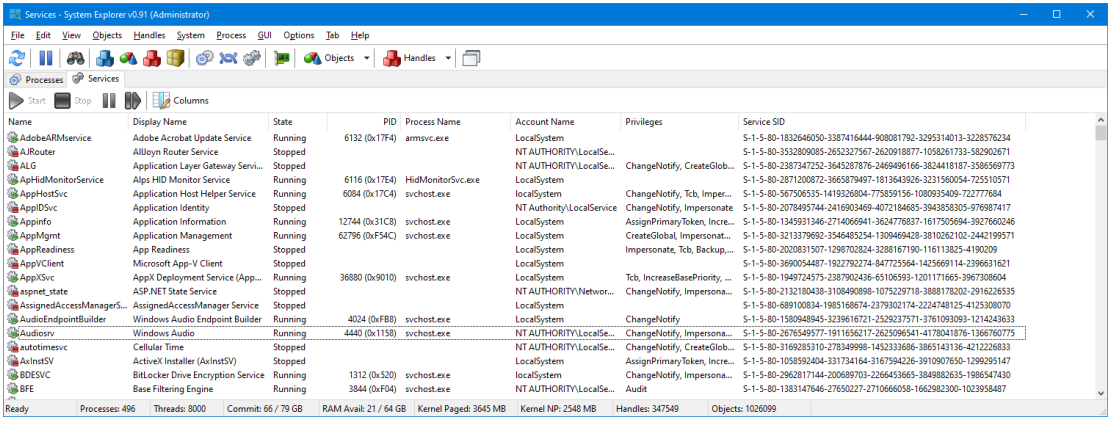


Figure 19-15: Services SID in *System Explorer*

A related configuration offered by `ChangeServiceConfig2` is the service SID type. This setting requires the `SERVICE_SID_INFO` structure that wraps a single `DWORD` value that indicates the type of SID to use for the service:

```
typedef struct _SERVICE_SID_INFO {
    DWORD  dwServiceSidType;     // Service SID type
} SERVICE_SID_INFO, *LPSERVICE_SID_INFO;
```

The default of `SERVICE_SID_TYPE_NONE` does nothing. The other two options (`SERVICE_SID_TYPE_RESTRICTED` and `SERVICE_SID_TYPE_UNRESTRICTED`) make some changes to the process token. Refer to the documentation for the full details. *System Explorer* can show the configuration of the SID type as well in the *SID Type* column.

Service Security Descriptor

By default, an admin account can open a service for any and all access. If restricting service access is desired, `SetServiceObjectSecurity` can be used to manipulate the security descriptor of the service itself:

```
BOOL SetServiceObjectSecurity(
    _In_  SC_HANDLE             hService,
    _In_  SECURITY_INFORMATION  dwSecurityInformation,
    _In_  PSECURITY_DESCRIPTOR  lpSecurityDescriptor);
```

The function is very similar to other security descriptor manipulation APIs described in chapter 16. The opposite function to query the security descriptor is available as well:

```
BOOL QueryServiceObjectSecurity(
    _In_   SC_HANDLE             hService,
    _In_   SECURITY_INFORMATION  dwSecurityInformation,
    _Out_  PSECURITY_DESCRIPTOR  lpSecurityDescriptor,
    _In_   DWORD                 cbBufSize,
    _Out_  LPDWORD               pcbBytesNeeded);
```

Process Explorer can show the security descriptor of a service (figure 19-16).

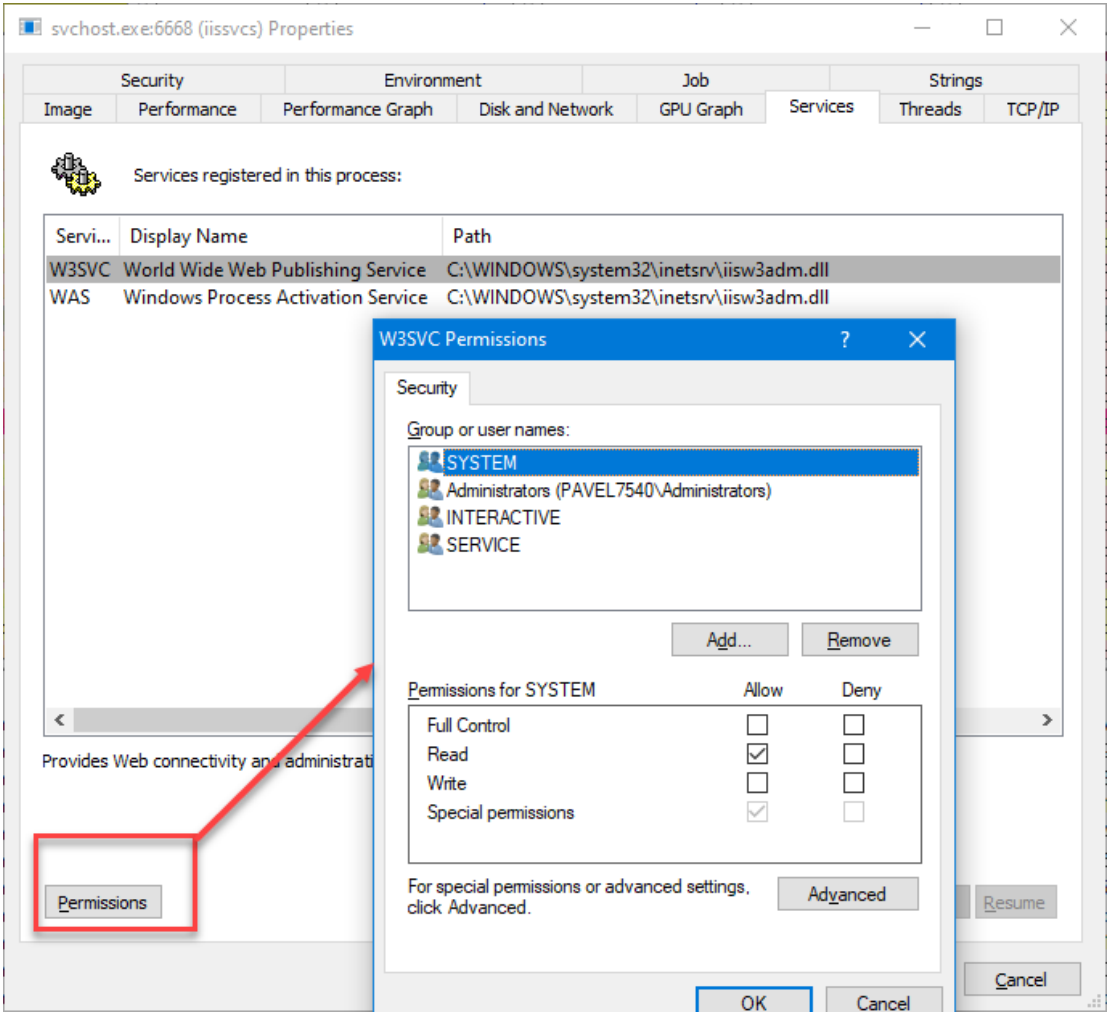


Figure 19-16: Service security descriptor *Process Explorer*

Per-User Services

Services are traditionally "single instance" - there is just one service instance running regardless of the number of users logged in concurrently to a system. In most cases, this is what is required, as a service is regarded as a system-wide facility.

In some cases, it's useful to have a service run under a logged on user's account and in the session of that user, to serve just that particular user. Windows 10 introduced *Per-User Services*, that provide such behavior. If a service is configured as per-user (by using one of the following values for the service type: `SERVICE_USER_OWN_PROCESS` or `SERVICE_USER_SHARE_PROCESS`), each user logging in to the system gets a service on its own, running in its session under that user. The name of that service is generated by taking

the base name (refered to as "template"), and appending a hex value based on the SHA-1 of the username and the service name.

These services are easily identifiable because of that special suffix, and are visible in the Registry (both template and specific instance), the *Services* applet (only the instances) and other tools (such as *Task Manager*, *Process Explorer*, *System Explorer*). Figure 19-17 shows the view in *Task Manager*. Notice the per-user services, where the template service is shown as well, but can never be started directly.

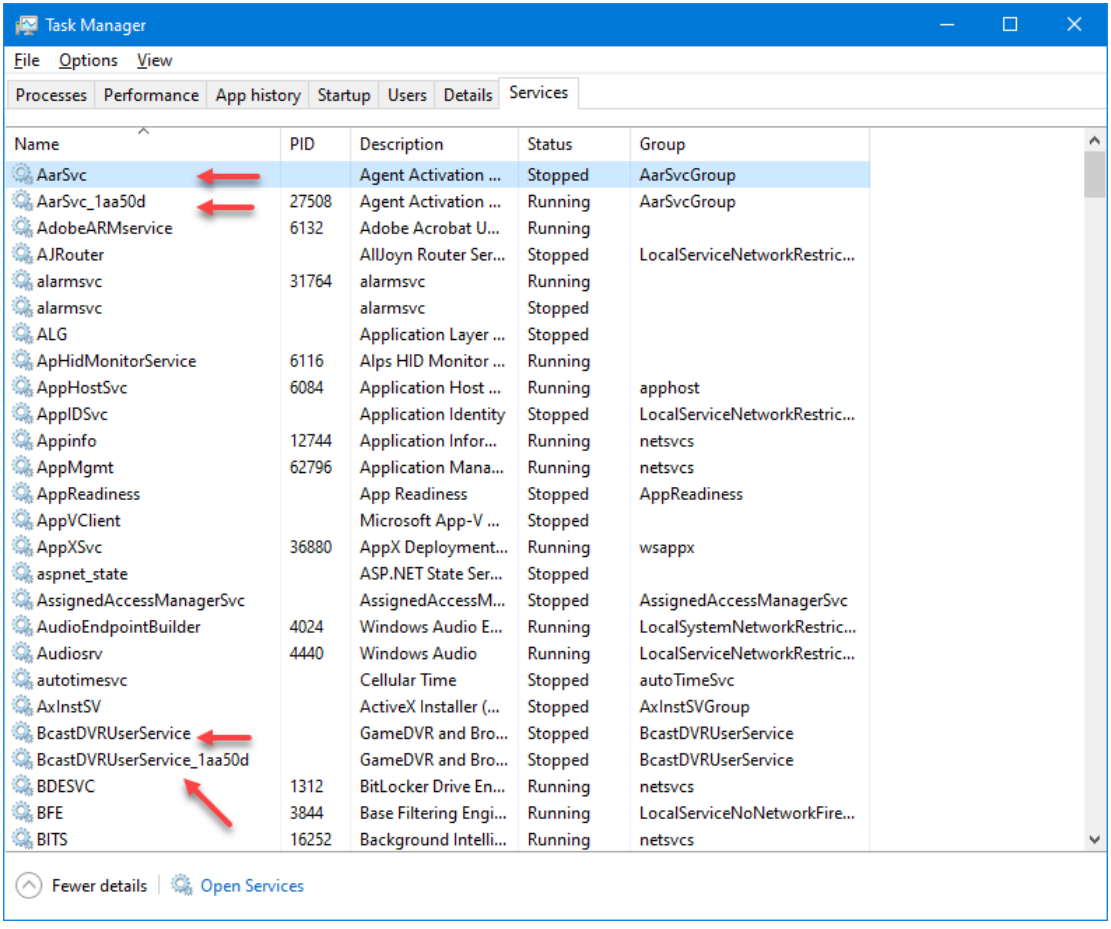


Figure 19-17: Services in *Task Manager*

The *Services* applet has a bug (at the time of writing) where it shows the *LocalSystem* account in the *Log On As* column for per-user services, rather than the correct user account.

Miscellaenous Functions

There are a few more functions worth mentioning related to services. `GetServiceDisplayName` retrieves a service display name given its name. The opposite function, `GetServiceKeyName` returns a service name given its display name.. Here are their definitions:

```
BOOL GetServiceDisplayNameA(
    _In_    SC_HANDLE hSCManager,
    _In_    LPCSTR    lpServiceName,
    _Out_   LPSTR     lpDisplayName,
    _Inout_ LPDWORD   lpcchBuffer);
BOOL GetServiceKeyNameA(
    _In_    SC_HANDLE hSCManager,
    _In_    LPCSTR    lpDisplayName,
    _Out_   LPSTR     lpServiceName,
    _Inout_ LPDWORD   lpcchBuffer);
```

`NotifyBootConfigStatus` allows a service running in the *LocalSystem* account (or another admin account) to report the success or failure of this boot sequence. If `TRUE` is specified, the current configuration is saved as "Last Known Good". This capablity is rarely needed by most services.

Summary

Services are critical to many capabilities provided by Windows. Third-party services can enjoy the ability to load automatically, manage failures, run under several built-in accounts, with any needed privileges to do their work. We've seen how to build a service and a client, how to configure it, and manage it. In the next chapter, we'll turn our attention to debugging and diagnostics capabilities developers can use to get visibility into what's going on inside their processes.

Chapter 20: Debugging and Diagnostics

Debugging is usually thought of as the process of fixing code defects using a specialized tool - a *Debugger*. However, not all problems can be solved by simply setting breakpoints, examining variables, looking at call stacks, memory, threads, etc. Sometimes the application produces correct results, but it may be too slow in producing them; or it might exhibit memory or other resource leaks. In such cases, using a debugger is seldom useful, requiring other means of tackling such issues.

In this chapter, we'll examine some mechanisms provided by Windows through the Windows API for gaining a deeper insight into what a process is doing. Such mechanisms are useful in a development environment, but even more so in production environments, where the execution is with "real" data and load.

In this chapter:

- **Debugger Output**
- **Performance Counters**
- **Process Snapshots**
- **Event Tracing for Windows**
- **Trace Logging**
- **Debuggers**

Debugger Output

Perhaps the simplest mechanism available to processes to describe operations being performed is by sending textual output to any tool that can "capture" it. Windows provides the `OutputDebugString` function for this purpose:

```
void OutputDebugString(_In_opt_ LPCTSTR lpOutputString);
```

The function is the definition of simplicity itself - just provide a string and that's it. In most cases, you would like the string to consist of variable values, or other program state, which would require building the string with other functions, such as the classic `sprintf_s`, or the Windows API variants such as `StringCchPrintf(Ex)`.

If you're working with C++, C++ 20 has the convenient and powerful `std::format` function for building such strings. At the time of this writing, the Visual C++ libraries do not support it, but by the time you read this, they probably will.

Regardless of how the string is built, these are the steps taken by `OutputDebugString`:

- If the process is being debugged, the text is sent to the debugger (see the section *Debuggers*, later in this chapter, describing how debuggers capture the text).
- Otherwise, two named events are opened by calling the standard `OpenEvent` function, one named "DBWIN_BUFFER_READY" and the other named "DBWIN_DATA_READY". If either or both are not found, the text is simply dropped. (See the next subsection for a detailed description of how these events are used.)
- If the events exist, a named Memory Mapped File is opened by calling the standard

`OpenFileMapping` function with the name "DBWIN_BUFFER". If not found, the text is dropped.

- Finally, if all three objects exist, `OutputDebugString` calls `MapViewOfFile` to map the MMF, and writes the process ID to the mapped memory (as `DWORD`) and then writes the text following the PID, with a `NULL` terminator.

The above description implies that an external process can capture output from other processes that use `OutputDebugString`, if these are not currently being debugged. This turns out to be very useful, as it allows capturing output from multiple processes in a single location. A useful tool that provides this capability is *DbgView* from *Sysinternals* (figure 20-1).

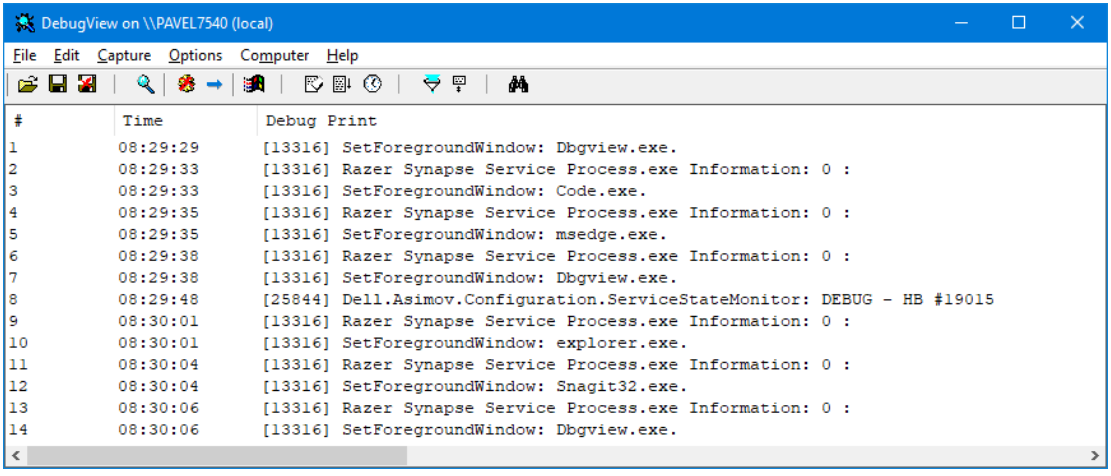


Figure 20-1: *DbgView* from *Sysinternals*

The tool provides filtering and highlight capabilities, as well as the ability to capture output from session 0 processes (*Capture / Global Win32* menu). It can also capture kernel debug output (similar to `OutputDebugString`, but from kernel device drivers). Consult its documentation for the details.

Libraries typically provide convenient wrappers over `OutputDebugString`. For example, The *Active Template Library* (ATL), which we were using in some code examples, provides the `ATLTRACE` macro that calls `OutputDebugString`, while providing two benefits:

- It allows formatting specifications similar to `printf` and its variants for building the string.
- It only compiles the call in *Debug* builds (more on this in a moment).

Here is an example:

```
ATLTRACE(L"In foo: x = %d\n", x);      // x is int
```

Other common examples of wrappers: the `TRACE` macro in MFC, the `Debug.Write*` and `Trace.Write*` methods in .NET. These all call `OutputDebugString` by default.

One possible issue with `OutputDebugString` is that if the text is actually captured (by the debugger or an external application), it slows down the calling application. This is because of the overhead associated with each call. Furthermore, `OutputDebugString` also synchronizes its write operation using a mutex (named "DBWinMutex"), which could hinder concurrency if multiple threads call `OutputDebugString` in close succession. If that's not enough, `OutputDebugStringW` (the Unicode version) translates its string to ASCII and calls `OutputDebugStringA` (this is the opposite of what most other Unicode APIs do), further degrading performance.

This explains why `ATLTRACE` is only compiled in *Debug* builds by default. The assumption is that these outputs are unwanted in *Release* builds, where the application must perform at its best. Practically speaking, having a few `OutputDebugString` calls here and there is not too bad, but using them a lot may be problematic, and should be avoided.

Even with its drawbacks, using `OutputDebugString` is an easy way to add simple tracing to an application in *Debug* builds, as the extra cost is mitigated by the usefulness of the output, which can help locate issues before testing with *Release* builds.

Another important caveat to remember about `OutputDebugString` because of its overhead, is that it changes timing, which may cause bugs to appear, or more likely, to disappear, especially when multithreading is involved. Don't assume that if the application works correctly in *Debug* build that it would work the same way in *Release* build. This is good advise in general, regardless of `OutputDebugString`, but even more important with it in play.

The *DebugPrint* Application

Knowing the way `OutputDebugString` works is useful, because custom tools can be created to capture and analyze such output. *DbgView* is a generically useful tool, but you may want to create something more specialized, perhaps to look for certain patterns in the output, or filter certain processes.

The *DebugPrint* application demonstrates how to create such a tool. The `main` function creates the two events mentioned above, as well as the file mapping object (error handling omitted):

```
HANDLE hBufferReady = ::CreateEvent(nullptr, FALSE, FALSE,
    L"DBWIN_BUFFER_READY");
HANDLE hDataReady = ::CreateEvent(nullptr, FALSE, FALSE,
    L"DBWIN_DATA_READY");

DWORD size = 1 << 12;
HANDLE hMemFile = ::CreateFileMapping(INVALID_HANDLE_VALUE, nullptr,
    PAGE_READWRITE, 0, size, L"DBWIN_BUFFER");

// map the MMF
auto buffer = (BYTE*)::MapViewOfFile(hMemFile, FILE_MAP_READ,
    0, 0, 0);
```

The logic to follow is this: The consumer (the tool) sets the "DBWIN_BUFFER_READY" event, indicating it's ready to receive data and then waits for the "DBWIN_DATA_READY" event to be signaled when `OutputDebugString` writes the data to the memory-mapped file. The tool can then extract the process ID and text, and loop back to set the "DBWIN_BUFFER_READY" event again, and so on. Here is the code:

```
while(WAIT_OBJECT_0 == ::SignalObjectAndWait(hBufferReady, hDataReady,
    INFINITE, FALSE)) {
    SYSTEMTIME local;
    ::GetLocalTime(&local);
    DWORD pid = *(DWORD*)buffer;
    printf("%ws.%03d %6d: %s\n",
        (PCWSTR)CTime(local).Format(L"%X"),
        local.wMilliseconds, pid,
        (const char*)(buffer + sizeof(DWORD)));
}
```

Using `SignalObjectAndWait` is perfect for this occasion, where we signal one object and wait on another. Once the wait completes successfully, the code reads the data from the mapped memory. First the process ID (`DWORD`), and then the string (notice it's ASCII).

The time of the call to `OutputDebugString` is not written anywhere, so the code calls `GetLocalTime` to grab the current time as the "DBWIN_BUFFER_READY" event is signaled. I use the `CTime` class from ATL that has a constructor accepting a `SYSTEMTIME` structure, and a convenient `Format` method.

Here is some example output from random processes on my system:

```
C:\>DebugPrint.exe
09:13:16.797  13316: Razer Synapse Service Process.exe Information: 0 :
09:13:16.799  13316: SetForegroundWindow: Code.exe.

09:13:29.070  13316: Razer Synapse Service Process.exe Information: 0 :
09:13:29.070  13316: SetForegroundWindow: devenv.exe.

09:13:30.938  31844: [3921-01-04/ 09:13:30.938:INFO:offscreen_window.cpp

09:13:31.388  13316: Razer Synapse Service Process.exe Information: 0 :
09:13:31.388  13316: SetForegroundWindow: explorer.exe.

09:13:32.650  13316: Razer Synapse Service Process.exe Information: 0 :
09:13:32.651  13316: SetForegroundWindow: Code.exe.

09:13:33.927  13316: Razer Synapse Service Process.exe Information: 0 :
09:13:33.928  13316: SetForegroundWindow: explorer.exe.
```

The events and MMF objects needed for capturing `OutputDebugString` text are created in the capturing process session, so can only capture output from processes in that session. Remember that kernel object names are on a per-session basis. It is possible, however, to capture output from session 0 by using object names prefixed with "Global\", as described in chapter 2 (in part 1). The only issue is that creating a memory-mapped file in session 0 namespace requires admin rights by default.

Add support to *DebugPrint* application to capture output from session 0 processes as well.

Performance Counters

Windows contains a large set of *Performance Counters*, a mechanism that allows producing and consuming counters - numbers - that represent properties of interest such as CPU utilization, CPU time, memory-related counters, network-related counters, and much more. In fact, applications and services can expose their own counters that can be queried/captured using the same API. In this section, we'll examine some of the APIs related to querying such counters. *Performance Counters* are lightweight, so capturing these is unlikely to have any adverse affect on the system.

The mechanism of working with *Performance Counters* is uniform regardless of the type of counters and the producing process/component. The counters can be either queried in real-time or captured to a file to be analyzed later.

The most well-known tool that allows configuring and capturing *Performance Counters* is the *Performance Monitor* built-in tool, that can be accessed through the administrative tools in the *Control Panel*, by running

perfmon, or simply by searching for "performance" within the Start menu. Figure 20-2 shows the initial view of that tool.

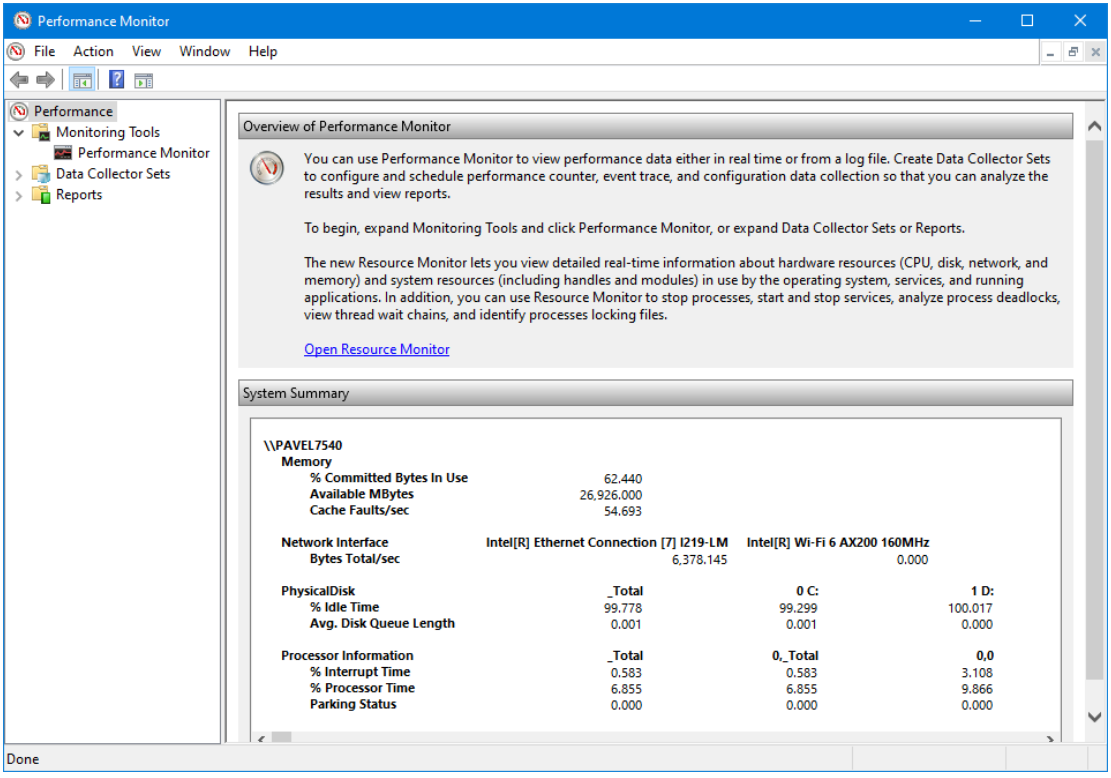


Figure 20-2: *Performance Monitor* tools

> We have used the *Performance Monitor* to show some aspects of thread scheduling in chapter 6 (part 1).

Under the *Monitoring Tools* node, you'll find the main view of the *Performance Monitor*, where a CPU graph (a counter) is shown by default (figure 20-3).

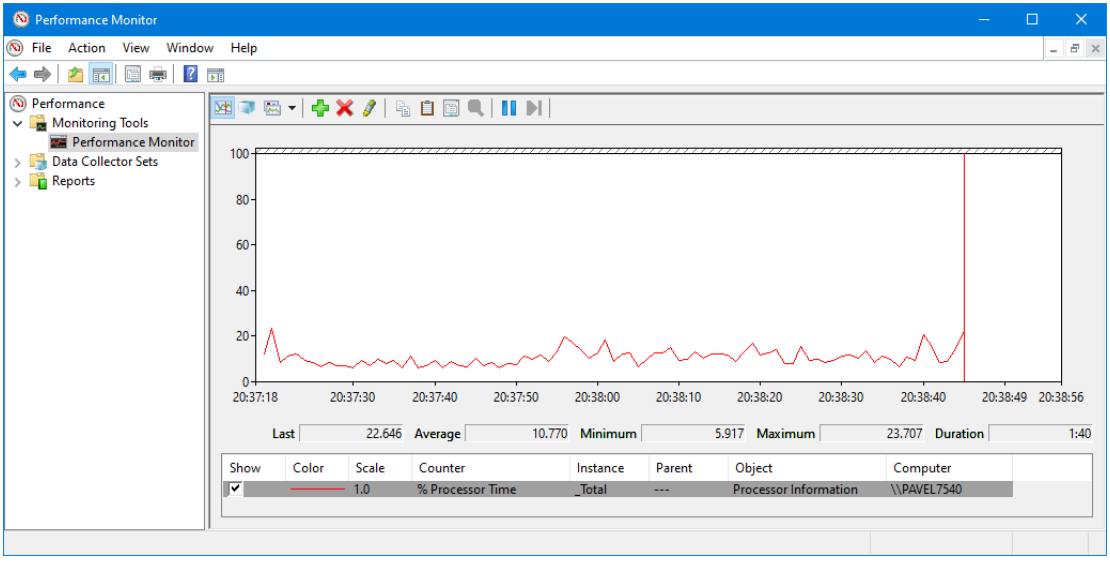


Figure 20-3: The "real" *Performance Monitor*

The details of the single counter show its name ("% Processor Time"), the instance ("_Toatl"), its lack of parent object, and the object name itself ("Processor Information"), followed by the computer name. It is possible (given the proper permissions) to look at performance counters on other machines, although impractical because the required services on the target machine is disabled by default. These parts of the single counter shown in figure 20-3 identify a counter uniquely by concatenating these names in a certain way. Double-clicking on the counter at the bottom shows its properties, where the *Data* tab indicates its full name (the computer name is omitted because the counter is local). This name is the way to identify the counter uniquely (figure 20-4).

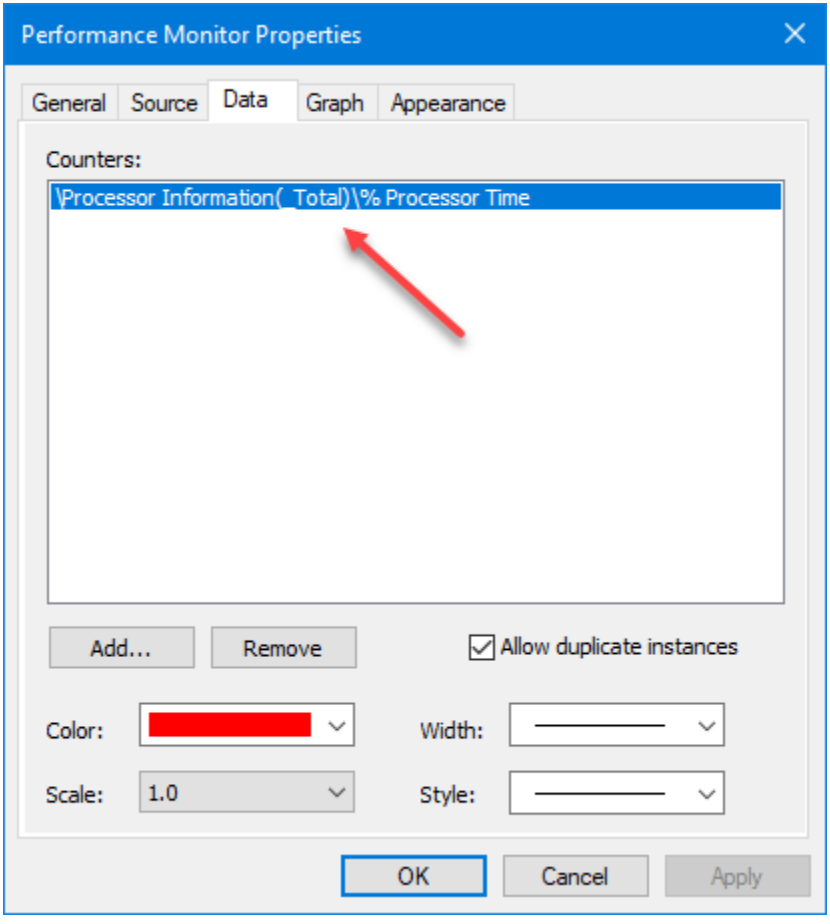


Figure 20-4: A counter's properties

The default-shown counter is the total CPU consumption on the machine, much like the graph than can be seen in *Task Manager*. We can simply delete this counter (by right-clicking and selecting *Remove All Counters* or pressing the *Del* key while the keyboard focus is on the counter at the bottom).

Adding counters to the *Performance Monitor* view is done by clicking the "Add" button (plus sign), or right-clicking the graph area and selecting *Add Counters...*. The *Add Counters* dialog box appears (figure 20-5). This is a system-provided dialog that can be shown (and manipulated to some extent) programmatically by calling `PdhBrowseCounters`:

```
typedef LONG PDH_STATUS;
PDH_STATUS
PdhBrowseCounters(_In_ PPDH_BROWSE_DLG_CONFIG pBrowseData);
```

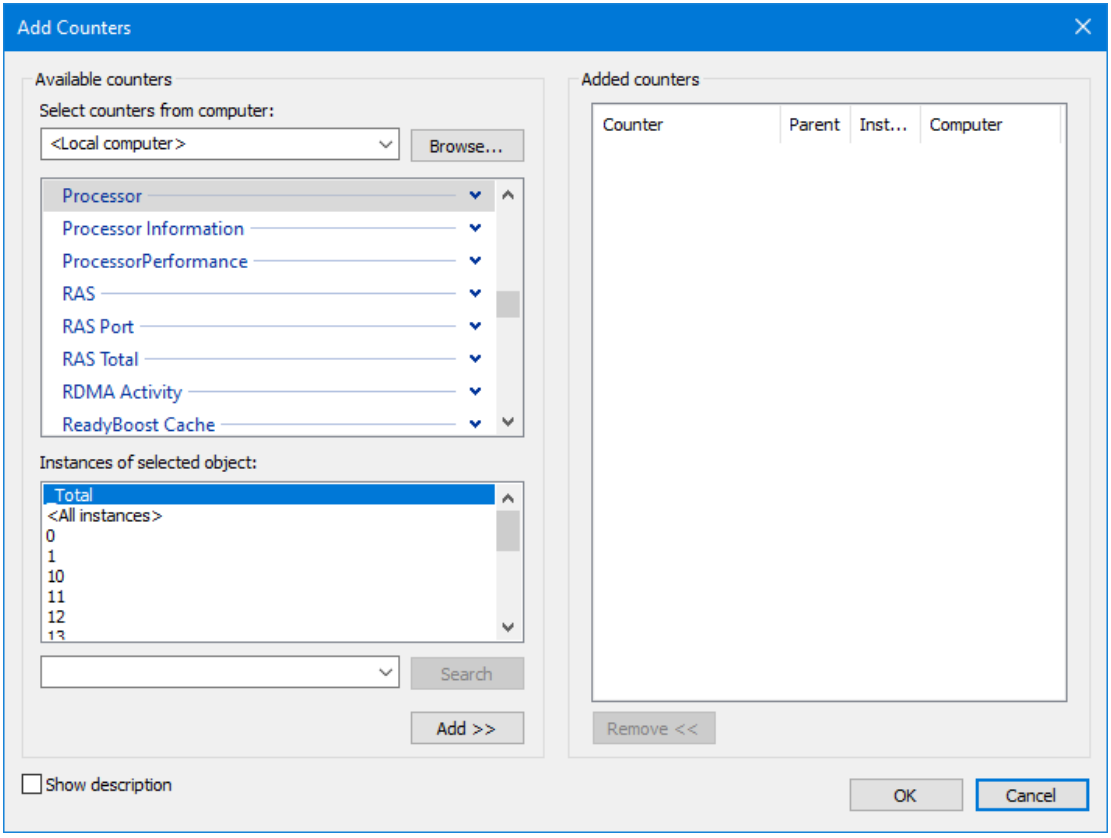


Figure 20-5: The Add Counters dialog box

`PDH_BROWSE_DLG_CONFIG` is a mandatory structure that provides configuration to the dialog to be shown, including a callback that must be specified and is invoked by the dialog box to handle counters selection by the user.

```
typedef struct _BrowseDlgConfig {
    DWORD   bIncludeInstanceIndex:1,       // show instances box
            bSingleCounterPerAdd:1,        // accept a single counter
            bSingleCounterPerDialog:1,
            bLocalCountersOnly:1,          // no remote counters
            bWildCardInstances:1,
            bHideDetailBox:1,
            bInitializePath:1,      // initialize to a specific counter
            bDisableMachineSelection:1,
            bIncludeCostlyObjects:1,
            bShowObjectBrowser:1,
            bReserved:22;
```

```
    HWND        hWndOwner;              // owner of the dialog
    LPTSTR      szDataSource;           // data source file or NULL
    LPTSTR      szReturnPathBuffer;     // counters path buffer
    DWORD       cchReturnPathLength;    // buffer length
    CounterPathCallBack pCallBack;      // callback
    DWORD_PTR   dwCallBackArg;          // callback context
    PDH_STATUS  CallBackStatus;         // status of the  path buffer
    DWORD       dwDefaultDetailLevel;   // detail level if detail box
                                        // is shown
    LPWSTR      szDialogBoxCaption;     // dialog title
} PDH_BROWSE_DLG_CONFIG, *PPDH_BROWSE_DLG_CONFIG;
```

The first set of members are Booleans representing certain dialog traits (check the documentation for the exact details). `szDataSource` can be a file name from which to retrieve the initial list of counters, where `NULL` indicates the local machine. `szReturnPathBuffer` is where the selected counter paths will be stored (as a set of strings, each terminated with zero plus a final zero at the end, exactly like `REG_MULTI_SZ` type for Registry multiple strings). This buffer must be allocated by the caller (and reallocated if needed by the callback function). `cchReturnPathLength` is the buffer's allocated length, so that any data written by the dialog box will not overflow the allocated buffer.

`pCallBack` is an optional callback invoked by the dialog box. It can have any name, but must have the following prototype:

```
PDH_STATUS CALLBACK CounterPathCallBack(DWORD_PTR arg);
```

The callback is invoked when the user selects counters to be added. The `arg` passed in is retrieved from the `dwCallBackArg` member - this can be anything, serving as a context for the callback. The typical response of the callback would be to add the requested counters by calling `PdhAddCounter` (discussed later in this section). The `CallbackStatus` member is set by the dialog box to an appropriate status of the counters buffer. `ERROR_SUCCESS` means all is well and the buffer is large enough to hold the counters, while `PDH_MORE_DATA` indicates the buffer is too small. In that case, the callback can re-allocate a large buffer and set this member to `PDH_RETRY` to force the dialog box to try again with the enlarged buffer.

The above paragraph is the theory and the way it *should* work. Currently, there seem to be two bugs related to *PdhBrowseCounters*. First, the `CallBackStatus` member is never set to `ERROR_MORE_DATA`, even if the provided buffer is too small to hold all selected counters. Instead, the buffer is filled with as many counters as possible, and the rest are discarded. Second, if *<All Instances>* is selected in the *instance* list box, an access violation exception occurs in an internal dialog box processing function when the OK/Add button is clicked.

`PdhBrowseCounters` returns `ERROR_SUCCESS` if the *OK* button was clicked (`bSingleCounterPerDialog` is `TRUE`). If *Cancel* or *Close* is clicked, `PDH_DIALOG_CANCELLED` is returned. Any other value indicates some error.

The *CounterSelector* application shows how to invoke the browse counters dialog box and collect the user's selected counters.

First, a few includes are required:

```
#include <Windows.h>
#include <Pdh.h>
#include <pdhmsg.h>
#include <stdio.h>
#include <vector>
#include <memory>
#include <string>
```

pdh.h is the header where all the *Performance Counters* API is declared. *pdhmsg.h* holds custom error codes (such as `PDH_DIALOG_CANCELLED`). The functions are implemented in *pdh.dll*, and so the code must link against its import library, *pdh.lib*:

```
#pragma comment(lib, "pdh")
```

For simplicity, the application stores the resulting counters in a global variable:

```
std::vector<std::wstring> counters;
```

The `main` function first sets up the `PDH_BROWSE_DLG_CONFIG` structure and allocates room for the resulting counters:

```
PDH_STATUS CALLBACK SelectorCallback(DWORD_PTR context);

int main() {
    PDH_BROWSE_DLG_CONFIG config = { 0 };
    auto size = 16 << 10;    // 16 KB
    auto buffer = std::make_unique<WCHAR[]>(size);
    WCHAR title[] = L"Select Counters";
    config.bDisableMachineSelection = FALSE;
    config.bLocalCountersOnly = TRUE;
    config.bSingleCounterPerDialog = TRUE;
    config.bIncludeInstanceIndex = TRUE;
    config.bHideDetailBox = TRUE;
    config.szReturnPathBuffer = buffer.get();
    config.cchReturnPathLength = size;
    config.pCallBack = SelectorCallback;
    config.dwCallBackArg = reinterpret_cast<DWORD_PTR>(&config);
    config.szDialogBoxCaption = title;
    config.szDataSource = nullptr;
    config.dwDefaultDetailLevel = PERF_DETAIL_EXPERT;
```

The callback function argument (`dwCallBackArg`) is set to the `config` structure itself. `szReturnPathBuffer` is set to the allocated buffer with the maximum size in

`cchReturnPathLength`. The `dwDefaultDetailLevel` member allows the dialog to filter counters based on "detail". Counter providers designate each counter with a level, from 100 to 400, with four constants defined for that: `PERF_DETAIL_NOVICE` (100), `PERF_DETAIL_ADVANCED` (200), `PERF_DETAIL_EXPERT` (300), and `PERF_DETAIL_WIZARD` (400).

The counters dialog shown by *Performance Monitor* uses the highest level (`PERF_DETAIL_WIZARD`).

Now it's time to invoke the dialog and handle any error:

```
auto error = ::PdhBrowseCounters(&config);
if (ERROR_SUCCESS != error && PDH_DIALOG_CANCELLED != error) {
    printf("Error invoking dialog (%u)\n", error);
    return 1;
}
```

Finally, the selected counters are displayed:

```
printf("Counters selected:\n");
for (auto& counter : counters)
    printf("%ws\n", counter.c_str());
```

How did the counters get there? This is where the callback function comes in:

```
PDH_STATUS CALLBACK SelectorCallback(DWORD_PTR context) {
    auto config = reinterpret_cast<PDH_BROWSE_DLG_CONFIG*>(context);
    if (config->CallBackStatus == ERROR_SUCCESS) {
        auto buffer = config->szReturnPathBuffer;
        for (auto p = buffer; *p; p += wcslen(p) + 1) {
            counters.push_back(p);
        }
    }
    return config->CallBackStatus;
}
```

The callback function takes the simplest approach. It checks the provided status, and if successful, it adds the counters to the `vector` by traversing the multiple string value provided in the buffer.

> You may be wondering where is the right part of the dialog shown by *Performance Monitor* where counters can be accumulated. This functionality exists in the counters browser dialog, but is not exposed publicly. *Performance Monitor*, as a Windows built-in tool, knows how to invoke that functionality.

Working with Counters

Querying performance counters first requires creating a query object by calling `PdhOpenQuery`:

```
PDH_STATUS PdhOpenQuery(
    _In_opt_ LPCTSTR      szDataSource,
    _In_     DWORD_PTR    dwUserData,
    _Out_    PDH_HQUERY * phQuery);
```

`szDataSource` is an optional path to a log file from which to query information. Such log files can be configured so that counter information is recorded into a file that can then be opened with *Performance Monitor* or custom tools. Specifying `NULL` indicates usage of real-time counters.

We won't be using log files in the samples of this chapter, only real-time counters. However, using a log file is no different in essence from real-time counters - just specify the file in question, and the rest of the code need not change.

`dwUserData` is a user-defined value that is associated with the query object. It can be used to retrieve some application-specific context without an extra value being dragged - just get it from a query when needed with `PdhGetCounterInfo`.

If successful, the function return `ERROR_SUCCESS` with the query handle in `*phQuery`. When the query is no longer needed, call `PdhCloseQuery` to release resources associated with the query:

```
PDH_FUNCTION PdhCloseQuery(PDH_HQUERY hQuery);
```

With a query in hand, counters can now be added by calling `PdhAddCounter` or `PdhAddEnglishCounter`:

```
PDH_STATUS PdhAddCounter(
    _In_  PDH_HQUERY     hQuery,
    _In_  LPCTSTR        szFullCounterPath,
    _In_  DWORD_PTR      dwUserData,
    _Out_ PDH_HCOUNTER * phCounter);
PDH_FUNCTION PdhAddEnglishCounter(
    _In_  PDH_HQUERY     hQuery,
    _In_  LPCTSTR        szFullCounterPath,
    _In_  DWORD_PTR      dwUserData,
    _Out_ PDH_HCOUNTER * phCounter);
```

The above functions are identical in functionality. `PdhAddCounter` expects a localized counter path in `szFullCounterPath`, whereas `PdhAddEnglishCounter` expects a counter path in English. `PdhAddCounter` is suitable for counters received from the counter browser dialog, or some other retriever of enumerated counters, while `PdhAddEnglishCounter` is suitable when adding counters manually and the code does not need to bother with the current locale to get to the correct counters.

`hQuery` is an open query, and `dwUserData` is an application-provided value stored with the counter. It can be accessed with `PdhGetCounterInfo`. A successful call returns a handle to the new counter in `*phCounter`. Note that function won't fail for a non-existent counter, since it cannot know whether the counter might exist later on when actually being queried. For example, a process instance name might be part of the counter path, which might be launched later on.

Once a counter is added to a query, it can be removed by calling `PdhRemoveCounter`:

```
PDH_STATUS PdhRemoveCounter(_In_ PDH_HCOUNTER hCounter);
```

Alternatively, when the query is closed, all counters are removed and cleaned up automatically.

Multiple counters can be added to a single query, and this is the preferred way of working with multiple counters. The alternative is to create multiple queries with a single counter in each. This latter approach performs poorly, and should be avoided.

With a query populated with counters, the next value of each counter is queried by calling `PdhCollectQueryData`:

```
PDH_STATUS PdhCollectQueryData(_Inout_ PDH_HQUERY hQuery);
```

Calling `PdhCollectQueryData` calculates the next value for each counter, but retrieving the values requires an additional call to one of several functions available for this task.

An extended version of `PdhCollectQueryData` - `PdhCollectQueryDataEx` - allows querying the counters with a specified interval using a separate thread.

Probably the simplest function to use is `PdhGetFormattedCounterValue`, which returns a formatted value based on a requested type:

```
PDH_FUNCTION PdhGetFormattedCounterValue(
    _In_      PDH_HCOUNTER          hCounter,
    _In_      DWORD                 dwFormat,
    _Out_opt_ LPDWORD               lpdwType,
    _Out_     PPDH_FMT_COUNTERVALUE pValue);
```

The function accepts a counter handle and a set of formatting flags. The first set indicates the format for the returned string. Possible values include `PDH_FMT_DOUBLE` (`double`), `PDH_FMT_LARGE` (64-bit integer) and `PDH_FMT_LONG` (32-bit integer). Any one of these flags can be combined with any of the following flags:

- `PDH_FMT_NOSCALE` - do not apply the default scaling factor for this counter. Some Performance counter providers scale their results by default. This option allows opting out of that.
- `PDH_FMT_NOCAP100` - some counters are capped at 100, typically for CPU-related percentage. However, in some cases the results are not correct, as with multiple processors the CPU percentage can be larger than 100 percent (each full processor represents 100 percent). Using this flag is recommended for such counters.
- `PD_FMT_1000` - multiplies the actual value by 1000.

The `lpdwType` parameter is an optional result indicating the type of the performance counter. This could be useful in rare cases where the code is generic and needs to handle unknown counters. Finally, the result for a successful call is provided within a `PDH_FMT_COUNTERVALUE` structure:

```
typedef struct _PDH_FMT_COUNTERVALUE {
    DWORD    CStatus;
    union {
        LONG        longValue;
        double      doubleValue;
        LONGLONG    largeValue;
        LPCSTR      AnsiStringValue;    // not supported
        LPCWSTR     WideStringValue;    // not supported
    };
} PDH_FMT_COUNTERVALUE, * PPDH_FMT_COUNTERVALUE;
```

The `CStatus` member indicates the validity of the result. If it's `ERROR_SUCCESS` (0), the union holds valid information. Note that the string members are never used - performance counters can only be numbers.

In some special cases, you may want to look at the original counter values before formatting is applied. As an alternative to `PdhGetFormattedCounterValue`, you can call `PdhGetRawCounterValue`:

```
PDH_FUNCTION PdhGetRawCounterValue(
    _In_      PDH_HCOUNTER      hCounter,
    _Out_opt_ LPDWORD           lpdwType,
    _Out_     PPDH_RAW_COUNTER  pValue);
```

The first two parameters are the same as in `PdhGetFormattedCounterValue`. The "raw" value is returned in a `PDH_RAW_COUNTER` structure:

```
typedef struct _PDH_RAW_COUNTER {
    volatile DWORD CStatus;
    FILETIME    TimeStamp;
    LONGLONG    FirstValue;
    LONGLONG    SecondValue;
    DWORD       MultiCount;
} PDH_RAW_COUNTER, * PPDH_RAW_COUNTER;
```

`CStatus` has the same role it does in `PdhGetFormattedCounterValue`. `TimeStamp` provides the local time in which the counter sample was taken. Curiously enough, this information is absent from `PDH_FMT_COUNTERVALUE`. Since this is "raw" data, it exposes the fact that internally performance counters are 64-bit values, and in some cases two of them are needed to generate the next "value" (for example, some counters are calculated by subtracting the previous value). The type of counter (returned in `lpdwType`) points to the formula used to calculate that counter. The final member of `PDH_RAW_COUNTER`, `MultiCount` points to an optional counter used as extra data for calculating the final value if multiple counters are involved.

How can we calculate the final value using `PDH_RAW_COUNTER`? Fortunately, there is a function for that:

```
PDH_FUNCTION PdhCalculateCounterFromRawValue(
    _In_  PDH_HCOUNTER          hCounter,
    _In_  DWORD                 dwFormat,
    _In_  PPDH_RAW_COUNTER      rawValue1,
    _In_  PPDH_RAW_COUNTER      rawValue2,
    _Out_ PPDH_FMT_COUNTERVALUE fmtValue);
```

`PdhCalculateCounterFromRawValue` combines the data from one or two `PDH_RAW_COUNTER` objects, with the flags from `PdhGetFormattedCounterValue`, returning the result in a `PDH_FMT_COUNTERVALUE` instance. `dwFormat` has the same meaning as it does in `PdhGetFormattedCounterValue`.

> Creating a performance counter provider is beyond the scope of this chapter. Check out the documentation for more details.

The *QSlice* Application

The *QSlice* application is a tribute to an old tool with the same name that was part of the Windows NT 4 resource kit. It showed the CPU consumption of processes in a simple graphic way. Our version of *QSlice* is going to be textual (a nicer graphic version is left as an exercise for the reader). Figure 20-6 shows a snapshot of the application in action.

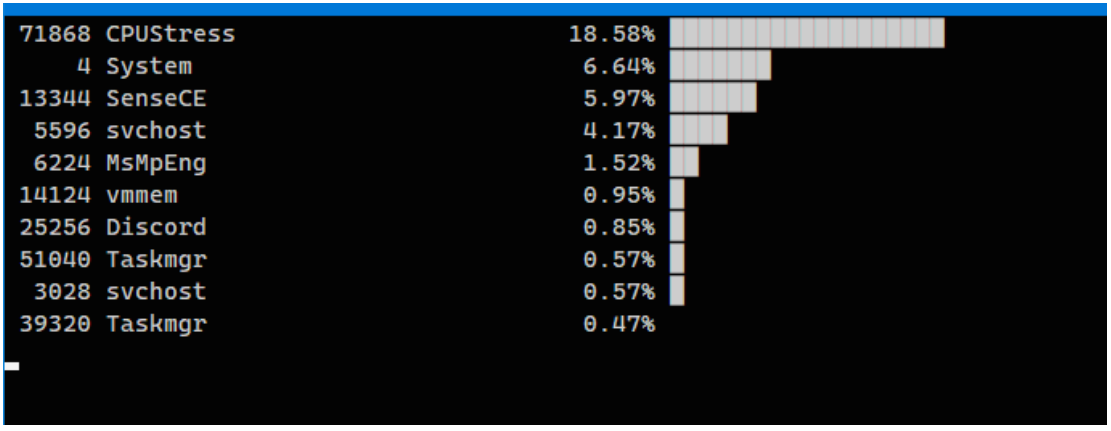


Figure 20-6: *QSlice* in action

The 10 most CPU consuming processes are shown, in descending order of CPU consumption, where each "block" represents one percent of CPU consumption. The output is updated once a second.

The project uses *WIL* to help manage handles. Since *WIL* does not currently provide wrappers for query and counter handles, we can define them like so (with the needed includes):

```
#include <wil\resource.h>
#include <Pdh.h>
#include <assert.h>

using unique_pdh_query = wil::unique_any_handle_null_only<
    decltype(&::PdhCloseQuery), ::PdhCloseQuery>;
using unique_pdh_counter = wil::unique_any_handle_null_only<
    decltype(&::PdhRemoveCounter), ::PdhRemoveCounter>;
```

We used this technique back in chapter 2 of part 1. We also need to link against the PDH library:

```
#pragma comment(lib, "pdh")
```

For each process, we'll define a structure to hold its relevant information: PID, name, and CPU consumption:

```
struct ProcessInfo {
    DWORD Pid;
    std::wstring Name;
    double CPU;
};
```

Now we're ready to start with the `main` function. First, we'll take an optional input from the user with the number of processes to show (default is 10):

```
int main(int argc, const char* argv[]) {
    auto display = argc > 1 ? atoi(argv[1]) : 10;
    if (display < 5)
        display = 5;
```

We will build a query that holds two counter groups: one for a list of process names and PIDs, and the other for CPU consumption. To that end, we can leverage the wildcard support performance counter queries have (error handling omitted):

```
unique_pdh_query hQuery;
::PdhOpenQuery(nullptr, 0, hQuery.addressof());

unique_pdh_counter hPidCounter, hCpuCounter;
::PdhAddEnglishCounter(hQuery.get(), L"\\Process(*)\\ID Process", 0,
    hPidCounter.addressof());
::PdhAddEnglishCounter(hQuery.get(),
    L"\\Process(*)\\% Processor Time", 0, hCpuCounter.addressof());
```

We must use `PdhAddEnglishCounter` to add the English version of the counters, so we don't have to match the locale. Note the object names specified as an asterisk wildcard. This should bring information for all the processes existing on the system at the time we perform a query:

```
::PdhCollectQueryData(hQuery.get());
::PdhCollectQueryData(hQuery.get());
```

The two calls to `PdhCollectQueryData` force getting two values, which are required for the CPU consumption case, as the current value is based on the previous one. Other counters don't have such a dependency (such as counters providing some absolute value). Before we get into a loop that gets the counter values every second, a few more preparations:

```
auto processors = ::GetActiveProcessorCount(ALL_PROCESSOR_GROUPS);

auto hConsole = ::GetStdHandle(STD_OUTPUT_HANDLE);
CONSOLE_SCREEN_BUFFER_INFO info = { sizeof(info) };
::GetConsoleScreenBufferInfo(hConsole, &info);
info.dwSize.X = 150;
::SetConsoleScreenBufferSize(hConsole, info.dwSize);
```

We store the number of processors on the system, so we can adjust the CPU consumption for a process to be no more than 100 percent. The console buffer width is set to 150, to make sure that even 100% CPU utilization would keep the output on the same line.

Now we can start the loop. The first thing to do is retrieve the process IDs:

```
for (;;) {
    DWORD size = 0;
    DWORD count;
    ::PdhGetFormattedCounterArray(hPidCounter.get(),
        PDH_FMT_LONG, &size,
        &count, nullptr);
    auto pidBuffer = std::make_unique<BYTE[]>(size);
    auto pidData = (PPDH_FMT_COUNTERVALUE_ITEM)pidBuffer.get();
    ::PdhGetFormattedCounterArray(hPidCounter.get(),
        PDH_FMT_LONG, &size,
    &count, pidData);
```

Since we have multiple counters to get values from, which depend on the number of processors on the system at the time, we need a convenient way to get everything. This is where `PdhGetFormattedCounterArray` comes in. It's similar to `PdhGetFormattedCounterValue`, but works on an array of counters where wildcards are used:

```
PDH_FUNCTION PdhGetFormattedCounterArray(
    _In_    PDH_HCOUNTER hCounter,
    _In_    DWORD        dwFormat,
    _Inout_ LPDWORD      lpdwBufferSize,
    _Out_   LPDWORD      lpdwItemCount,
    _Out_   PPDH_FMT_COUNTERVALUE_ITEM ItemBuffer);
```

The function accepts the counter handle (that internally contains an array of counters), the format (same flags used for `PdhGetFormattedCounterValue`), and a buffer size for the result that must be set to sufficient size to get the data. As output, the function returns the number of values (in `*lpdwItemCount`), and the values themselves in an array of `PDH_FMT_COUNTERVALUE_ITEM` structures, defined like so:

```
typedef struct _PDH_FMT_COUNTERVALUE_ITEM {
    LPTSTR                  szName;
    PDH_FMT_COUNTERVALUE    FmtValue;
} PDH_FMT_COUNTERVALUE_ITEM, * PPDH_FMT_COUNTERVALUE_ITEM;
```

The returned value for each counter contains the counter name (this is the process name in our case, as the wildcard is specified for process instance names), and the same `PDH_FMT_COUNTERVALUE` as returned with `PdhGetFormattedCounterValue`.

The usual way of working with `PdhGetFormattedCounterArray` is by calling it twice: the first time with a size of zero, for which the function returns the required size, and then a second time after a buffer has been properly allocated.

Now that we have an array of PIDs (and process names), we can do the same for the CPU consumption counter:

```
size = 0;
DWORD count2;
::PdhGetFormattedCounterArray(hCpuCounter.get(),
    PDH_FMT_DOUBLE | PDH_FMT_NOCAP100,
    &size, &count2, nullptr);
auto cpuBuffer = std::make_unique<BYTE[]>(size);
auto cpuData = (PPDH_FMT_COUNTERVALUE_ITEM)cpuBuffer.get();
::PdhGetFormattedCounterArray(hCpuCounter.get(),
    PDH_FMT_DOUBLE | PDH_FMT_NOCAP100,
    &size, &count2, cpuData);

assert(count == count2);
```

The `assert` call is there for a sanity check, making sure the same number of instances (processes) is returned.
Now it's time to build the array of processes, take items from the first and second counter arrays to compose a full process array:

```
std::vector<ProcessInfo> processes;
processes.reserve(count);
for (DWORD i = 0; i < count; i++) {
    auto& p = pidData[i];
    auto& p2 = cpuData[i];
    if (p.FmtValue.longValue == 0)
        continue;

    //
    // sanity check: instance names are the same
    //
    assert(::_wcsicmp(p.szName, p2.szName) == 0);

    ProcessInfo pi;
    pi.Name = p.szName;
    pi.Pid = p.FmtValue.longValue;
    pi.CPU = p2.FmtValue.doubleValue;
    processes.push_back(std::move(pi));
}
```

PID 0 is not put into the vector, since that's the idle process, which does not "consume" CPU in the normal sense. The vector is needed so it's easy to sort:

```
std::sort(processes.begin(), processes.end(),
    [](const auto& p1, const auto& p2) {
    // p1 and p2 are of type const ProcessInfo&
        return p1.CPU > p2.CPU;
    });
```

The process objects are sorted based on their CPU values, using the `std::sort` algorithm, where the callback for comparing objects is specified using a lambda function. At this point, we can display the topmost consuming processes based on the `display` variable initialized at the beginning of `main`:

```
for (int i = 0; i < display; i++) {
    auto& p = processes[i];
    auto cpu = p.CPU;
    int blocks = (int)(cpu / processors + .5);
    printf("%6d %-30ws %6.2f%% %s%s\n",
        p.Pid, p.Name.c_str(), cpu / processors,
        std::string(blocks, (char)219).c_str(),
        std::string(101 - blocks, ' ').c_str());
}
```

The extended ASCII character number 219 is the filled rectangle shown in figure 20-6. Of course, any character can be used instead.

The last step before closing the loop body is to wait a second before adjusting the cursor to the beginning of the output and querying for the next set of values:

```
        ::Sleep(1000);
        printf("\r\033[%dA", display);
        ::PdhCollectQueryData(hQuery.get());
    }
    return 0;
}
```

The trick with the `033[nA` above allows moving up `n` lines. This works with the Visual Studio command window (such as when pressing Ctrl+F5 to run), and also in the new *Windows Terminal*, but does not work in the standard command window (at the time of this writing).

Process Snapshots

Windows 8.1 introduced the ability to take process snapshots while a process is executing, where various aspects of a process are captured without any adverse effect to the process itself. This includes information

about handles, threads, virtual memory layout and even the memory itself. This information can then be persisted, displayed and even compared to later snapshots for the same process (or even earlier runs of the same image), that could provide valuable information about the process behavior and resource usage.

All snapshotting functions are declared in *<processsnapshot.h>* and are implemented in *kernel32.dll*, so no extra libraries are needed for linking. Capturing a snapshot of a process is done with `PssCaptureSnapshot`:

```
DWORD PssCaptureSnapshot(
    _In_ HANDLE ProcessHandle,
    _In_ PSS_CAPTURE_FLAGS CaptureFlags,
    _In_opt_ DWORD ThreadContextFlags,
    _Out_ HPSS* SnapshotHandle);
```

The first parameter is an open process handle. The handle access mask is not specified explicitly in the documentation because it depends on the snapshot information you would like to capture. The best option is to use `MAXIMUM_ALLOWED` when opening a handle to the target process to get the highest possible access. The `CaptureFlags` enum indicates what parts of the process to capture:

```
typedef enum {
    PSS_CAPTURE_NONE                                = 0x00000000,
    PSS_CAPTURE_VA_CLONE                            = 0x00000001,
    PSS_CAPTURE_HANDLES                             = 0x00000004,
    PSS_CAPTURE_HANDLE_NAME_INFORMATION             = 0x00000008,
    PSS_CAPTURE_HANDLE_BASIC_INFORMATION            = 0x00000010,
    PSS_CAPTURE_HANDLE_TYPE_SPECIFIC_INFORMATION    = 0x00000020,
    PSS_CAPTURE_HANDLE_TRACE                        = 0x00000040,
    PSS_CAPTURE_THREADS                             = 0x00000080,
    PSS_CAPTURE_THREAD_CONTEXT                      = 0x00000100,
    PSS_CAPTURE_THREAD_CONTEXT_EXTENDED             = 0x00000200,
    PSS_CAPTURE_VA_SPACE                            = 0x00000800,
    PSS_CAPTURE_VA_SPACE_SECTION_INFORMATION        = 0x00001000,
    PSS_CAPTURE_IPT_TRACE                           = 0x00002000,

    PSS_CREATE_BREAKAWAY_OPTIONAL                   = 0x04000000,
    PSS_CREATE_BREAKAWAY                            = 0x08000000,
    PSS_CREATE_FORCE_BREAKAWAY                      = 0x10000000,
    PSS_CREATE_USE_VM_ALLOCATIONS                   = 0x20000000,
    PSS_CREATE_MEASURE_PERFORMANCE                  = 0x40000000,
    PSS_CREATE_RELEASE_SECTION                      = 0x80000000
} PSS_CAPTURE_FLAGS;
```

Table 20-1 details the flags with their meaning. Some basic process information is always captured, regardless of the capture flags (even with `PSS_CAPTURE_NONE`).

Table 20-1: `PssCaptureProcess` capture flags

Flag (`PSS_CAPTURE_`)	Description
`NONE`	Capture nothing
`HANDLES`	Handle table (values only)
`HANDLES_NAME_INFORMATION`	Object names
`HANDLE_BASIC_INFORMATION`	Basic information for handles/objects (handle count, pointer count, access, etc.)
`HANDLE_TYPE_SPECIFIC_INFORMATION`	Extra information for supported object types (process, thread, mutex, event, semaphore, section)
`HANDLE_TRACE`	Handle trace table (semi-documented handle usage tracing facility)
`THREADS`	Thread IDs
`THREAD_CONTEXT`	Context for each thread (`CONTEXT` structure)
`THREAD_CONTEXT_EXTENDED`	Extended context for each thread (`CONTEXT_XTATE` structure)
`VA_SPACE`	Address space information (similar to `VirtualQueryEx`)
`VA_SPACE_SECTION_INFORMATION`	Gets mapped files information. `PROCESS_VM_READ` access required
`VA_CLONE`	Clones memory by creating a process using the internal `fork` facility
`IPT_TRACE`	Capture *Intel Performance Trace* data (if supported)

In addition to the capture flags from table 20-1, a few more flags can be specified that relate to the clone process (if created), shown in table 20-2.

Table 20-2: `PssCaptureProcess` clone process flags

Flag (`PSS_CREATE_`)	Description
`BREAKWAY`	The clone process is created outside if the original's process job. This is equivalent to using `CreateProcess` with `CREATE_BREAKAWAY_FROM_JOB`
`FORCE_BREAKAWAY`	The clone process is forcefully broken from the original's process job. This is only allowed for callers having the *SeTcbPrivilege*
`BREAKAWAY_OPTIONAL`	Must be specified with one or both the above flags. If the clone cannot be created outside the job, it's created inside the job
`USE_VM_ALLOCATION`	Specifies that the capturing facility should use `VirtualAlloc` for allocations rather than the heap API. This could be needed if the heap might be corrupted
`RELEASE_SECTION`	The clone process does not hold a reference to the underlying image. Some functions on the clone process might fail in such a case

The power of process snapshotting comes from the internal *fork* facility supported by the Windows kernel

but not exposed through the Windows API. This mechanism allows creating a clone process as a relatively cheap operation.

The *ThreadContextFlags* parameter to `PssCreateSnapshot` indicates which parts of a thread's `CONTEXT` structure to capture. Refer to the declaration of the `CONTEXT` for the possible values. The simplest is `CONTEXT_ALL` to capture the entire context.

The last parameter to `PssCaptureContext` returns an opaque handle to the snapshot. The function itself returns `ERROR_SUCCESS` if the capture is successful.

When you're done with the snapshot, destroy it with `PssFreeSnapshot`:

```
DWORD PssFreeSnapshot(
    _In_ HANDLE ProcessHandle,
    _In_ HPSS SnapshotHandle);
```

`SnapshotHandle` is the returned handle from `PssCaptureSnapshot`. *ProcessHandle* is normally `GetCurrentProcess()`, but could be a different process handle if the snapshot handle was duplicated by calling
`PssDuplicateSnapshot`:

```
DWORD PssDuplicateSnapshot(
    _In_ HANDLE SourceProcessHandle,
    _In_ HPSS SnapshotHandle,
    _In_ HANDLE TargetProcessHandle,
    _Out_ HPSS* TargetSnapshotHandle,
    _In_opt_ PSS_DUPLICATE_FLAGS Flags);
```

The function is similar, in principle, to `DuplicateHandle`. The source process handle must have `PROCESS_DUP_HANDLE` and `PROCESS_VM_READ` access mask bits. *SnapshotHandle* is the snapshot handle (`HPSS`) in the context of the source process. The target process handle must have `PROCESS_VM_OPERATION`, `PROCESS_VM_WRITE` and `PROCESS_DUP_HANDLE` access. *TargetSnapshotHandle* will hold the duplicated handle if the operation is successful. Finally, *Flags* is usually `PSS_DUPLICATE_NONE`, but can be `PSS_DUPLICATE_CLOSE_SOURCE` to close the original snapshot. The latter option can only work if `PSS_CREATE_USE_VM_ALLOCATIONS` wa specified as part of `PssCaptureSnapshot`.

The snapshot handle is not a normal kernel object handle (there is no "snapshot" type of object), so normal handle operations like `CloseHandle` cannot succeed.

Querying a Snapshot

Once a snapshot is available, it can be queried in two ways. The "singleton" information in the snapshot, such as basic process information is retrieved with `PssQuerySnapshot`:

```
DWORD PssQuerySnapshot(
    _In_ HPSS SnapshotHandle,
    _In_ PSS_QUERY_INFORMATION_CLASS InformationClass,
    _Out_writes_bytes_(BufferLength) void* Buffer,
    _In_ DWORD BufferLength);
```

The *InformationClass* parameter is an enumeration for selecting the information to read. This information must have been captured in the first place. The enum values and the associated structure that must be passed in the *Buffer* parameter are shown in table 20-3.

Table 20-3: **PSS_QUERY_INFORMATION_CLASS** enum

Value (**PSS_QUERY_**)	Associated structure	Description
PROCESS_INFORMATION	PSS_PROCESS_INFORMATION	General process information
VA_CLONE_INFORMATION	PSS_VA_CLONE_INFORMATION	Handle to the cloned process (if clone created)
VA_SPACE_INFORMATION	PSS_VA_SPACE_INFORMATION	Number of memory regions
HANDLE_INFORMATION	PSS_HANDLE_INFORMATION	Number of captured handles
THREAD_INFORMATION	PSS_THREAD_INFORMATION	Number of threads captured and context size
HANDLE_TRACE_INFORMATION	PSS_HANDLE_TRACE_-INFORMATION	Section (memory mapped file) handle and size where the trace data is stored
PERFORMANCE_COUNTERS	PSS_PERFORMANCE_COUNTERS	A selection of performance counters captured for the process
AUXILIARY_PAGES_-INFORMATION	PSS_AUXILIARY_PAGES_-INFORMATION	Number of auxilary pages captured

For example, given a snapshot handle, the following function displays some process informaion obtained from the snapshot:

```
void DisplayProcessInfo(HPSS hSnapshot) {
    PSS_PROCESS_INFORMATION psinfo;
    if(ERROR_SUCCESS == ::PssQuerySnapshot(hSnapshot,
        PSS_QUERY_PROCESS_INFORMATION,
        &psinfo, sizeof(psinfo))) {
        printf("Image file: %ws\n", psinfo.ImageFileName);
        printf("PID: %u\n", psinfo.ProcessId);
        printf("Parent PID: %u\n", psinfo.ParentProcessId);
        printf("Working set: %zd MB\n", psinfo.WorkingSetSize >> 20);
        printf("Commit size: %zd MB\n", psinfo.PagefileUsage >> 20);
    }
}
```

If the initial capturing flags require detailed information on threads, handles or virtual memory, then multiple pieces of information are expected. Obtaining this data requires "walking" the snapshot with *PssWalkSnapshot*:

```
DWORD PssWalkSnapshot(
    _In_ HPSS SnapshotHandle,
    _In_ PSS_WALK_INFORMATION_CLASS InformationClass,
    _In_ HPSSWALK WalkMarkerHandle,
    _Out_writes_opt_(BufferLength) void* Buffer,
    _In_ DWORD BufferLength);
```

Traversing the snapshot only makes sense for certain details, designated by the `PSS_WALK_INFORMATION_CLASS` enumeration:

```
typedef enum {
    PSS_WALK_AUXILIARY_PAGES = 0,    // PSS_AUXILIARY_PAGE_ENTRY
    PSS_WALK_VA_SPACE = 1,           // PSS_VA_SPACE_ENTRY
    PSS_WALK_HANDLES = 2,            // PSS_HANDLE_ENTRY
    PSS_WALK_THREADS = 3             // PSS_THREAD_ENTRY
} PSS_WALK_INFORMATION_CLASS;
```

The term "auxilary pages" used by `PSS_WALK_INFORMATION_CLASS` refers to the `KSHARED_USER_DATA` structure we met in chapter 3 (part 1), and is only partially documented. The rest of the flags relate to the virtual address space, handles and threads, respectively.

Before starting a "walk" over the snapshot, a "walk marker" is required that keeps context information used while traversing the data. Such a marker must be first obtained with `PssWalkMarkerCreate`:

```
DWORD PssWalkMarkerCreate(
    _In_opt_ PSS_ALLOCATOR const *Allocator,
    _Out_ HPSSWALK* WalkMarkerHandle);
```

The `PSS_ALLOCATOR` structure allows customizing the way the marker allocates and frees memory for its internal usage. Passing `NULL` is fine and lets the marker use its default (which uses the default process heap).

Once the walk marker is no longer needed, call `PssWalkMarkerFree` to free any resources used by the marker:

```
DWORD PssWalkMarkerFree(_In_ HPSSWALK WalkMarkerHandle);
```

Each walk type has an associated data structure returned for each iteration. The way to perform the "walking" is to call `PssWalkSnapshot` in a loop until it returns `ERROR_NO_MORE_ITEMS`, indicating there are no more items of the walk type.

The following code snippet (taken from the *snapproc* demo described in the next section) shows such a loop:

```
HPSSWALK hWalk;
if(ERROR_SUCCESS == ::PssWalkMarkerCreate(nullptr, &hWalk)) {
    PSS_THREAD_ENTRY thread;
    while(ERROR_SUCCESS == ::PssWalkSnapshot(hSnapshot,
        PSS_WALK_THREADS, hWalk, &thread, sizeof(thread))) {
        printf("TID: %6u Created: %s Priority: %2d "
            "User: %s Kernel: %s\n",
            thread.ThreadId,
            TimeToString(thread.CreateTime).c_str(),
            thread.Priority,
            TimeSpanToString(thread.UserTime).c_str(),
            TimeSpanToString(thread.KernelTime).c_str());
    }
    ::PssWalkMarkerFree(hWalk);
}
```

There are a few more functions related to walk marker manipulation: `PssWalkerMarkerGetPosition`, `PssWalkMarkerSeekToBeginning`, and `PssWalkMarkerSetPosition`. Check out the documentation for the details.

The *snapproc* Application

The *snapproc* application demonstrates the basic usage of process snapshoting. It accepts a process ID on the command line, and an optional set of flags to use for capturing. Here is the beginning of the `main` function that captures parameters from the command line:

```
int main(int argc, const char* argv[]) {
    if(argc < 2) {
        printf("Usage: snapproc <pid> [htvm]\n");
        return 0;
    }

    DWORD pid = atoi(argv[1]);
    DWORD error;
    PSS_CAPTURE_FLAGS flags = PSS_CAPTURE_NONE;
    if(argc > 2) {
        char ch;
        const char* s = argv[2];
        while(ch = *s++) {
            switch(ch) {
                case 'h': case 'H':
```

```
                    flags |= PSS_CAPTURE_HANDLES |
                        PSS_CAPTURE_HANDLE_NAME_INFORMATION |
                        PSS_CAPTURE_HANDLE_BASIC_INFORMATION;
                    break;

                case 't': case 'T':
                    flags |= PSS_CAPTURE_THREADS;
                    break;

                case 'v': case 'V':
                    flags |= PSS_CAPTURE_VA_SPACE;
                    break;

                case 'm': case 'M':
                    flags |= PSS_CAPTURE_VA_CLONE;
                    break;

            }
        }
    }
```

It's fairly easy to add more capturing flags. The rest of `main` calls helper functions to capture the snapshot, display information and free the snapshot:

```
auto hSnapshot = CreateSnapshot(pid, flags, error);
if(hSnapshot == nullptr) {
    printf("Error capturing snapshot: %u\n", error);
    return 1;
}

DisplayProcessInfo(hSnapshot);
if(flags & PSS_CAPTURE_HANDLES)
    DisplayHandlesInfo(hSnapshot);

if(flags & PSS_CAPTURE_THREADS)
    DisplayThreadInfo(hSnapshot);

::PssFreeSnapshot(::GetCurrentProcess(), hSnapshot);
```

Let's look at these helper functions in turn. First, capturing a snapshot:

```
HPSS CreateSnapshot(DWORD pid, PSS_CAPTURE_FLAGS flags,
    DWORD& error) {
    HANDLE hProcess = ::OpenProcess(MAXIMUM_ALLOWED, FALSE, pid);
    if(!hProcess) {
        error = ::GetLastError();
        return nullptr;
    }

    HPSS hSnapShot;
    error = ::PssCaptureSnapshot(hProcess, flags, 0, &hSnapShot);
    ::CloseHandle(hProcess);

    if(ERROR_SUCCESS != error)
        return nullptr;

    return hSnapShot;
}
```

The process in question is opened with maximum permissions the caller can muster (this is good example of using `MAXIMUM_ALLOWED`). If that works, a process snapshot is captured by calling `PssCaptureSnapshot` with the flags provided by the caller. Assuming all is well, the process handle can be closed and the snapshot handle returned.

`DisplayProcessInfo` is called to show basic information for the process. Remember, that even if no flags are used for the capture, this information is always retrieved:

```
void DisplayProcessInfo(HPSS hSnapshot) {
    PSS_PROCESS_INFORMATION psinfo;
    if(ERROR_SUCCESS == ::PssQuerySnapshot(hSnapshot,
        PSS_QUERY_PROCESS_INFORMATION,
        &psinfo, sizeof(psinfo))) {
        printf("Image file: %ws\n", psinfo.ImageFileName);
        printf("PID: %u\n", psinfo.ProcessId);
        printf("Parent PID: %u\n", psinfo.ParentProcessId);
        printf("Create time: %s\n",
            TimeToString(psinfo.CreateTime).c_str());
        printf("User time: %s\n",
            TimeSpanToString(psinfo.UserTime).c_str());
        printf("Kernel time: %s\n",
            TimeSpanToString(psinfo.KernelTime).c_str());
        printf("Working set: %zd MB\n", psinfo.WorkingSetSize >> 20);
        printf("Commit size: %zd MB\n", psinfo.PagefileUsage >> 20);
        printf("Virtual size: %zd MB\n", psinfo.VirtualSize >> 20);
```

```
    }
}
```

`TimeSpanToString` and `TimeToString` are helper functions (you can find them as part of the project), that format `FILETIME` structures as time span or absolute time, as appropriate.

Displaying information for handles requires walking (unless only the `PSS_CAPTURE_HANDLES` flag is used for capturing the snapshot, which would not bring handle details) the snapshot with handle information:

```
void DisplayHandlesInfo(HPSS hSnapshot) {
    PSS_HANDLE_INFORMATION info;
    if(ERROR_SUCCESS != ::PssQuerySnapshot(hSnapshot,
        PSS_QUERY_HANDLE_INFORMATION, &info, sizeof(info))) {
        printf("No handle information\n");
        return;
    }

    printf("Handles captured: %u\n", info.HandlesCaptured);

    HPSSWALK hWalk;
    if(ERROR_SUCCESS == ::PssWalkMarkerCreate(nullptr, &hWalk)) {
        PSS_HANDLE_ENTRY handle;
        while(ERROR_SUCCESS == ::PssWalkSnapshot(hSnapshot,
            PSS_WALK_HANDLES, hWalk, &handle, sizeof(handle))) {
            printf("Handle: %4u  Name: %ws Type: %ws\n",
                HandleToULong(handle.Handle),
                std::wstring(handle.ObjectName,
                    handle.ObjectNameLength / sizeof(WCHAR)).c_str(),
                std::wstring(handle.TypeName,
                    handle.TypeNameLength / sizeof(WCHAR)).c_str());
        }
        ::PssWalkMarkerFree(hWalk);
    }
}
```

Walking the handle information returns a `PSS_HANDLE_ENTRY` structure, parts of which are documented, and the rest can be guessed. Part of the structure is a union, where object-specific information is available for some object types.

Similarly, thread information is available as well (if captured):

```
void DisplayThreadInfo(HPSS hSnapshot) {
    PSS_THREAD_INFORMATION info;
    if(ERROR_SUCCESS != ::PssQuerySnapshot(hSnapshot,
        PSS_QUERY_THREAD_INFORMATION, &info, sizeof(info))) {
        printf("No thread information\n");
        return;
    }

    printf("Threads captured: %u\n", info.ThreadsCaptured);

    HPSSWALK hWalk;
    if(ERROR_SUCCESS == ::PssWalkMarkerCreate(nullptr, &hWalk)) {
        PSS_THREAD_ENTRY thread;
        while(ERROR_SUCCESS == ::PssWalkSnapshot(hSnapshot,
            PSS_WALK_THREADS, hWalk, &thread, sizeof(thread))) {
            //
            // display some thread details
            //
            printf("TID: %6u Created: %s Priority: %2d "
                "User: %s Kernel: %s\n",
                thread.ThreadId,
                TimeToString(thread.CreateTime).c_str(),
                thread.Priority,
                TimeSpanToString(thread.UserTime).c_str(),
                TimeSpanToString(thread.KernelTime).c_str());
        }
        ::PssWalkMarkerFree(hWalk);
    }
}
```

Exercises

1. Add code to display virtual address space details.
2. Add code to allow taking two snapshots, a few seconds apart, displaying the differences.
3. Add code to use the cloned process handle to read data at some virtual address.

Event Tracing for Windows

We've seen in the preceding sections two ways for reporting activities by a process: using `OutputDebugString` and performance counters. These, however, suffer from two potential problems:

- These mechanisms are fairly slow and should not be used with high frequency.
- They lack a schema. For `OutputDebugString` - well, it's just a string, which could be anything. There is no way to know what to expect on the receiving end. For performance counters - these are numbers only, which for some cases is not rich enough.

Event Tracing for Windows (ETW) is a mechanism introduced back in Windows 2000, that provides a high-performance, schema-based mechanism for publishing events. ETW is based around the following concepts:

- Providers - components that send notifications. These can be in user-mode or kernel-mode.
- Sessions - a session establish a set of providers, along with configuration information. A session's events can be bound (persisted) to a file or delivered in "real-time", so that events are not automatically stored.
- Consumers - these consume a session's events (either from a file or in real-time).
- Controllers - these control a session, such as starting or stopping collection, configuring providers, etc. In many cases, controllers and consumers are one and the same.

The architecture of ETW is depicted in figure 20-7.

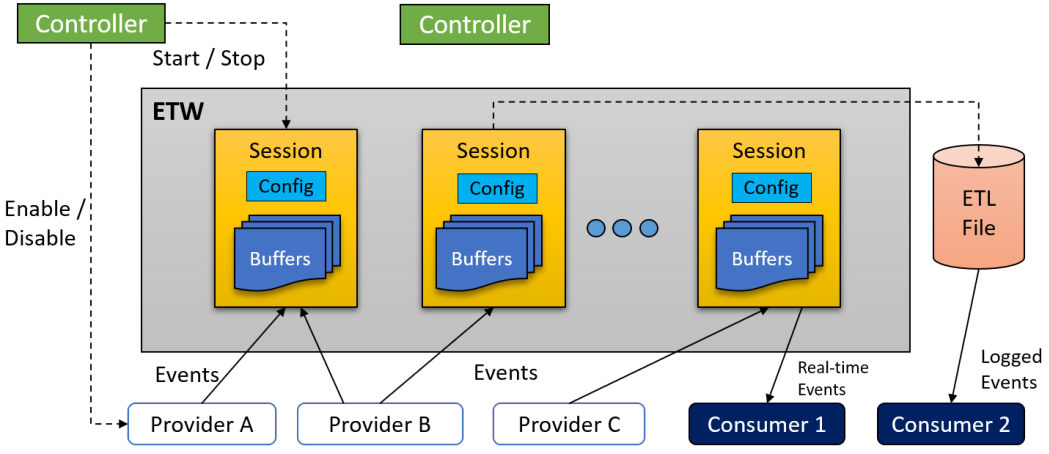


Figure 20-7: ETW architecture

ETW is a big topic that cannot be covered fully in this chapter (probably requires its own book). Consult the documentation and possibly other online resources for information not presented in this chapter.

Windows has a large set of built-in, registered providers. Each provider is identified by a GUID, but also has a friendly name for easier human consumption. ETW providers don't have to be registered, but if they are, APIs exist for enumerating them. Open a command window and type the following:

```
logman query providers
```

logman.exe is a built-in Windows tool to work with ETW. A long list of registered ETW providers is shown (more than 1000 providers), alphabetically sorted by name. Here is the beginning of the list (formatted slightly differently for printing):

```
Provider                                  GUID
-------------------------------------------------
.NET Common Language Runtime
    {E13C0D23-CCBC-4E12-931B-D9CC2EEE27E4}
ACPI Driver Trace Provider
    {DAB01D4D-2D48-477D-B1C3-DAAD0CE6F06B}
Active Directory Domain Services: SAM
    {8E598056-8993-11D2-819E-0000F875A064}
Active Directory: Kerberos Client
    {BBA3ADD2-C229-4CDB-AE2B-57EB6966B0C4}
Active Directory: NetLogon
    {F33959B4-DBEC-11D2-895B-00C04F79AB69}
ADODB.1
    {04C8A86F-3369-12F8-4769-24E484A9E725}
...
```

You can see a list of active ETW sessions on a system by going to the *Performance* MMC snapin (search for *Performance* or just run *perfmon*), and navigate to the *Data Collector Sets* node, and then to the *Event Trace Sessions* node (figure 20-8).

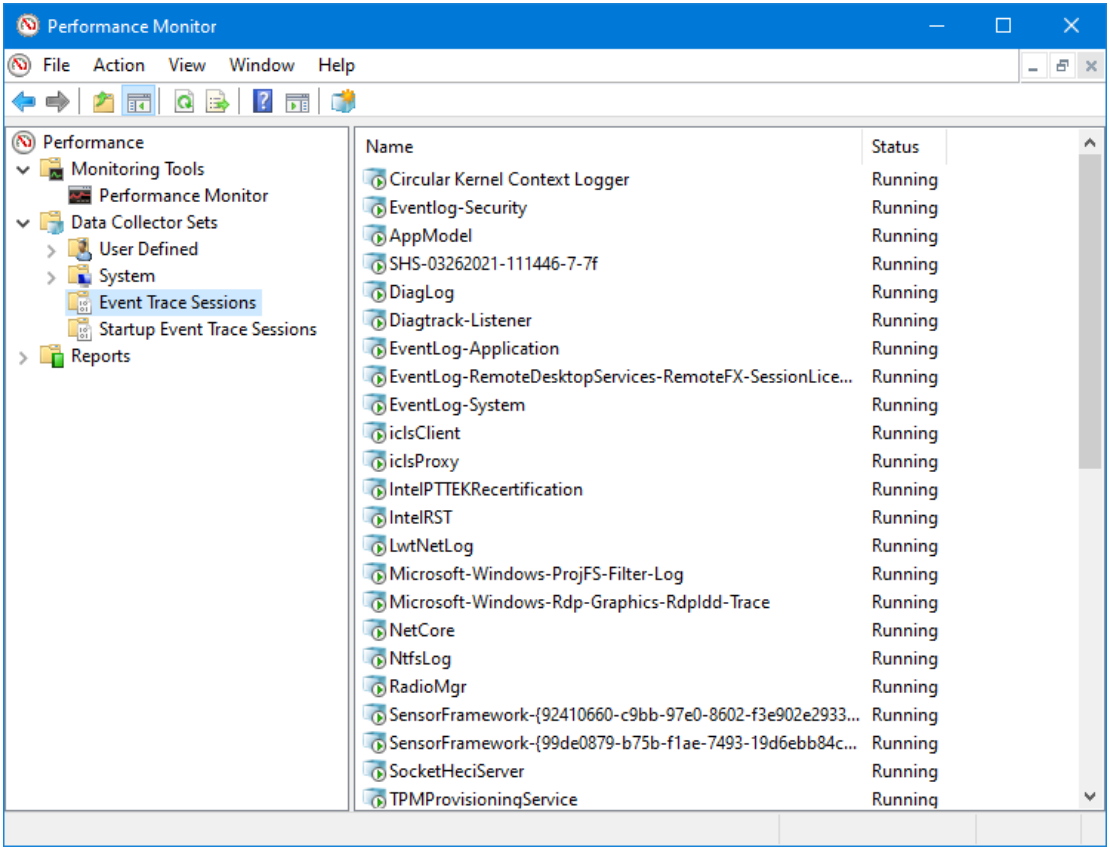


Figure 20-8: ETW sessions on the machine

You can retrieve the same information with `TdhEnumerateProviders`:

```
typedef struct _PROVIDER_ENUMERATION_INFO {
    ULONG NumberOfProviders;
    ULONG Reserved;
    TRACE_PROVIDER_INFO TraceProviderInfoArray[ANYSIZE_ARRAY];
} PROVIDER_ENUMERATION_INFO;
typedef PROVIDER_ENUMERATION_INFO *PPROVIDER_ENUMERATION_INFO;

TDHSTATUS TdhEnumerateProviders(
    _Out_ PPROVIDER_ENUMERATION_INFO pBuffer,
    _Inout_ ULONG* pBufferSize);
```

As is the case with other APIs, this one requires two calls. The first is with `pBuffer` set to `NULL` and `*pBufferSize` initialized to zero. The function returns the required number of bytes in `*pBufferSize`. Each provider is of type `TRACE_PROVIDER_INFO`:

```
typedef struct _TRACE_PROVIDER_INFO {
    GUID ProviderGuid;
    ULONG SchemaSource;
    ULONG ProviderNameOffset;
} TRACE_PROVIDER_INFO;
```

Each provider has its identifying GUID, its friendly name (provided as an offset from the beginning of the buffer), and where its schema format (0 for XML, 1 for WMI (*Windows Management Instrumentation*) MOF (*Managed Object Format*) - not important for the following discussion).

Many ETW functions are defined in *tdh.h* and require linking with *tdh.lib*. The following code, taken from the *EtwMeta* sample project displays all the registered providers, just as the previous *logman* invocation:

```
#include <Windows.h>
#include <tdh.h>
#include <string>
#include <stdio.h>
#include <memory>
#include <assert.h>
#include <vector>
#include <algorithm>

#pragma comment(lib, "tdh")

struct EtwProvider {
    std::wstring Name;
    GUID Guid;
};

std::vector<EtwProvider> EnumProviders() {
    std::vector<EtwProvider> providers;

    // first call with NULL and zero
    ULONG size = 0;
    auto error = ::TdhEnumerateProviders(nullptr, &size);
    assert(error == ERROR_INSUFFICIENT_BUFFER);

    // allocate with the required size
    auto buffer = std::make_unique<BYTE[]>(size);
    if (!buffer)
        return providers;

    auto data = reinterpret_cast<PROVIDER_ENUMERATION_INFO*>(
        buffer.get());
```

```
    // second call
    error = ::TdhEnumerateProviders(data, &size);
    assert(error == ERROR_SUCCESS);
    if (error != ERROR_SUCCESS)
        return providers;

    // build a vector of providers
    providers.reserve(names.size());
    for (ULONG i = 0; i < data->NumberOfProviders; i++) {
        const auto& item = data->TraceProviderInfoArray[i];
        EtwProvider provider;
        provider.Name.assign((PCWSTR)(buffer.get()
            + item.ProviderNameOffset));
        provider.Guid = item.ProviderGuid;
        providers.push_back(std::move(provider));
    }

    // sort vector by name
    std::sort(providers.begin(), providers.end(),
        [](const auto& p1, const auto& p2) {
        return ::_wcsicmp(p1.Name.c_str(), p2.Name.c_str()) < 0;
        });

    return providers;
}

int main() {
//...
    auto providers = EnumProviders();
    WCHAR sguid[64];
    for(auto& provider : providers) {
        ::StringFromGUID2(provider.Guid, sguid, _countof(sguid));
        printf("%-50ws %ws\n", provider.Name.c_str(), sguid);
    }
    printf("%u Providers.\n", (uint32_t)providers.size());

//...
```

My machine lists 1192 providers.

Every provider exposes a set of events, each of which has an optional set of properties. Providers that provide a manifest can be queried statically with those details retrieved. Retrieving the events of a provider that has a manifest is possible with `TdhEnumerateManifestProviderEvents` (Windows 8.1 and later):

```
TDHSTATUS TdhEnumerateManifestProviderEvents(
    _In_ LPGUID ProviderGuid,
    _Out_ PPROVIDER_EVENT_INFO Buffer,
    _Inout_ ULONG* BufferSize);
```

ProviderGuid is the GUID of the provider, identifying it uniquely. As with `TdhEnumerateProviders`, `TdhEnumerateManifestProviderEvents` should be called twice: first, to get the required buffer size, allocate the buffer, and then a second call to get the data. Each event is represented by a `PROVIDER_EVENT_INFO` structure:

```
typedef struct _EVENT_DESCRIPTOR {
    USHORT      Id;
    UCHAR       Version;
    UCHAR       Channel;
    UCHAR       Level;
    UCHAR       Opcode;
    USHORT      Task;
    ULONGLONG   Keyword;
} EVENT_DESCRIPTOR, *PEVENT_DESCRIPTOR;

typedef struct _PROVIDER_EVENT_INFO {
    ULONG NumberOfEvents;   // number of events
    ULONG Reserved;
    EVENT_DESCRIPTOR EventDescriptorsArray[ANYSIZE_ARRAY];
} PROVIDER_EVENT_INFO;
typedef PROVIDER_EVENT_INFO *PPROVIDER_EVENT_INFO;
```

Read the `EVENT_DESCRIPTOR` structure comments in its header file (*evntprov.h*), which explains all its members in detail.

The following code taken from the *ETWMeta* project shows how to query the events of a selected provider:

```
bool DumpProviderEvents(const GUID& guid) {
    ULONG size = 0;
    auto error = ::TdhEnumerateManifestProviderEvents(
        (LPGUID)&guid, nullptr, &size);
    if (error != ERROR_INSUFFICIENT_BUFFER)
        return false;

    auto buffer = std::make_unique<BYTE[]>(size);
    if (!buffer)
        return false;

    auto data = reinterpret_cast<PROVIDER_EVENT_INFO*>(buffer.get());
    error = ::TdhEnumerateManifestProviderEvents(
        (LPGUID)&guid, data, &size);
    if (error != ERROR_SUCCESS)
        return false;

    printf("Events: %u\n", data->NumberOfEvents);
    for(ULONG i = 0; i < data->NumberOfEvents; i++) {
        auto& info = data->EventDescriptorsArray[i];
        DumpEventInfo(guid, info);  // see next code segment
    }
    return true;
}
```

Each event may expose its data via properties, that can be enumerated with `TdhGetManifestEventInformation`:

```
TDHSTATUS TdhGetManifestEventInformation(
    _In_    LPGUID ProviderGuid,
    _In_    PEVENT_DESCRIPTOR EventDescriptor,
    _Out_   PTRACE_EVENT_INFO Buffer,
    _Inout_ ULONG* BufferSize);
```

You can probably guess the meaning of *ProviderGuid* and *EventDescriptor*, representing the provider and the event of interest, respectively. As with the previous functions, the returned buffer contains an array of `TRACE_EVENT_INFO` structures detailing event information, including properties:

```
typedef enum _TEMPLATE_FLAGS {
    TEMPLATE_EVENT_DATA = 1,  // custom xml is not specified
    TEMPLATE_USER_DATA = 2,   // custom xml is specified
    TEMPLATE_CONTROL_GUID = 4 // EventGuid contains the manifest
                              // control GUID
} TEMPLATE_FLAGS;

typedef struct _TRACE_EVENT_INFO {
    GUID ProviderGuid;
    GUID EventGuid;
    EVENT_DESCRIPTOR EventDescriptor;
    DECODING_SOURCE DecodingSource;
    ULONG ProviderNameOffset;
    ULONG LevelNameOffset;
    ULONG ChannelNameOffset;
    ULONG KeywordsNameOffset;
    ULONG TaskNameOffset;
    ULONG OpcodeNameOffset;
    ULONG EventMessageOffset;
    ULONG ProviderMessageOffset;
    ULONG BinaryXMLOffset;
    ULONG BinaryXMLSize;

    union {
        ULONG EventNameOffset;
        ULONG ActivityIDNameOffset;
    };

    union {
        ULONG EventAttributesOffset;
        ULONG RelatedActivityIDNameOffset;
    };

    ULONG PropertyCount;
    _Field_range_(0, PropertyCount)
    ULONG TopLevelPropertyCount;
    union {
        TEMPLATE_FLAGS Flags;
        struct {
            ULONG Reserved : 4; // TEMPLATE_FLAGS values
            ULONG Tags : 28;
        };
    };
```

```
    _Field_size_(PropertyCount)
    EVENT_PROPERTY_INFO EventPropertyInfoArray[ANYSIZE_ARRAY];
} TRACE_EVENT_INFO;
typedef TRACE_EVENT_INFO *PTRACE_EVENT_INFO;
```

As you can see, there is quite a bit of information here. The declaration of `TRACE_EVENT_INFO` in the header file (*tdh.h*) is heavily documented, definitely worth a read. Going through every detail is beyond the scope of this chapter.

An event has its own attributes and an optional array of properties. Here is some code that dumps the event attributes and calls another helper function to dump properties:

```
void DumpEventDescriptor(const EVENT_DESCRIPTOR& desc) {
    printf("Id: %u (0x%X) Version: %u Channel: %u Level: %u Opcode:"
        " %u Task: %u Keyword: 0x%llX\n",
        desc.Id, desc.Id, desc.Version, desc.Channel, desc.Level,
        desc.Opcode, desc.Task, desc.Keyword);
}

bool DumpEventInfo(const GUID& guid, const EVENT_DESCRIPTOR& desc) {
    DumpEventDescriptor(desc);

    ULONG size = 0;
    auto error = ::TdhGetManifestEventInformation((LPGUID)&guid,
        (PEVENT_DESCRIPTOR)&desc, nullptr, &size);
    if(error != ERROR_INSUFFICIENT_BUFFER)
        return false;

    auto buffer = std::make_unique<BYTE[]>(size);
    if (!buffer)
        return false;

    auto data = reinterpret_cast<TRACE_EVENT_INFO*>(buffer.get());
    error = ::TdhGetManifestEventInformation((LPGUID)&guid,
        (PEVENT_DESCRIPTOR)&desc, data, &size);
    if (ERROR_SUCCESS != error)
        return false;

    if(data->EventNameOffset)
        printf("Event Name: %ws\n",
            (PCWSTR)(buffer.get() + data->EventNameOffset));
    if(data->KeywordsNameOffset)
        printf("Keyword: %ws\n",
            (PCWSTR)(buffer.get() + data->KeywordsNameOffset));
```

```
    if(data->TaskNameOffset)
        printf("Task: %ws\n",
            (PCWSTR)(buffer.get() + data->TaskNameOffset));
    if(data->ChannelNameOffset)
        printf("Channel: %ws\n",
            (PCWSTR)(buffer.get() + data->ChannelNameOffset));
    if(data->LevelNameOffset)
        printf("Level: %ws\n",
            (PCWSTR)(buffer.get() + data->LevelNameOffset));
    if(data->OpcodeNameOffset)
        printf("Opcode: %ws\n",
            (PCWSTR)(buffer.get() + data->OpcodeNameOffset));
    if(data->ProviderMessageOffset)
        printf("Provider Message: %ws\n",
            (PCWSTR)(buffer.get() + data->ProviderMessageOffset));
    if(data->EventMessageOffset)
        printf("Event Message:\n%ws\n",
            (PCWSTR)(buffer.get() + data->EventMessageOffset));

    printf("Property Count: %u\n", data->PropertyCount);
    for(ULONG i = 0; i < data->TopLevelPropertyCount; i++) {
        auto& prop = data->EventPropertyInfoArray[i];
        DumpPropertyInfo(buffer.get(), prop);
    }
    printf("\n");
    return true;
}
```

All the strings are described by an offset from the beginning of the returned buffer, so careful pointer management is important. If an offset is zero, it indicates no string for that element.

The last piece of the puzzle are the properties. Each property is described by the `EVENT_PROPERTY_INFO` structure, accessible from the `EventPropertyInfoArray` array member from `TRACE_EVENT_INFO`:

```
typedef enum _PROPERTY_FLAGS {
   PropertyStruct           = 0x1,  // Type is struct
   PropertyParamLength      = 0x2,  // Length is index of param with len\
gth
   PropertyParamCount       = 0x4,  // Count is index of param with count
   PropertyWBEMXmlFragment  = 0x8,  // WBEM extension flag for property
   PropertyParamFixedLength = 0x10, // Length of the parameter is fixed
   PropertyParamFixedCount  = 0x20, // Count of the parameter is fixed
   PropertyHasTags          = 0x40, // The Tags field has been initializ\
ed
```

```
   PropertyHasCustomSchema  = 0x80, // Type is described with a custom s\
chema
} PROPERTY_FLAGS;

typedef struct _EVENT_PROPERTY_INFO {
    PROPERTY_FLAGS Flags;
    ULONG NameOffset;
    union {
        struct _nonStructType {
            USHORT InType;
            USHORT OutType;
            ULONG MapNameOffset;
        } nonStructType;
        struct _structType {
            USHORT StructStartIndex;
            USHORT NumOfStructMembers;
            ULONG padding;
        } structType;
        struct _customSchemaType {
            USHORT InType;
            USHORT OutType;
            ULONG CustomSchemaOffset;
        } customSchemaType;
    };
    union {
        USHORT count;
        USHORT countPropertyIndex;
    };
    union {
        USHORT length;
        USHORT lengthPropertyIndex;
    };
    union {
        ULONG Reserved;
        struct {
            ULONG Tags : 28;
        };
    };
} EVENT_PROPERTY_INFO;
typedef EVENT_PROPERTY_INFO *PEVENT_PROPERTY_INFO;
```

Properties can be structure types or simple types. A property's name is pointed to by the `NameOffset` member. The most common property type is a non-structure one

(`nonStructType` member). `InType` (of type `TDH_IN_TYPE`) and `OutType` (`TDH_OUT_TYPE`) represent its type. `OutType` is more like a recommendation of a good way to interpret the `InType`, which is more fundamental. For many types, these are the same, but could be different if a more explicit representation exists. For example, a `TDH_INTYPE_UNICODESTRING` could be mapped to a generic string `TDH_OUTTYPE_STRING`, but could be more specific, such as `TDH_OUTTYPE_JSON` (a string formatted as JSON).

The header file describing `TDH_IN_TYPE` and `TDH_OUT_TYPE` is heavily commented, explaining the different types.

Here is a function used to dump a simple property's information:

```
void DumpPropertyInfo(const BYTE* buffer,
    const EVENT_PROPERTY_INFO& info) {
    printf("\tProperty name: %ws (Flags: %ws)\n",
        (PCWSTR)(buffer + info.NameOffset),
        PropertyFlagsToString(info.Flags).c_str());
    if ((info.Flags & PropertyStruct) == 0) {
        // handle simple properties only
        auto inType = info.nonStructType.InType;
        auto outType = info.nonStructType.OutType;
        std::wstring mapName;
        // optional logical property mapping
        if (info.nonStructType.MapNameOffset)
            mapName = (PCWSTR)(buffer
                + info.nonStructType.MapNameOffset);
        printf("\t\tIn: %s Out: %s%ws\n",
            InTypeToString(
                inType).c_str(),
                OutTypeToString(outType).c_str(),
                mapName.empty() ? L"" : (L" (" + mapName + L")").c_str()\
);
    }
}
```

`PropertyFlagsToString`, `InTypeToString`, and `OutTypeToString` are simple helpers to translate enumerations to strings. You can find them as part of the *ETWMeta* project.

The *ETWMeta* tool can be invoked without any arguments, in which case it lists all the providers. If an argument is provided, it's interpreted as a provider name of a GUID, causing display of the information for the provider, its events and their properties using the functions shown earlier. The missing piece is the part of searching for the requested provider and invoking `DumpProviderEvents`:

```
bool DisplayProviderInfo(const std::wstring& name) {
    //
    // retrieve all providers
    //
    auto providers = EnumProviders();
    WCHAR sguid[64];
    //
    // lambda function to match provider name or GUID
    //
    auto findByNameOrGuid = [&](auto& p) {
        if(_wcsicmp(p.Name.c_str(), name.c_str()) == 0)
            return true;
        ::StringFromGUID2(p.Guid, sguid, _countof(sguid));
        return _wcsicmp(name.c_str(), sguid) == 0;
    };

    //
    // get the requested provider (if any)
    //
    auto it = std::find_if(providers.begin(), providers.end(),
        findByNameOrGuid);
    if(it == providers.end())
        return false;

    auto& provider = *it;
    printf("Provider Name: %ws\n", (PCWSTR)provider.Name.c_str());
    ::StringFromGUID2(provider.Guid, sguid, _countof(sguid));
    printf("Provider GUID: %ws\n", sguid);

    if(!DumpProviderEvents(provider.Guid))
        printf("No event information provided\n");
    return true;
}
```

Here is some partial output running *ETWMeta* with a provider:

```
C:\>EtwMeta.exe Microsoft-Windows-Kernel-File
Provider Name: Microsoft-Windows-Kernel-File
Provider GUID: {EDD08927-9CC4-4E65-B970-C2560FB5C289}
Events: 43
Id: 10 (0xA) Version: 0 Channel: 16 Level: 4 Opcode: 0 Task: 10
    Keyword: 0x8000000000000010
Keyword: KERNEL_FILE_KEYWORD_FILENAME
Task: NameCreate
Channel: Microsoft-Windows-Kernel-File/Analytic
Level: Information
Opcode: Info
Provider Message: Microsoft-Windows-Kernel-File
Property Count: 2
        Property name: FileKey (Flags: None)
                In: Pointer Out: Hex Int64
        Property name: FileName (Flags: None)
                In: Unicode String Out: String

Id: 11 (0xB) Version: 0 Channel: 16 Level: 4 Opcode: 0 Task: 11
    Keyword: 0x8000000000000010
Keyword: KERNEL_FILE_KEYWORD_FILENAME
Task: NameDelete
Channel: Microsoft-Windows-Kernel-File/Analytic
Level: Information
Opcode: Info
Provider Message: Microsoft-Windows-Kernel-File
Property Count: 2
        Property name: FileKey (Flags: None)
                In: Pointer Out: Hex Int64
        Property name: FileName (Flags: None)
                In: Unicode String Out: String
...
```

Try running it with other providers. For some providers, you'll get a failure. For example:

```
C:\>EtwMeta.exe "Windows Kernel Trace"
Provider Name: Windows Kernel Trace
Provider GUID: {9E814AAD-3204-11D2-9A82-006008A86939}
No event information provided
```

Some providers just don't have any a manifest describing their events "offline". The only way to know about their events is with "standard" documentation, or during runtime, when an event of that provider is consumed (more on that in the next section).

Still, many providers do provide manifests that can be queried "offline". A few years back I created a tool called *ETWExplorer* that allows browsing this information in an easier way than through a command line tool. It can be downloaded from my Github repo at https://github.com/zodiacon/EtwExplorer. When run, select *Open provider...* and you'll be presented with a list of all registered providers (figure 20-9).

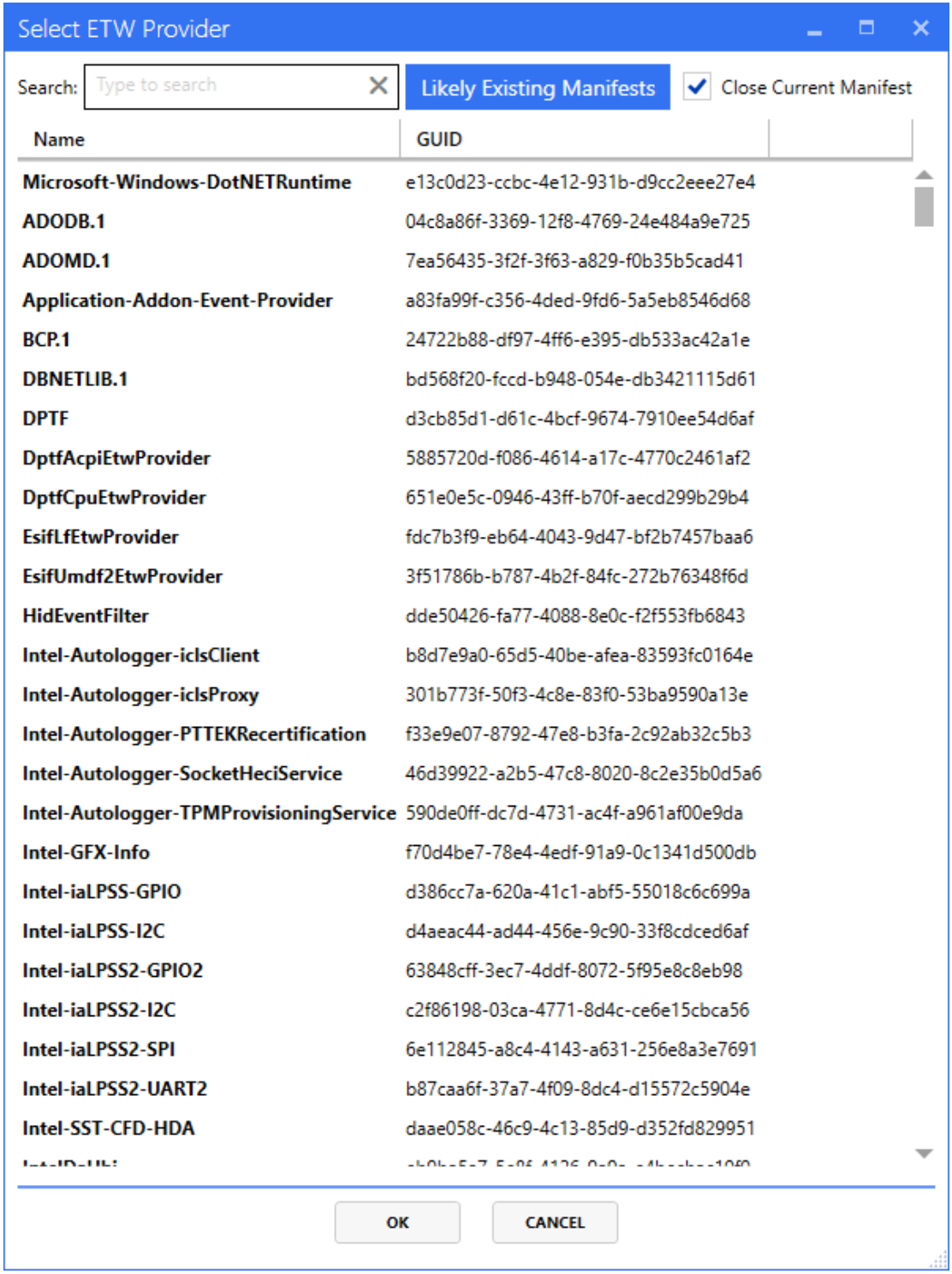


Figure 20-9: *ETWExplorer* provider search

You can type to quickly search for a provider by name. Click *OK* to open the provider's manifest (if available). You'll see a set of tabs with information from the manifest. The most interesting is the *Events* tab, where you'll see the list of events. Selecting an event opens up its properties (figure 20-10).

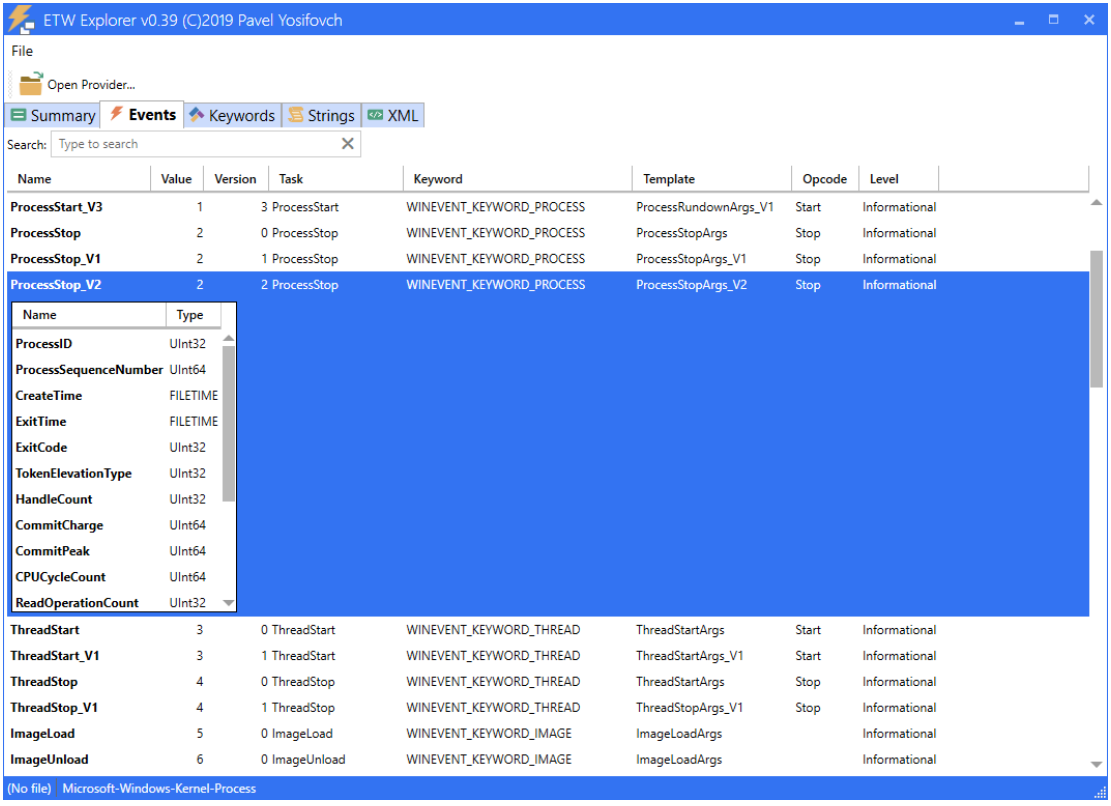


Figure 20-10: *ETWExplorer* showing events

There is a lot of interesting providers and events. *ETWExplorer* is a useful tool to explore the registered providers.

The tool is written in C#, the UI implemented in WPF. I expect to write an extended tool in C++ (that may already exist by the time you read this).

Creating ETW Sessions

The metadata aspect of ETW is required to correctly interpret traces. Now it's time to look at creating ETW sessions, so traces can be captured and later analyzed. Windows supports up to 64 ETW sessions

running at the same time. Many such sessions already exist on a typical system (figure 20-8), which is why you should strive to have a single session running at a time.

Working with ETW sessions, especially enabling providers, requires the user be part of one of the following groups: Administrators or Performance Log Users. Also, services running with the *LocalSystem*, *LocalService* or *NetworkService* accounts are allowed to do so as well.

Creating a session requires calling the `StartTrace` API:

```
ULONG StartTrace (
    _Out_   PTRACEHANDLE TraceHandle,
    _In_    LPCTSTR InstanceName,
    _Inout_ PEVENT_TRACE_PROPERTIES Properties);
```

Calling `StartTrace` may return error 1450 (`ERROR_NO_SYSTEM_RESOURCES`). This means the system has reached the maximum ETW sesssions of 64. Windows 10 version 1709 allows increasing this number up to 256 (or lowering to a minimum of 32). To do that, create a `DWORD` value under the key *HKLM\SYSTEM\CurrentControlSet\Control\WMI* and set it to the desired value. You'll have to restart the system for the change to take effect.

This innocent-looking function is not at all trivial because of the `EVENT_TRACE_PROPERTIES` structure that is expected to be filled correctly. This is its definition (the comments are from the header file itself), along with its first member's type, `WNODE_HEADER`:

```
typedef struct _WNODE_HEADER {
    ULONG BufferSize;  // Size of entire buffer including this ULONG
    ULONG ProviderId;  // Provider Id of driver returning this buffer
    union {
        ULONG64 HistoricalContext;  // Logger use
        struct
            {
            ULONG Version;       // Reserved
            ULONG Linkage;       // Linkage field reserved for WMI
        };
    };
    union {
        ULONG CountLost;         // Reserved
        HANDLE KernelHandle;     // Kernel handle for data block
        LARGE_INTEGER TimeStamp; // Timestamp as returned in units
                                 // of 100ns since 1/1/1601
    };
```

```
    GUID Guid;      // Guid for data block returned with results
    ULONG ClientContext;
    ULONG Flags;
} WNODE_HEADER, *PWNODE_HEADER;

typedef struct _EVENT_TRACE_PROPERTIES {
    WNODE_HEADER Wnode;

    ULONG BufferSize;           // buffer size for logging (kbytes)
    ULONG MinimumBuffers;       // minimum to preallocate
    ULONG MaximumBuffers;       // maximum buffers allowed
    ULONG MaximumFileSize;      // maximum logfile size (in MBytes)
    ULONG LogFileMode;          // sequential, circular
    ULONG FlushTimer;           // buffer flush timer, in seconds
    ULONG EnableFlags;          // trace enable flags
    union {
        LONG  AgeLimit;         // unused
        LONG  FlushThreshold;   // Number of buffers to fill before flus\
hing
    };
    ULONG NumberOfBuffers;      // no of buffers in use
    ULONG FreeBuffers;          // no of buffers free
    ULONG EventsLost;           // event records lost
    ULONG BuffersWritten;       // no of buffers written to file
    ULONG LogBuffersLost;       // no of logfile write failures
    ULONG RealTimeBuffersLost;  // no of rt delivery failures
    HANDLE LoggerThreadId;      // thread id of Logger
    ULONG LogFileNameOffset;    // Offset to LogFileName
    ULONG LoggerNameOffset;     // Offset to LoggerName
} EVENT_TRACE_PROPERTIES, *PEVENT_TRACE_PROPERTIES;
```

Instead of going through every member, we'll see how to use the most common settings to configure a session with the results written to a file, and then we'll see how to consume a session from a file or in real-time. The tool we'll build will allow creation of an ETW session with (almost) any set of providers, and run it. The project's name is *RunETW* in the code samples solution for this chapter.

RunETW is a console application. The `main` function should accept command-line arguments to configure subsequent operations:

```
int wmain(int argc, const wchar_t* argv[]) {
    if(argc < 2) {
        printf(
            "Usage: RunETW [-o path] [-r] <provider1> [provider2 ... ]\n\
");
        return 0;
    }

    PCWSTR filename = nullptr;
    bool realTime = false;

    std::vector<PCWSTR> names;

    for(int i = 1; i < argc; i++) {
        if(::_wcsicmp(argv[i], L"-o") == 0) {
            if(argc == i + 1) {
                printf("Missing file name\n");
                return 1;
            }
            filename = argv[i + 1];
            i++;
        }
        else if(::_wcsicmp(argv[i], L"-r") == 0) {
            realTime = true;
        }
        else {
            names.push_back(argv[i]);
        }
    }

    if(filename == nullptr && !realTime) {
        printf("You must specify -o or -r or both\n");
        return 1;
    }

    if(names.empty()) {
        printf("No providers specified\n");
        return 1;
    }
```

The command-line arguments require specifying a file name (with the `-o` switch), and/or the `-r` switch for a real-time session. Specifying both is acceptable. In this section we'll focus on a file target, and look at a real-time session later in this chapter. Following the switches are provider names (at least one, otherwise

the session is pretty useless).

Next, we need to take the vector of provider names and turn them into GUIDs. This is required because the APIs used to configure providers work with provider GUIDs rather than the human-readable names. Once the GUIDs are retrieved, the session can be built and run. Here is the last piece of `main`:

```
    auto providers = GetProviders(names);
    if(providers.size() < names.size()) {
        printf("Not all providers found");
        return 1;
    }

    if(!RunSession(providers, filename, realTime)) {
        printf("Failed to run session\n");
        return 1;
    }

    return 0;
}
```

Two functions remain: `GetProviders` turns the vector of names into a vector of GUIDs. There is no straightforward function to do that, so we have to iterate over the registered providers, and compare names, collecting the GUIDs along the way. Here is `GetProviders`:

```
std::vector<GUID> GetProviders(std::vector<PCWSTR> names) {
    std::vector<GUID> providers;

    ULONG size = 0;
    auto error = ::TdhEnumerateProviders(nullptr, &size);
    assert(error == ERROR_INSUFFICIENT_BUFFER);

    // allocate with the required size
    auto buffer = std::make_unique<BYTE[]>(size);
    if(!buffer)
        return providers;

    auto data = reinterpret_cast<PROVIDER_ENUMERATION_INFO*>(
        buffer.get());
    // second call
    error = ::TdhEnumerateProviders(data, &size);
    assert(error == ERROR_SUCCESS);
    if(error != ERROR_SUCCESS)
        return providers;
```

```
    // build a vector of providers
    providers.reserve(names);
    int found = 0;
    for(ULONG i = 0; i < data->NumberOfProviders &&
        found < names.size(); i++) {
        const auto& item = data->TraceProviderInfoArray[i];
        auto name = (PCWSTR)(buffer.get() + item.ProviderNameOffset);
        for(auto n : names) {
            if(_wcsicmp(name, n) == 0) {
                providers.push_back(item.ProviderGuid);
                found++;
                break;
            }
        }
    }

    return providers;
}
```

The function is very similar to the `EnumProviders` example from the previous section. The loop is different, as its purpose is to compare the provided names to those retrieved with `TdhEnumerateProviders`. The vector of GUIDs is returned to `main`. `main` makes sure all names were successfully converted to GUIDs, otherwise it complains and exits.

The interesting function is `RunSession` that does all the work of creating, configuring and running the session based on the supplied provider GUIDs and the file name and/or real-time flag.

Every ETW session must have a unique name and a session GUID. Here is the start of the function defining these values:

```
bool RunSession(const std::vector<GUID>& providers, PCWSTR filename,
    bool realTime) {
    // {7791FF3D-2E25-4C3B-B90D-E33D4ADA8A36}
    static const GUID sessionGuid =
    { 0x7791ff3d, 0x2e25, 0x4c3b,
        { 0xb9, 0xd, 0xe3, 0x3d, 0x4a, 0xda, 0x8a, 0x36 } };

    const WCHAR sessionName[] = L"Chapter20Session";
```

The GUID was generated by running the *Create GUID* tool, normally available under Visual Studio's *Tools* menu. Run the tool and select option 3 for the generated format, click *Copy* (figure 20-11), and then paste it in the code and override the dummy name with something sensible, `sessionGuid` in the example.

Chapter 21 has more information on GUIDs in general. For the purpose of this chapter, a GUID is a unique identifier defined as a structure, as it's 128 bit in size (no such native type in C/C++ as of this writing).

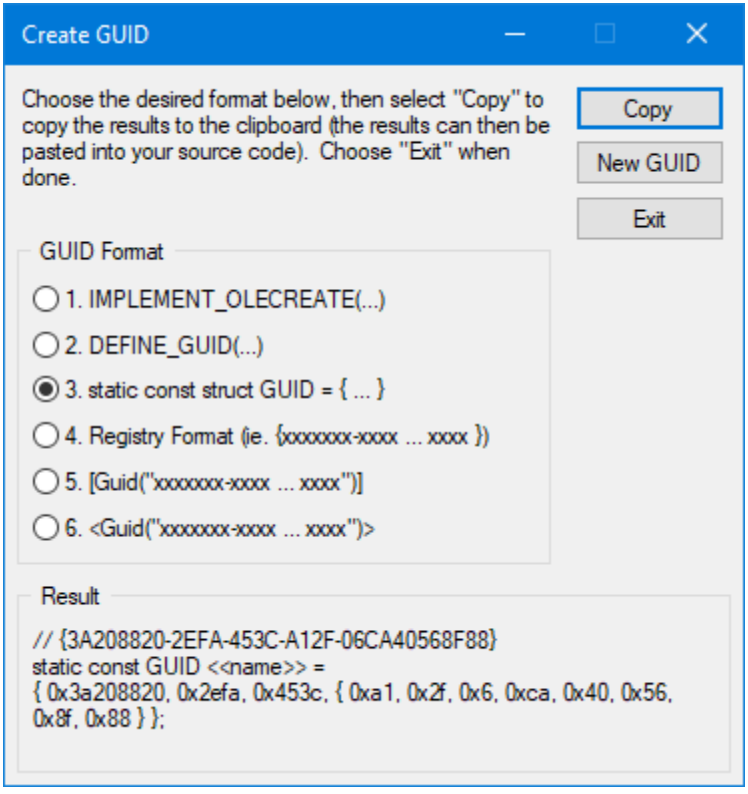


Figure 20-11: The *Create GUID* tool

The next step is calling `StartTrace`, and for that we need to fill in a `EVENT_TRACE_PROPERTIES` structure. It's not that simple, however, as after the bytes making up the structure in memory, the session name and the file name (if any) need to be stored. Figure 20-12 shows the expected layout.

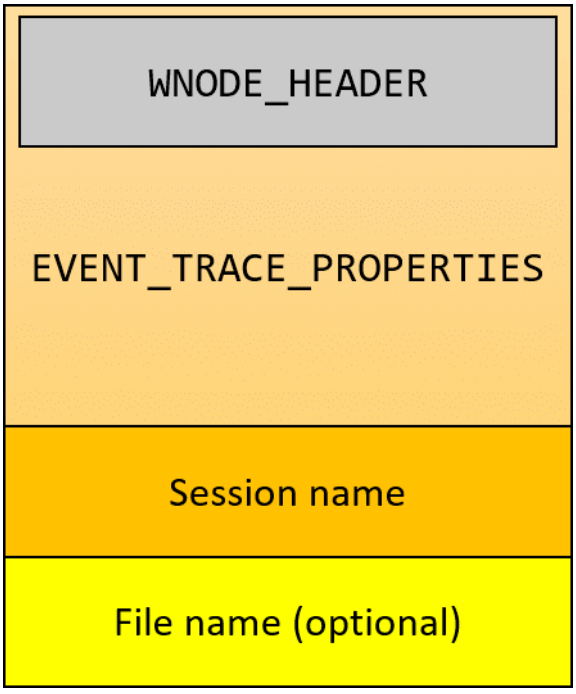


Figure 20-12: Layout of expected **`EVENT_TRACE_PROPERTIES`** to **`StartTrace`**

First, we calculate the size needed for the entire thing:

```
    auto size = sizeof(EVENT_TRACE_PROPERTIES)
        + (filename ? ((::wcslen(filename) + 1) * sizeof(WCHAR)) : 0)
        + sizeof(sessionName);
```

Notice the size is in bytes. The session name is declared as a simple static string, so the `sizeof` operator gives the number of bytes directly. `filename` is a pointer (that can be `NULL`), so `wcslen` has to be called, one added (for the terminated `NULL`), and multiplied by `sizeof(WCHAR)` to get to bytes.

Now we can allocate the buffer given the size:

```
auto buffer = std::make_unique<BYTE[]>(size);
if(!buffer)
    return false;
::ZeroMemory(buffer.get(), size);
```

We also zero out the size, so that members we don't fill will have a default value of zero.

`ZeroMemory` is a macro expanding to another macro, `RtlZeroMemory`, which simply calls `memset` with a zero value.

Next we start filling in the structure, string with the header:

```
auto props = reinterpret_cast<EVENT_TRACE_PROPERTIES*>(buffer.get());
props->Wnode.BufferSize = (ULONG)size;
props->Wnode.Flags = WNODE_FLAG_TRACED_GUID;
props->Wnode.ClientContext = 1;      // QueryPerformanceCounter
props->Wnode.Guid = sessionGuid;
```

The size specified must be the total size (as we calculated), rather than the size of the `EVENT_TRACE_PROPERTIES` structure only. The flag `WNODE_FLAG_TRACED_GUID` indicates the session is identified by a GUID (mandatory), with some additional flags available that are not typically used. The oddly named `ClientContext` indicates which timer to use when reporting event time stamps. The default (zero, also the same as one) is to use the high resolution timer normally provided by calling `QueryPerformanceCounter` (this is the recommended option). The last setting in `Wnode` is the session GUID itself.

Now we continue with filling in more details in the main structure:

```
props->LogFileMode = (filename ? EVENT_TRACE_FILE_MODE_SEQUENTIAL : 0)
    | (realTime ? EVENT_TRACE_REAL_TIME_MODE : 0);
props->MaximumFileSize = 100;        // 100 MB
props->LoggerNameOffset = sizeof(*props);
props->LogFileNameOffset =
    filename ? sizeof(*props) + sizeof(sessionName) : 0;
```

`LogFileMode` indicates where the data is to be recorded:

- `EVENT_TRACE_FILE_MODE_NONE` - don't use a file
- `EVENT_TRACE_FILE_MODE_SEQUENTIAL` - file is written to sequentially, up to the set up limit
- `EVENT_TRACE_FILE_MODE_CIRCULAR` - if the file reaches its size, older events are dropped to make room for new events
- `EVENT_TRACE_FILE_MODE_APPEND` - append to the file (sequential only)

Another flag that can be added is `EVENT_TRACE_REAL_TIME_MODE` to indicate a real-time session. In the code above, a sequential mode is selected if a file name is provided, along with the real-time flag, if so indicated.

`MaximumFileSize` sets the maximum file size (if there is a file of course), where the value is provided in `MB`. The example uses 100 MB. Finally, two offsets need to be set: the session name's and the file name's. The next lines of code will copy the session name right after the `EVENT_TRACE_PROPERTIES` structure, so the offset from the beginning of the structure of the session name is exactly `sizeof(EVENT_TRACE_PROPERTIES)`, written as `sizeof(*props)`, which is the same thing.

The file name (if any) will be copied right following the session name in memory, so its offset must take into account the session name's size.

Next, we need to copy the names as promised:

```
// copy session name
::wcscpy_s((PWSTR)(props + 1), ::wcslen(sessionName) + 1,
    sessionName);

// copy filename
if(filename)
    ::wcscpy_s((PWSTR)(buffer.get()
        + sizeof(*props) + sizeof(sessionName)),
        ::wcslen(filename) + 1, filename);
```

If you are confused as to why the above statements do the right thing, just remember that in pointer arithmetic in C/C++ add/subtract the number provided times the size of each pointed-to item. For example, adding 10 to a `int*` moves the pointer 40 bytes, while adding the same 10 to a `char*` moves the pointer 10 bytes only, even though both pointers were added the value `10`.

Finally we are ready to create the ETW session:

```
TRACEHANDLE hTrace = 0;
auto status = ::StartTrace(&hTrace, sessionName, props);
if(ERROR_SUCCESS != status)
    return false;
```

The session name must be provided as the second argument (even though it's also copied as part of the `EVENT_TRACE_PROPERTIES` structure). The result of the call is a `TRACEHANDLE`, which curiously enough, is typed as `ULONG64`, so initializing it with `nullptr` will not compile (but `NULL` will).

The session at this time is pretty useless because it enables zero providers. To enable (or later disable) a provider, call one of the following functions:

```
ULONG EnableTrace(
    _In_ ULONG Enable,
    _In_ ULONG EnableFlag,
    _In_ ULONG EnableLevel,
    _In_ LPCGUID ControlGuid,
    _In_ TRACEHANDLE TraceHandle);

ULONG EnableTraceEx(
    _In_ LPCGUID ProviderId,
    _In_opt_ LPCGUID SourceId,
    _In_ TRACEHANDLE TraceHandle,
    _In_ ULONG IsEnabled,
    _In_ UCHAR Level,
```

```
    _In_ ULONGLONG MatchAnyKeyword,
    _In_ ULONGLONG MatchAllKeyword,
    _In_ ULONG EnableProperty,
    _In_opt_ PEVENT_FILTER_DESCRIPTOR EnableFilterDesc);

ULONG EnableTraceEx2(
    _In_ TRACEHANDLE TraceHandle,
    _In_ LPCGUID ProviderId,
    _In_ ULONG ControlCode,
    _In_ UCHAR Level,
    _In_ ULONGLONG MatchAnyKeyword,
    _In_ ULONGLONG MatchAllKeyword,
    _In_ ULONG Timeout,
    _In_opt_ PENABLE_TRACE_PARAMETERS EnableParameters);
```

`EnableTrace` is the oldest API and is no longer recommended. It's equivalent to calling `EnableTraceEx` like so:

```
ULONG EnableTrace(ULONG Enable, ULONG EnableFlag, ULONG EnableLevel,
    LPCGUID ControlGuid, TRACEHANDLE TraceHandle) {
    return EnableTraceEx(
        ControlGuid, nullptr, TraceHandle,
        Enable, EnableLevel, EnableFlag,
        0, 0, nullptr);
}
```

`EnableTraceEx` itself is implemented in terms of `EnableTraceEx2` on Windows 7 and later. We will use `EnableTraceEx` like so:

```
for(auto& guid : providers) {
    status = ::EnableTraceEx(&guid, nullptr, hTrace, TRUE,
        TRACE_LEVEL_INFORMATION, 0, 0, 0, nullptr);
    if(ERROR_SUCCESS != status) {
        ::StopTrace(hTrace, sessionName, props);
        return false;
    }
}
```

The first parameter is the provider's GUID we want to enable or disable for the session. *SourceId* is an optional GUID that is passed to the provider's enable callback (out of scope for this chapter). Next is the trace handle itself. `IsEnabled` is `TRUE` to enable and `FALSE` to disable the provider.

The next parameter (*Level*) allows a controller to specify a basic filter for events of a certain severeness. You might recall that each ETW event may be associated with a level. This parameter can be one of

the following: `TRACE_LEVEL_CRITICAL` (1), `TRACE_LEVEL_ERROR` (2), `TRACE_LEVEL_WARNING` (3), `TRACE_LEVEL_INFORMATION` (4), or `TRACE_LEVEL_VERBOSE` (5). Specifying a certain level would result in events with that level or lower to be recorded.

You can use *logman query providers <providername>* to display the available keywords and levels (see also the section *More Metadata*, later in this chapter).

The next two parameters, `MatchAnyKeyword` and `MatchAllKeyword` allow certian filtering (if supported by the provider). `MatchAnyKeyword` specifies keyword bits to let through. If an event's keyword is zero, it is always reported. Specifying zero for `MatchAnyKeyowrd` allows all keywords for manifest-based providers, and all 1s for a classic provider (`0xffffffff`, 32 bits are enough for a classic provider).

For example, here is the output for *logman query providers microsoft-windows-kernel-file* as it relates to keywords (output reformatted for printing):

```
Provider                            GUID
------------------------------------------------------------------------
Microsoft-Windows-Kernel-File  {EDD08927-9CC4-4E65-B970-C2560FB5C289}

Value                Keyword              Description
------------------------------------------------------------------------
0x0000000000000010  KERNEL_FILE_KEYWORD_FILENAME
0x0000000000000020  KERNEL_FILE_KEYWORD_FILEIO
0x0000000000000040  KERNEL_FILE_KEYWORD_OP_END
0x0000000000000080  KERNEL_FILE_KEYWORD_CREATE
0x0000000000000100  KERNEL_FILE_KEYWORD_READ
0x0000000000000200  KERNEL_FILE_KEYWORD_WRITE
0x0000000000000400  KERNEL_FILE_KEYWORD_DELETE_PATH
0x0000000000000800  KERNEL_FILE_KEYWORD_RENAME_SETLINK_PATH
0x0000000000001000  KERNEL_FILE_KEYWORD_CREATE_NEW_FILE
0x8000000000000000  Microsoft-Windows-Kernel-File/Analytic
```

For such a provider, specifying `0xa0` for `MatchAnyKeyword` would cause the provider to report events where the keywords are *KERNEL_FILE_KEYWORD_FILEIO* (0x20) or *KERNEL_FILE_KEYWORD_CREATE* (0x80).

`MatchAllKeyword` only has meaning if `MatchAnyKeyword` is not zero, and can further limit the reported events. If it's not zero, it indicates which bits must be present to report an event. Continuing from the previous example, if `MatchAllkeyword` is 0xa0, only events with both keywords would pass through. Most events map to a single keyword, but some might have multiple keyword bits set.

The `EnableProperty` parameter indicates which extra information is provided with an event. It can be zero, or a combination flags - here are some common ones:

- `EVENT_ENABLE_PROPERTY_SID` - include the user's SID

- `EVENT_ENABLE_PROPERTY_TS_ID` - include the terminal session ID
- `EVENT_ENABLE_PROPERTY_STACK_TRACE` - include stack trace

This extra information is written to an event's extended data (see the next section for more details).

The last parameter to `EnableTraceEx` is an optional pointer to an `EVENT_FILTER_DESCRIPTOR` structure that allows more complex filtering to be used. We'll look at those in the section *Providing Events*, later in this chapter.

Now that `EnableTraceEx` has been called for each provider, events will start to come in. The session needs to remain open for as long as event capturing is needed. For our example, we'll wait until the user pressed ENTER before stopping the session and returning to `main`:

```
    printf("Session running... press ENTER to stop\n");

    char dummy[4];
    gets_s(dummy);

    ::StopTrace(hTrace, sessionName, props);

    return true;
}
```

`StopTrace` is the opposite of `StartTrace`, defined like so:

```
ULONG StopTrace (
    _In_ TRACEHANDLE TraceHandle,
    _In_opt_ LPCTSTR InstanceName,
    _Inout_ PEVENT_TRACE_PROPERTIES Properties);
```

It is essentially identical to `StartTrace`, except the trace handle is provided as input in the first parameter. Notice also that `EVENT_TRACE_PROPERTIES` must be provided, so this structure must be kept alive at least until the call to `StopTrace`.

Now we can test our application. Open an elevated command window and navigate to the executable path:

```
C:\>RunETW.exe -o c:\temp\test.etl microsoft-windows-kernel-file
    microsoft-windows-kernel-process
Session running... press ENTER to stop
```

Once run, I have pressed ENTER after a few seconds. In between, I used Explorer and launched Notepad. The final file (*test.etl*) was about 30 MB in size. The ETL (*Event Tracing Log*) is the standard extension for an ETW run result.

What can we do with this file? It's in some binary format that we can convert to something more readable. One way of doing that is with another built-in Windows tool called *tracerpt*. For example:

```
C:\temp>tracerpt test.etl -lr -o test.xml -summary

Input
----------------
File(s):
     test.etl

Output
----------------
DumpFile:           test.xml

The command completed successfully.
```

The result of the previous command generated two files: *summary.txt* and *test.xml*. Here are the first lines in *summary.txt* (with some reformatting):

```
Files Processed:
    test.etl
Total Buffers Processed 483
Total Events  Processed 268016
Total Events  Lost      0
Start Time              Saturday, April 10, 2021
End Time                Saturday, April 10, 2021
Elapsed Time            19 sec
+-------------------------------------------------------------------+
|Event Count   Event Name           Task     Opcode   Version  Guid |
+-------------------------------------------------------------------+
|  1   EventTrace       0   PartitionInfoExtension   2
    {68fdd900-4a3e-11d1-84f4-0000f80464e3}|
|  1   EventTrace       0   Header                   2
    {68fdd900-4a3e-11d1-84f4-0000f80464e3}|
|  9   Microsoft-Windows-Kernel-File     26  Info  1 {edd08927-9cc4-4e65\
-...}|
|  9   Microsoft-Windows-Kernel-File     18  Info  1 {edd08927-9cc4-4e65\
-...}|
| 35   Microsoft-Windows-Kernel-File     11  Info  0 {edd08927-9cc4-4e65\
-...}|
| 10   Microsoft-Windows-Kernel-File     27  Info  1 {edd08927-9cc4-4e65\
-...}|
| 10   Microsoft-Windows-Kernel-File     19  Info  1 {edd08927-9cc4-4e65\
-...}|
| 70   Microsoft-Windows-Kernel-File     10  Info  0 {edd08927-9cc4-4e65\
-...}|
```

```
| 24   Microsoft-Windows-Kernel-File     30  Info  1 {edd08927-9cc4-4e65\
-...}|
|398   Microsoft-Windows-Kernel-File     20  Info  1 {edd08927-9cc4-4e65\
-...}|
...


|8183   Mind-Kernel-File                 12  Info  1 {edd08927-9cc4-4e65\
-...}|
|106238   Microsoft-Windows-Kernel-File  24  Info  0 {edd08927-9cc4-4e65\
-...}|
|174   Microsoft-Windows-Kernel-File     25  Info  1 {edd08927-9cc4-4e65\
-...}|
|65123   Microsoft-Windows-Kernel-File   15  Info  1 {edd08927-9cc4-4e65\
-...}|
|  3   Microsoft-Windows-Kernel-Process  11  Stop  1 {22fb2cd6-0e7b-422b\
-...}|
|  4   Microsoft-Windows-Kernel-Process  13  Start 0 {22fb2cd6-0e7b-422b\
-...}|
|  6   Microsoft-Windows-Kernel-Process  1   Start 3 {22fb2cd6-0e7b-422b\
-...}|
|  2   Microsoft-Windows-Kernel-Process  11  Start 1 {22fb2cd6-0e7b-422b\
-...}|
|  5   Microsoft-Windows-Kernel-Process  14  Stop  0 {22fb2cd6-0e7b-422b\
-...}|
|  6   Microsoft-Windows-Kernel-Process  2   Stop  2 {22fb2cd6-0e7b-422b\
-...}|
...
```

This file is a brief summary of the information available in *test.etl*. Notice the large number of events collected (268016) in just 19 seconds. The file shows the number of events of each type collected.

The actual event data has been converted to XML format and stored in *test.xml*. Other formats available are CSV (*Comma Separated Values*) and EVTX (used by the Event Log viewer) (specified with the `-of` switch to *tracerpt*). The XML file generated is about 336 MB and contains almost 9 million lines of XML! This should give you an idea of the amount of information available through ETW. The difficult aspect of ETW is making sense of the data, getting to the relevant information for the scenario at hand.

Searching for *notepad* within the XML file yields 105 results. Here is one event involving *notepad*:

```
<Event xmlns="http://schemas.microsoft.com/win/2004/08/events/event">
    <System>
        <Provider Name="Microsoft-Windows-Kernel-File"
            Guid="{edd08927-9cc4-4e65-b970-c2560fb5c289}" />
        <EventID>20</EventID>
        <Version>1</Version>
        <Level>4</Level>
        <Task>20</Task>
        <Opcode>0</Opcode>
        <Keywords>0x8000000000000020</Keywords>
        <TimeCreated
            SystemTime="2021-04-10T18:04:57.011091700+02:59" />
        <Correlation
            ActivityID="{00000000-0000-0000-0000-000000000000}" />
        <Execution ProcessID="16216" ThreadID="32292" ProcessorID="9"
            KernelTime="165" UserTime="15" />
        <Channel>Microsoft-Windows-Kernel-File/Analytic</Channel>
        <Computer />
    </System>
    <EventData>
        <Data Name="Irp">0xFFFF9C8BAA907908</Data>
        <Data Name="FileObject">0xFFFF9C8BAB2C3AC0</Data>
        <Data Name="FileKey">0xFFFFE70C6CAEA7C0</Data>
        <Data Name="IssuingThreadId">   32292</Data>
        <Data Name="Length">     616</Data>
        <Data Name="InfoClass">       3</Data>
        <Data Name="FileIndex">       0</Data>
        <Data Name="FileName">notepad&amp;quot;*</Data>
    </EventData>
    <RenderingInfo Culture="en-US">
        <Level>Information </Level>
        <Opcode>Info </Opcode>
        <Keywords>
            <Keyword>KERNEL_FILE_KEYWORD_FILEIO</Keyword>
        </Keywords>
        <Task>DirEnum</Task>
        <Channel>Microsoft-Windows-Kernel-File/Analytic</Channel>
        <Provider>Microsoft-Windows-Kernel-File </Provider>
    </RenderingInfo>
</Event>
```

The XML is nice enough to contain some of the event's metadata in readable form, such as the keyword (`KERNEL_FILE_KEYWORD_FILEIO`) and task (`DirEnum`) that saves us from going to the metadata to

make the correlation ourselves. Using a CSV file might be somewhat easier, as it can be open by a tool such as *Microsoft Excel*:

```
C:\temp>tracerpt test.etl -of CSV -o test.csv
```

The resulting CSV file is about 105 MB in size. Opening it in *Excel* shows the events in a table form, each row showing an event's properties.

Yet another option for viewing the data is by opening it with the *Event Viewer* built-in tool. Using the *Open Saved Log* from the *Action* menu, the *Event Viewer* is capable of opening ETL files directly (it will suggest converting to the newer EVTX format, though). The file will be located under the *Saved Logs* node (figure 20-13).

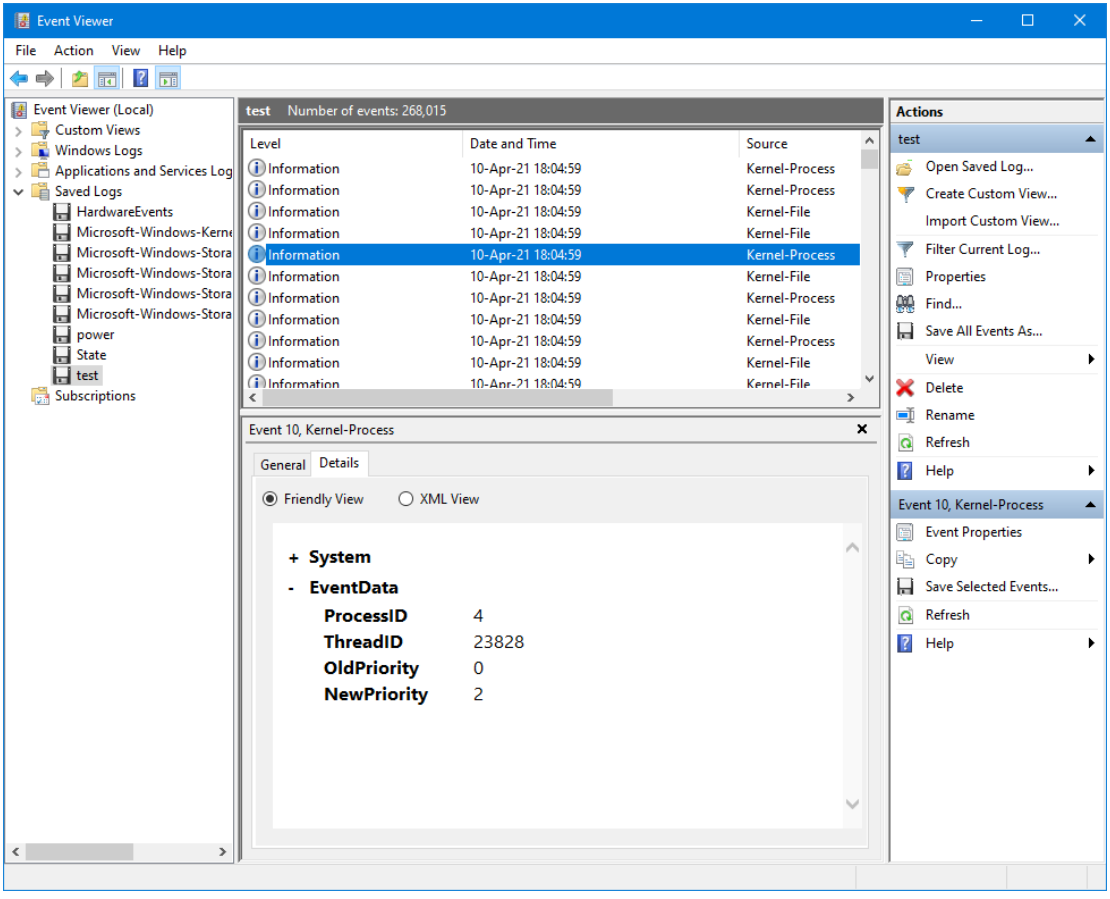


Figure 20-13: *Event Viewer* with an ETL file open

Finally, the *Windows Performance Analyzer* (WPA) tool, part of the *Windows Performance Toolkit* (can be installed as part of the Windows SDK), has advanced analysis capabilities for ETW logs, with graphs, filters, and more. Figure 21-14 shows the same ETL file loaded into WPA with some views open.

If you can't find WPA on your system, it's likely wasn't installed as part of installing the Windows SDK. If you installed the SDK, re-run the installer and use the wizard to modify the installed components to include WPT.

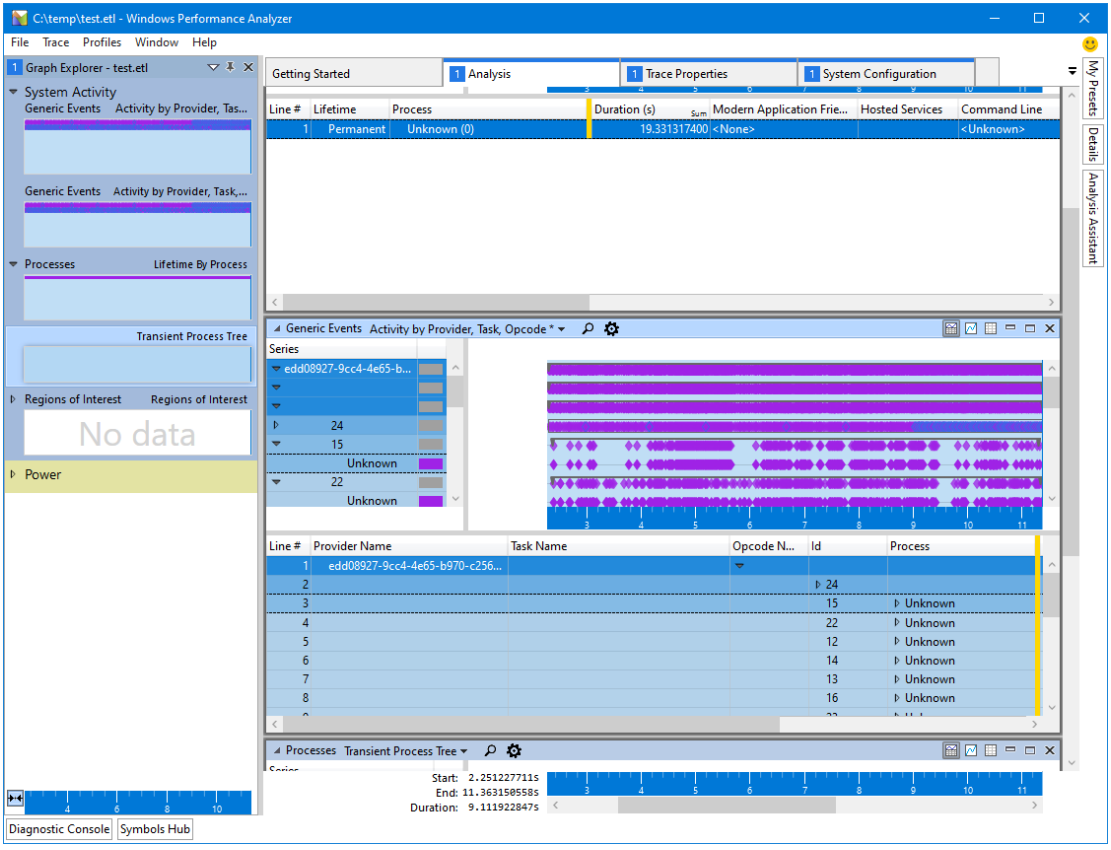


Figure 20-14: *Windows Performance Analyzer* with an ETL file open

WPA is beyond the scope of this book. There are some videos online that show the basics of using it.

Capturing ETW traces to a file can be done with existing tools. *logman* is one such tool. The Windows Performance Toolkit also provides the tools *xperf*, *wpr* and *wprui* for the same purpose. Each one has a different way of specifying providers and other configuration, but they all generate ETL files. *wprui* has a simple GUI for capturing traces using common providers and even has some knowledge of scenarios with a set of recommended providers to use. The focus of this chapter is using the APIs, rather than using existing tools, but they are useful for capturing traces nonetheless.

Processing Traces

Although there are tools to open ETL files, we can use APIs to look into a trace, whether it's in a file or as part of a real-time session. We can see all events recorded in the trace, and do any kind of analysis using code. This is generally more flexible than using a tool, no matter how sofisticated.

To gain access to a trace (file-based or real-time), a call to `OpenTrace` is required:

```
typedef struct _EVENT_TRACE_LOGFILE {
    LPTSTR      LogFileName;  // Logfile Name
    LPTSTR      LoggerName;   // LoggerName
    LONGLONG    CurrentTime;  // timestamp of last event
    ULONG       BuffersRead;  // buffers read to date
    union {
        // Mode of the logfile
        ULONG               LogFileMode;
        // Processing flags
        ULONG               ProcessTraceMode;
    };
    EVENT_TRACE   CurrentEvent;   // Current Event from this stream
    TRACE_LOGFILE_HEADER LogfileHeader;  // logfile header structure
    PEVENT_TRACE_BUFFER_CALLBACK    // callback before each buffer
                    BufferCallback; // is read

    // following variables are filled for BufferCallback
    ULONG                   BufferSize;
    ULONG                   Filled;
    ULONG                   EventsLost;

    // following needs to be propagated to each buffer
    union {
        // Callback with EVENT_TRACE
        PEVENT_CALLBACK         EventCallback;
        // Callback with EVENT_RECORD on Vista+
        PEVENT_RECORD_CALLBACK  EventRecordCallback;
    };

    ULONG       IsKernelTrace;    // TRUE for kernel logfile
    PVOID       Context;
} EVENT_TRACE_LOGFILE, *PEVENT_TRACE_LOGFILE;

TRACEHANDLE OpenTrace(_Inout_ PEVENT_TRACE_LOGFILE Logfile);
```

The `EVENT_TRACE_LOGFILE` structure has quite a few members. Fortunately, only few are required as

input. For a file-based trace, only three members are needed to be set on input. The rest of the members should be zeroed out.

The *ParseETW* project demonstrates parsing ETW events from a file. Its command line accepts a file path to parse and extract events from. The call to `OpenTrace` looks like this:

```
EVENT_TRACE_LOGFILE etl = { 0 };
etl.LogFileName = argv[1];      // filename
etl.ProcessTraceMode = PROCESS_TRACE_MODE_EVENT_RECORD;
etl.EventRecordCallback = OnEvent;

TRACEHANDLE hTrace = ::OpenTrace(&etl);
if(hTrace == INVALID_PROCESSTRACE_HANDLE) {
    printf("Failed to open trace\n");
    return 1;
}
```

Event processing is based on invoking a callback for each event in the trace. There are two possible callbacks: the old one (`EventCallback` member) and the newer (Vista+) specified in the `EventRecordCallback` member. The flag `PROCESS_TRACE_MODE_EVENT_RECORD` indicates usage of the latter, which is recommended. The callback itself is named `OnEvent` in the preceding code snippet. The callback can be named anything you like, but must have the following prototype:

```
void WINAPI OnEvent(PEVENT_RECORD record);
```

`EVENT_RECORD` is discussed shortly. A successful call to `OpenTrace` does not yet invoke the callback. At this point, the `EVENT_TRACE_LOGFILE` is populated with some general information from the trace source.

A successful call to `OpenTrace` returns a trace handle that is then passed to `ProcessTrace`. Specifically, the `LogfileHeader` member has some useful information that can be displayed like so:

```
auto& header = etl.LogfileHeader;
printf("OS Version: %d.%d Build: %d\n",
    header.VersionDetail.MajorVersion,
    header.VersionDetail.MinorVersion,
    header.ProviderVersion);
printf("Processors: %u\n", header.NumberOfProcessors);
printf("Start: %ws\nEnd:   %ws\n",
    (PCWSTR)CTime(*(FILETIME*)&header.StartTime).Format(L"%c"),
    (PCWSTR)CTime(*(FILETIME*)&header.EndTime).Format(L"%c"));
```

The code uses the `CTime` ATL class to format the start and end times of the trace. Here is the output for the previously captured *test.etl*:

```
OS Version: 10.0 Build: 19043
Processors: 16
Start: Sat Apr 10 18:04:40 2021
End:   Sat Apr 10 18:04:59 2021
```

You can see the trace was active for about 19 seconds. Check out the documentation for `EVENT_TRACE_LOGFILE` for other potentially useful data.

To start processing the events, call `ProcessTrace`:

```
ULONG ProcessTrace (
    _In_reads_(HandleCount) PTRACEHANDLE HandleArray,
    _In_ ULONG HandleCount,
    _In_opt_ LPFILETIME StartTime,
    _In_opt_ LPFILETIME EndTime);
```

`ProcessTrace` is able to process multiple traces provided via an array of trace handles. In the common case, there is just one. `StartTime` and `EndTime` are optional `FILETIME` values that can be used to limit the events processing to a certain range. Each value is in the "standard" form of being the number of 100 nsec units since January 1, 1601 UTC. Specifying `NULL` for both is common, as it delivers all events.

`ProcessTrace` takes control of the current thread, making calls to the registered callback from `OpenTrace` sequentially, until all events (possibly in a time range) are processed; only then the function returns. In some cases this inconvenient, so a typical approach is to create a thread and call `ProcessTrace` on that thread, so it does not block the current flow of execution.

The callback receives an `EVENT_RECORD` structure pointer, defined like so:

```
typedef struct _EVENT_RECORD {
    EVENT_HEADER       EventHeader;       // Event header
    ETW_BUFFER_CONTEXT BufferContext;     // Buffer context
    USHORT             ExtendedDataCount; // Number of extended
                                          // data items
    USHORT             UserDataLength;    // User data length
    PEVENT_HEADER_EXTENDED_DATA_ITEM      // Pointer to an array of
                       ExtendedData;      // extended data items
    PVOID              UserData;          // Pointer to user data
    PVOID              UserContext;       // Context from OpenTrace
} EVENT_RECORD, *PEVENT_RECORD;
```

The `EventHeader` member holds the basic information of the event, for which interpretation is needed. Here is its definition:

```
typedef struct _EVENT_HEADER {
    USHORT              Size;                   // Event Size
    USHORT              HeaderType;             // Header Type
    USHORT              Flags;                  // Flags
    USHORT              EventProperty;          // User given event prop\
erty
    ULONG               ThreadId;               // Thread Id
    ULONG               ProcessId;              // Process Id
    LARGE_INTEGER       TimeStamp;              // Event Timestamp
    GUID                ProviderId;             // Provider Id
    EVENT_DESCRIPTOR    EventDescriptor;        // Event Descriptor
    union {
        struct {
            ULONG       KernelTime;             // Kernel Mode CPU ticks
            ULONG       UserTime;               // User mode CPU ticks
        };
        ULONG64         ProcessorTime;          // Processor Clock
                                                // for private session e\
vents
    };
    GUID                ActivityId;             // Activity Id
} EVENT_HEADER, *PEVENT_HEADER;
```

The most interesting members of `EVENT_HEADER` are `ProcessId` (the process in which context the event was raised), `ThreadId` (the specific thread raising the event), `TimeStamp` (event raising time), `ProviderId` (the event's provider), and `EventDescriptor`, another structure giving the aspects of the event described in the previous sections, such as keyword and level (shown earlier in this chapter, repeated here for convenience):

```
typedef struct _EVENT_DESCRIPTOR {
    USHORT      Id;
    UCHAR       Version;
    UCHAR       Channel;
    UCHAR       Level;
    UCHAR       Opcode;
    USHORT      Task;
    ULONGLONG   Keyword;
} EVENT_DESCRIPTOR, *PEVENT_DESCRIPTOR;
```

Based on the above structures, we can display some common event information:

```
void DisplayGeneralEventInfo(PEVENT_RECORD rec) {
    WCHAR sguid[64];
    auto& header = rec->EventHeader;
    ::StringFromGUID2(header.ProviderId, sguid, _countof(sguid));

    printf("Provider: %ws Time: %ws PID: %u TID: %u\n",
        sguid,
        (PCWSTR)CTime(*(FILETIME*)&header.TimeStamp).Format(L"%c"),
        header.ProcessId, header.ThreadId);
}
```

Add code to dump the event descriptor.

Now we need to dig deeper into the event. The full information of the event (although not that easy to parse) is available dynamically with `TdhGetEventInformation`:

```
TDHSTATUS TdhGetEventInformation(
    _In_ PEVENT_RECORD Event,
    _In_ ULONG TdhContextCount,
    _In_opt_(TdhContextCount) PTDH_CONTEXT TdhContext,
    _Out_opt_(*BufferSize) PTRACE_EVENT_INFO Buffer,
    _Inout_ PULONG BufferSize);
```

Notice that unlike the "static" metadata information, this function requires an `EVENT_RECORD` as its first parameter, which is only available while processing events. `TdhContextCount` and `TdhContext` are useful for decoding events from non-registered providers, which is out of scope for this chapter. Specifying zero and `NULL` is the way to go for registered providers. The last two parameters mimic the usual pattern of calling the function twice: first with a zero size and a `NULL` pointer, and then with an allocated object that starts with a `TRACE_EVENT_INFO` structure:

```
typedef struct _TRACE_EVENT_INFO {
    GUID ProviderGuid;
    GUID EventGuid;
    EVENT_DESCRIPTOR EventDescriptor;
    DECODING_SOURCE DecodingSource;
    ULONG ProviderNameOffset;
    ULONG LevelNameOffset;
    ULONG ChannelNameOffset;
    ULONG KeywordsNameOffset;
    ULONG TaskNameOffset;
```

```
    ULONG OpcodeNameOffset;
    ULONG EventMessageOffset;
    ULONG ProviderMessageOffset;
    ULONG BinaryXMLOffset;
    ULONG BinaryXMLSize;
    union {
        ULONG EventNameOffset;
        ULONG ActivityIDNameOffset;
    };
    union {
        ULONG EventAttributesOffset;
        ULONG RelatedActivityIDNameOffset;
    };
    ULONG PropertyCount;
    _Field_range_(0, PropertyCount)
    ULONG TopLevelPropertyCount;
    union {
        TEMPLATE_FLAGS Flags;
        struct {
            ULONG Reserved : 4; // TEMPLATE_FLAGS values
            ULONG Tags : 28;
        };
    };
    _Field_size_(PropertyCount)
    EVENT_PROPERTY_INFO EventPropertyInfoArray[ANYSIZE_ARRAY];
} TRACE_EVENT_INFO;
typedef TRACE_EVENT_INFO *PTRACE_EVENT_INFO;
```

The structure is heavily documented in the header file (*tdh.h*) and is recommended as extra reading. Following the basic structure is all other information related to the event, such as various strings (`KeywordsNameOffset`, `OpcodeNameOffset`, etc.) and the event's properties themselves. Their number is not known in advance, which is why calling `TdhGetEventInformation` twice is necessary. The number of properties is stored in `TopLevelPropertyCount`, where the actual properties are in an array of `EVENT_PROPERTY_INFO` structures called `EventPropertyInfoArray` (the comments are from the header file):

```
typedef enum _PROPERTY_FLAGS {
   PropertyStruct           = 0x1,  // Type is struct
   PropertyParamLength      = 0x2,  // Length field is index of
                                    // param with length
   PropertyParamCount       = 0x4,  // Count field is index of
                                    // param with count
   PropertyWBEMXmlFragment  = 0x8,  // WBEM extension flag
                                    // for property
   PropertyParamFixedLength = 0x10, // Length of the parameter
                                    // is fixed
   PropertyParamFixedCount  = 0x20, // Count of the parameter
                                    // is fixed
   PropertyHasTags          = 0x40, // The Tags field has
                                    // been initialized
   PropertyHasCustomSchema  = 0x80, // Type is described with a
                                    // custom schema
} PROPERTY_FLAGS;

typedef struct _EVENT_PROPERTY_INFO {
    PROPERTY_FLAGS Flags;
    ULONG NameOffset;   // offset to property name
    union {
        struct _nonStructType {
            USHORT InType;
            USHORT OutType;
            ULONG MapNameOffset;
        } nonStructType;
        struct _structType {
            USHORT StructStartIndex;
            USHORT NumOfStructMembers;
            ULONG padding;
        } structType;
        struct _customSchemaType {
            USHORT InType;
            USHORT OutType;
            ULONG CustomSchemaOffset;
        } customSchemaType;
    };
    union {
        USHORT count;
        USHORT countPropertyIndex;
    };
    union {
```

```
        USHORT length;
        USHORT lengthPropertyIndex;
    };
    union {
        ULONG Reserved;
        struct {
            ULONG Tags : 28;
        };
    };
} EVENT_PROPERTY_INFO;
```

> _TRACE_EVENT_INFO has a TopLevelPropertyCount and PropertyCount. What's the difference? Some properties may contain sub-properties, so the total property count is in PropertyCount. TopLevelPropertyCount only accounts for the top level properties. For most events,

TopLevelPropertyCount is the same as PropertyCount.

You might feel somewhat overwhelmed at the amount of details present. The various options are necessary to get the flexibility and power available with ETW.

Let's start our processing callback by calling the DisplayGeneralEventInfo helper implemented earlier and then obtain the event information with TdhGetEventInformation:

```
void CALLBACK OnEvent(PEVENT_RECORD rec) {
    DisplayGeneralEventInfo(rec);

    ULONG size = 0;
    auto status = ::TdhGetEventInformation(rec, 0, nullptr,
        nullptr, &size);
    assert(status == ERROR_INSUFFICIENT_BUFFER);

    auto buffer = std::make_unique<BYTE[]>(size);
    if(!buffer) {
        printf("Out of memory!\n");
        ::ExitProcess(1);
    }

    auto info = reinterpret_cast<PTRACE_EVENT_INFO>(buffer.get());
    status = ::TdhGetEventInformation(rec, 0, nullptr,
        info, &size);
    if(status != ERROR_SUCCESS) {
```

```
        printf("Error processing event!\n");
        return;
    }

    DisplayEventInfo(rec, info);    // see later
}
```

We'll implement `DisplayEventInfo` to show more information about the event including its properties and values. We'll start by showing some of the strings corresponding to numerical values for keywords, level, etc. These are provided as offsets from the beginning of the `TRACE_EVENT_INFO` object. If the value of the offset is zero, then there is no string:

```
void DisplayEventInfo(PEVENT_RECORD rec, PTRACE_EVENT_INFO info) {
    if(info->KeywordsNameOffset)
        printf("Keywords: %ws ",
            (PCWSTR)((BYTE*)info + info->KeywordsNameOffset));
    if(info->OpcodeNameOffset)
        printf("Opcode: %ws ",
            (PCWSTR)((BYTE*)info + info->OpcodeNameOffset));
    if(info->LevelNameOffset)
        printf("Level: %ws ",
            (PCWSTR)((BYTE*)info + info->LevelNameOffset));
    if(info->TaskNameOffset)
        printf("Task: %ws ",
            (PCWSTR)((BYTE*)info + info->TaskNameOffset));
    if(info->EventMessageOffset)
        printf("\nMessage: %ws",
            (PCWSTR)((BYTE*)info + info->EventMessageOffset));
```

Next, we would like to display values for the properties of the event. This requires iterating over the property array, getting the property's name and value and optionally displaying it for human consumption. The property data itself is located at offset `UserData` of `EVENT_RECORD` and the total length of property data is available via `UserDataLength` in the same structure.

In this example, we are going to display the properties values for human consumption. In a more "backend" application, the property values would be analyzed with no regard to their human-readable display. For the former purpose, the ETW API provides the `TdhFormatProperty` to get a human-readable value of a property with relatively little work on the developer's part:

```
TDHSTATUS TdhFormatProperty(
    _In_ PTRACE_EVENT_INFO EventInfo,
    _In_opt_ PEVENT_MAP_INFO MapInfo,
    _In_ ULONG PointerSize,
    _In_ USHORT PropertyInType,
    _In_ USHORT PropertyOutType,
    _In_ USHORT PropertyLength,
    _In_ USHORT UserDataLength,
    _In_reads_bytes_(UserDataLength) PBYTE UserData,
    _Inout_ PULONG BufferSize,
    _Out_writes_bytes_opt_(*BufferSize) PWCHAR Buffer,
    _Out_ PUSHORT UserDataConsumed);
```

`EventInfo` is the pointer we already have to the event information. The optional `MapInfo` pointer allows mapping values of enumerations and bit fields to strings. If `NULL`, no such mapping is attempted. Getting the map information (if available) is accomplished with `TdhGetEventMapInformation`, which we'll see momentarilly.

`PointerSize` is the size of a pointer (4 or 8 bytes), which depends on the process issuing the event. Fortunately, it's easy to get from the `Flags` member of `EVENT_RECORD`. `PropertyInType` and `PropertyOutType` are the types available in `EVENT_PROPERTY_INFO.nonStructType.InType` and `OutType`. `PropertyLength` is the length of the value, also available as part of `EVENT_PROPERTY_INFO` (with some possible tweaks). `UserDataLength` is the event data size remaining with information, followed by the pointer itself to the beginning of the property data (`UserData`).

The string result is returned in `Buffer`, which has a size of `*BufferSize` (in bytes). The standard way would be to call `TdhFormatProperty` twice, similarly to other ETW APIs. Finally, `*UserDataConsumed` returns the number of bytes consumed for reading the event's value; it's expected to be the same as the property's length.

All this may seem quite complex. This is compounded by the fact that some properties are not "simple" - they may be arrays or of structure types, which requires extra coding to deal with these variants. In the following code snippets, we'll deal with simple properties only (non-structure and non-arrays). The compound ones are left as an exercise for the reader. Fortunately, compound types properties are rare.

First, we'll display the property count and get the pointer and size to the data, along with some other initialization:

```
printf("\nProperties: %u\n", info->TopLevelPropertyCount);
// properties data length and pointer
auto userlen = rec->UserDataLength;
auto data = (PBYTE)rec->UserData;
auto pointerSize =
    (rec->EventHeader.Flags & EVENT_HEADER_FLAG_32_BIT_HEADER)
        ? 4 : 8;
ULONG len;
WCHAR value[512];
```

We also retrieve the pointer size needed for `TdhFormatProperty` by examining the flags in the event header. Now we're ready start iterating over the properties, while displaying each property's name:

```
for(DWORD i = 0; i < info->TopLevelPropertyCount; i++) {
    auto& pi = info->EventPropertyInfoArray[i];
    auto propName = (PCWSTR)((BYTE*)info + pi.NameOffset);
    printf(" Name: %ws ", propName);
```

It's time to deal with simple properties only by skipping other types. The only caveat is making sure the pointer and data size advance correctly even if a property is skipped, as all data is contigous:

```
len = pi.length;
if((pi.Flags & (PropertyStruct | PropertyParamCount)) == 0) {
    // simple properties only
    //...
}
else {
    printf("(not a simple property)\n");
}
// advance to next property
userlen -= len;
data += len;
```

The following code is inside the above `if` block, dealing with a simple property only. The first (optional) order of business is getting map information, if available:

```
PEVENT_MAP_INFO mapInfo = nullptr;
std::unique_ptr<BYTE[]> mapBuffer;
PWSTR mapName = nullptr;
// retrieve map information (if any)
if(pi.nonStructType.MapNameOffset) {
    ULONG size = 0;
    mapName = (PWSTR)((BYTE*)info + pi.nonStructType.MapNameOffset);
    if(ERROR_INSUFFICIENT_BUFFER == ::TdhGetEventMapInformation(
        rec, mapName, mapInfo, &size)) {
        mapBuffer = std::make_unique<BYTE[]>(size);
        mapInfo = reinterpret_cast<PEVENT_MAP_INFO>(mapBuffer.get());
        if(ERROR_SUCCESS != ::TdhGetEventMapInformation(
            rec, mapName, mapInfo, &size))
            mapInfo = nullptr;
    }
}
```

If there is map information `pi.nonStructType.MapNameOffset` is not zero, `TdhGetEventMapInformation` is used to get that information, with the double call routine used before. Refer to the documentation for the function details, but we can just treat the map buffer as opaque information provided to `TdhFormatProperty`. Now we're ready to make the call:

```
    ULONG size = sizeof(value);
    USHORT consumed;
    // special case for IPv6 address
    if (pi.nonStructType.InType == TDH_INTYPE_BINARY &&
        pi.nonStructType.OutType == TDH_OUTTYPE_IPV6)
        len = sizeof(IN6_ADDR);

    auto error = ::TdhFormatProperty(info, mapInfo, pointerSize,
        pi.nonStructType.InType, pi.nonStructType.OutType,
        (USHORT)len, userlen, data, &size, value, &consumed);
    if(ERROR_SUCCESS == error) {
        printf("Value: %ws", value);
        len = consumed;
        if(mapName)
            printf(" (%ws)", (PCWSTR)mapName);
        printf("\n");
    }
    else if(mapInfo) {
        error = ::TdhFormatProperty(info, nullptr, pointerSize,
            pi.nonStructType.InType, pi.nonStructType.OutType,
            (USHORT)len, userlen, data, &size, value, &consumed);
```

```
        if(ERROR_SUCCESS == error)
            printf("Value: %ws\n", value);
    }
    if(ERROR_SUCCESS != error)
        printf("(failed to get value)\n");
}
```

Instead of calling `TdhFormatProperty` twice, the code assumes the value is not going to be longer than 512 characters (1024 bytes). All the arguments to `TdhFormatProperty` are within reach. If the first call fails and there is map information, then we make a second call without the map information, in case the map information is incomplete (for whatever reason), we'll at least get the numeric value of the property.

If you run the application on our *test.etl* file, you'll get tons of output with events and their properties. Here are a few examples:

```
Provider: {22FB2CD6-0E7B-422B-A0C7-2FAD1FD0E716}
    Time: Sat Apr 10 18:04:44 2021 PID: 39056 TID: 45212
Keywords: WINEVENT_KEYWORD_IMAGE Opcode: Info
    Level: Information  Task: ImageUnload
Message: Process %3 had an image unloaded with name %7.
Properties: 7
 Name: ImageBase Value: 0x7FFAE0B60000
 Name: ImageSize Value: 0xA000
 Name: ProcessID Value: 39056
 Name: ImageCheckSum Value: 55158
 Name: TimeDateStamp Value: 340988162
 Name: DefaultBase Value: 0x7FFAE0B60000
 Name: ImageName
    Value: \Device\HarddiskVolume3\Windows\System32\version.dll

Provider: {22FB2CD6-0E7B-422B-A0C7-2FAD1FD0E716}
    Time: Sat Apr 10 18:04:44 2021 PID: 3400 TID: 48272
Keywords: WINEVENT_KEYWORD_WORK_ON_BEHALF Opcode: Info
    Level: Information  Task: ThreadWorkOnBehalfUpdate
Properties: 2
 Name: OldWorkOnBehalfThreadID Value: 1184
 Name: NewWorkOnBehalfThreadID Value: 0

Provider: {22FB2CD6-0E7B-422B-A0C7-2FAD1FD0E716}
    Time: Sat Apr 10 18:04:45 2021 PID: 1888 TID: 44876
Keywords: WINEVENT_KEYWORD_PROCESS Opcode: Start
    Level: Information  Task: ProcessStart
Message: Process %1 started at time %3 by parent %4
    running in session %6 with name %11.
```

```
Properties: 15
 Name: ProcessID Value: 16296
 Name: ProcessSequenceNumber Value: 18928
 Name: CreateTime Value:
 Name: ParentProcessID Value: 1888
 Name: ParentProcessSequenceNumber Value: 14
 Name: SessionID Value: 1
 Name: Flags Value: PackageId  (ProcessFlags)
 Name: ProcessTokenElevationType Value: 3
 Name: ProcessTokenIsElevated Value: 0
 Name: MandatoryLabel Value: S-1-16-4096
 Name: ImageName Value:
    \Device\HarddiskVolume3\Windows\System32\backgroundTaskHost.exe
 Name: ImageChecksum Value: 0x13BB3
 Name: TimeDateStamp Value: 0x1D3A15E7
 Name: PackageFullName
    Value: Microsoft.Windows.ContentDeliveryManager_10.0.19041.
    423_neutral_neutral_cw5n1h2txyewy
 Name: PackageRelativeAppId Value: App
```

Extended Information

Recall from the section *Creating ETW Sessions* that it's possible to request adding extended information for each event, such as the user's SID, the session ID and a stack trace. If present, this information is available via the `ExtendedDataCount` and `ExtendedData` members of `EVENT_RECORD`.

`ExtendedData` is a pointer to an array of `EVENT_HEADER_EXTENDED_DATA_ITEM` structures, that form headers for each data item type:

```
typedef struct _EVENT_HEADER_EXTENDED_DATA_ITEM {
    USHORT      Reserved1;        // Reserved for internal use
    USHORT      ExtType;          // Extended info type
    struct {
        USHORT  Linkage    :  1;
        USHORT  Reserved2  : 15;
    };
    USHORT      DataSize;         // Size of extended info data
    ULONGLONG   DataPtr;          // Pointer to extended info data
} EVENT_HEADER_EXTENDED_DATA_ITEM, *PEVENT_HEADER_EXTENDED_DATA_ITEM;
```

`ExtType` indicates the type of item, such as `EVENT_HEADER_EXT_TYPE_TS_ID` and `EVENT_HEADER_EXT_TYPE_TS_ID`. The actual data is stored in `DataPtr` and its size is in `DataSize`. The documentation indicates which data structure is pointed to by `DataPtr`. For example, in the case of a SID, it points to the standard `SID` structure, but in the case of a session ID it points to `EVENT_EXTENDED_ITEM_TS_ID`, holding the session ID number.

Add the SID flag when capturing events with *RunETW* and add the required parsing code in *ParseETW*, displaying the SID for each event.

A more ambitious challenge is to add stack trace information and display them. An extension to that would be to correlate addresses with symbols from the Microsoft public symbol server. This is beyond the scope of this chapter, but you can look at the *DbgHelp* API for achieving this fit.

Real-Time Event Processing

Real-time processing of events is not much different than parsing events from a file. Before calling `ProcessTrace` for a real-time session, it must be opened similarly to a file-based session with `OpenTrace`, where the event callback is specified.

One thing is different, however. The processing is done while events continue to come in. It's single threaded processing, where ETW uses the session's buffers to hold on to information even if event volume is high. If processing the event is too slow, the buffers might fill completely, causing old events to be dropped, thus resulting in lost events. It's up to your callback to handle events quickly. One option is to increase the number of buffers and/or their size (part of `EVENT_TRACE_PROPERTIES`). Another option is to create your own queuing mechanism, where you push the event data to your queue, returning quickly, so that events are not lost. Then you can utilize one or more threads to process the events.

To show the differences from the purely file-based logging, another project is created, *RunETW2*. We'll highlight the changes from the original version, that would support file-based and/or real-time processing.

The `RunSession` function starts as it did in *RunETW*, by allocating the correct buffer size consisting of an `EVENT_TRACE_PROPERTIES` object plus the space required for the session name and the optional log file. Next, we need to call `StartTrace` as before, but there is a twist that is also applicable to a purely file-based logging. The call to `StartTrace` is in a potential loop:

```
auto props = reinterpret_cast<EVENT_TRACE_PROPERTIES*>(buffer.get());
DWORD status;
TRACEHANDLE hTrace = 0;

do {
    ::ZeroMemory(buffer.get(), size);

    props->Wnode.BufferSize = (ULONG)size;
    props->Wnode.Flags = WNODE_FLAG_TRACED_GUID;
    props->Wnode.ClientContext = 1;    // QueryPerformanceCounter
    props->Wnode.Guid = sessionGuid;
    props->LogFileMode = (filename ? EVENT_TRACE_FILE_MODE_SEQUENTIAL : \
0)
        | (realTime ? EVENT_TRACE_REAL_TIME_MODE : 0);
    props->MaximumFileSize = 100;    // 100 MB
    props->LoggerNameOffset = sizeof(*props);
```

```
    props->LogFileNameOffset =
        filename ? sizeof(*props) + sizeof(sessionName) : 0;

    // copy session name
    ::wcscpy_s((PWSTR)(props + 1), ::wcslen(sessionName) + 1,
        sessionName);

    // copy filename
    if(filename)
        ::wcscpy_s((PWSTR)(buffer.get()
            + sizeof(*props) + sizeof(sessionName)),
            ::wcslen(filename) + 1, filename);

    status = ::StartTrace(&hTrace, sessionName, props);
    if(status == ERROR_ALREADY_EXISTS) {
        status = ::ControlTrace(hTrace, sessionName, props,
            EVENT_TRACE_CONTROL_STOP);
        continue;
    }
    break;
} while(true);

if(ERROR_SUCCESS != status)
    return false;
```

The gist of the code is the same as it was in *RunETW*, but `StartTrace` can return `ERROR_ALREADY_EXISTS` (183), which is an error value. This indicates that the session with the specified name or GUID already exists. This might seem weird, and in some sense it is. The point is that an ETW session has a "life of its own", and does not depend on the lifetime of the process creating the session. This means that if some crash occurs in the process, and `StopTrace` isn't called, the session is still very much alive. (You can find it in the *Event Trace Session* node as seen in figure 20-8).

This means that the next time the application runs, it might try to start an already-running session. One easy way to handle that is to stop the session and start it from scratch with the required parameters. This is one type of action supported by the `ControlTrace` API:

```
ULONG ControlTrace (
    _In_ TRACEHANDLE TraceHandle,
    _In_opt_ LPCTSTR InstanceName,
    _Inout_ PEVENT_TRACE_PROPERTIES Properties,
    _In_ ULONG ControlCode);
```

`ControlTrace` provides the capabilities of the `StopTrace` API we used already, as well as `QueryTrace`, `UpdateTrace` and `FlushTrace`. Its `ControlCode` parameter indicates the type of operation to perform.

`EVENT_TRACE_CONTROL_STOP` is used in the above code to stop the session. A `StopTrace` call would get the job done just as well, but it's worthwhile getting to know `ControlTrace`.

If the error returned from `StartTrace` is not `ERROR_ALREADY_EXISTS`, then we break out of the loop and then return `false` in case there is any error.

Now we have to handle the real-time case by calling `OpenTrace` with proper flags indicating real-time processing:

```
TRACEHANDLE hParse = 0;
HANDLE hThread = nullptr;
if(realTime) {
    g_hStop = ::CreateEvent(nullptr, TRUE, FALSE, nullptr);

    EVENT_TRACE_LOGFILE etl{};
    etl.LoggerName = (PWSTR)sessionName;
    etl.ProcessTraceMode = PROCESS_TRACE_MODE_EVENT_RECORD
        | PROCESS_TRACE_MODE_REAL_TIME;
    etl.EventRecordCallback = OnEvent;
    hParse = ::OpenTrace(&etl);
    if(hParse == INVALID_PROCESSTRACE_HANDLE) {
        printf("Failed to open a read-time session\n");
    }
```

The event object (`g_hStop`) is going to be used to stop the session when the user presses *Ctrl+C* if a real-time session is running, because the console will be very busy displaying events and no *ENTER* key input would be easily detected. `OpenTrace` is called with no log file name but with the `PROCESS_TRACE_MODE_REAL_TIME` flag, indicating real-time delivery. The `OnEvent` callback is the same one used in the *ParseETW* sample.

Once the trace is open, we need to call `ProcessTrace` so that event delivery can commence. Since we know that `ProcessTrace` "hijacks" the thread it's called from, we'll create a dedicated thread to call `ProcessTrace`. Although it's not absolutely necessary in a console based application, it's a good example of how to handle this scenario in more realistic applications.

```
    else {
        hThread = ::CreateThread(nullptr, 0, [](auto param) -> DWORD {
            FILETIME now;
            ::GetSystemTimeAsFileTime(&now);
            ::ProcessTrace(static_cast<TRACEHANDLE*>(param),
            1, &now, nullptr);
            return 0;
            }, &hParse, 0, nullptr);
    }
}
```

The thread function is provided with a C++ lambda function, that calls `ProcessTrace` from the current time. This is optional, indicating past events we already missed (if any) are of no interest to us. When `ProcessTrace` returns (someone called `CloseTrace`), the thread function returns, terminating the thread.

The next step is to enable the providers by calling `EnableTraceEx` or one of its variants in a loop. This is the same code from *RunETW*, so is not shown here.

For a real-time session, we'll register a control handler and set the `g_hStop` event if *Ctrl+C* is hit. Then, we just wait until the event is signaled, at which point we close the trace and wait for the thread to make its exit:

```
if(realTime) {
    ::SetConsoleCtrlHandler([](auto code) {
        if(code == CTRL_C_EVENT) {
            ::SetEvent(g_hStop);
            return TRUE;
        }
        return FALSE;
        }, TRUE);
    ::WaitForSingleObject(g_hStop, INFINITE);
    ::CloseTrace(hParse);
    ::WaitForSingleObject(hThread, INFINITE);
    ::CloseHandle(g_hStop);
    ::CloseHandle(hThread);
}
else {
    // not a real-time session
    printf("Session running... press ENTER to stop\n");

    char dummy[4];
    gets_s(dummy);
}

::StopTrace(hTrace, sessionName, props);
```

`SetConsoleCtrlHandler` can be used to get notifications for special key presses, such as *Ctrl+C*. Its callback is provided here with a lambda function (again). When *Ctrl+C* is hit, it just sets the event, causing the main thread to be released from its wait, thereby closing the trace (`CloseTrace`) and closing the open handles that are no longer needed.

You can run the sample with the `-r` switch to indicate a real-time session, and see events displayed on the console.

The Kernel Provider

The ETW system recognizes a few special providers, treating them differently. The most useful one is known as the *Kernel Provider*. The name is somewhat misleading, as many ETW registered providers raise their events from the kernel. Its name as enumerated is "Windows Kernel Trace" with the GUID {9E814AAD-3204-11D2-9A82-006008A86939} also available with the variable `SystemTraceControlGuid` defined in *evntrace.h*. Its event keywords give some hints at its capabilities:

```
C:\>logman query providers "Windows Kernel Trace"

Provider                       GUID
-------------------------------------------------------------------
Windows Kernel Trace           {9E814AAD-3204-11D2-9A82-006008A86939}

Value               Keyword              Description
-------------------------------------------------------------------
0x0000000000000001  process        Process creations/deletions
0x0000000000000002  thread         Thread creations/deletions
0x0000000000000004  img            Image load
0x0000000000000008  proccntr       Process counters
0x0000000000000010  cswitch        Context switches
0x0000000000000020  dpc            Deferred procedure calls
0x0000000000000040  isr            Interrupts
0x0000000000000080  syscall        System calls
0x0000000000000100  disk           Disk IO
0x0000000000000200  file           File details
0x0000000000000400  diskinit       Disk IO entry
0x0000000000000800  dispatcher     Dispatcher operations
0x0000000000001000  pf             Page faults
0x0000000000002000  hf             Hard page faults
0x0000000000004000  virtalloc      Virtual memory allocations
0x0000000000010000  net            Network TCP/IP
0x0000000000020000  registry       Registry details
0x0000000000100000  alpc           ALPC
0x0000000000200000  splitio        Split IO
0x0000000000800000  driver         Driver delays
0x0000000001000000  profile        Sample based profiling
0x0000000002000000  fileiocompletion   File IO completion
0x0000000004000000  fileio             File IO
```

This provider's schema is not available using `TdhEnumerateManifestProviderEvents`, as it's defined with the older MOF format. The metadata is available using WMI metadata APIs (beyond the scope of this chapter), and can be viewed with *ETWExplorer*. The kernel provider's events are documented to some extent online at https://docs.microsoft.com/en-us/windows/win32/etw/msnt-systemtrace.

Prior to Windows 8, only a single session could be created with the kernel provider, and it has to be the only provider. The session name must be "NT Kernel Logger", also defined with the `KERNEL_LOGGER_NAME` macro. Manipulating the kernel provider requires admin privileges.

Starting with Windows 8, the single session limitation was lifted, so that up to 8 sessions can use the kernel provider (this rule applies to any provider), and the session name can be anything.

Several tools use the kernel provider. For example, the *Process Monitor*, *Process Explorer*, and *TcpView* tools from *Sysinternals* use the kernel provider to get network-related events. My own *ProcMonXv2* tool is based around the kernel provider's events as well. The *MD5Calculator* samples from chapter 7 (and others) use the kernel provider to get notification for DLL load events.

The *KernelETW* sample application is a variant of *RunETW2* that uses the kernel provider to log events of selected keywords. The various keywords are selected by setting the `EnableFlags` member of `EVENT_TRACE_PROPERTIES` instead of calling one of the `EnableTrace*` APIs. Here is the first part of the modified `RunSession` function, accepting the flags that comprise the event categories to enable:

```
bool RunSession(DWORD flags, PCWSTR filename, bool realTime) {
    // session name must be KERNEL_LOGGER_NAME for Win7 or earlier
    // assume Windows 8+
    const WCHAR sessionName[] = L"Chapter20KernelSession";

    auto size = sizeof(EVENT_TRACE_PROPERTIES)
        + (filename ? ((::wcslen(filename) + 1) * sizeof(WCHAR)) : 0)
        + sizeof(sessionName);

    auto buffer = std::make_unique<BYTE[]>(size);
    if(!buffer)
        return false;

    auto props = reinterpret_cast<EVENT_TRACE_PROPERTIES*>(
        buffer.get());
    DWORD status;
    TRACEHANDLE hTrace = 0;

    do {
        ::ZeroMemory(buffer.get(), size);

        props->Wnode.BufferSize = (ULONG)size;
        props->Wnode.Flags = WNODE_FLAG_TRACED_GUID;
        props->Wnode.ClientContext = 1;
        // no need to set kernel provider GUID specifically
        //props->Wnode.Guid = SystemTraceControlGuid;
```

```
        props->EnableFlags = flags;
        props->LogFileMode =
            (filename ? EVENT_TRACE_FILE_MODE_SEQUENTIAL : 0)
            | (realTime ? EVENT_TRACE_REAL_TIME_MODE : 0)
            // indicates kernel provider
            | EVENT_TRACE_SYSTEM_LOGGER_MODE;

        props->MaximumFileSize = 100;
        props->LoggerNameOffset = sizeof(*props);
        props->LogFileNameOffset =
            filename ? sizeof(*props) + sizeof(sessionName) : 0;

        // copy session name
        ::wcscpy_s((PWSTR)(props + 1), ::wcslen(sessionName) + 1,
            sessionName);

        // copy filename
        if(filename)
            ::wcscpy_s((PWSTR)(buffer.get()
                + sizeof(*props) + sizeof(sessionName)),
                ::wcslen(filename) + 1, filename);

        status = ::StartTrace(&hTrace, sessionName, props);
        if(status == ERROR_ALREADY_EXISTS) {
            status = ::ControlTrace(hTrace, sessionName, props,
                EVENT_TRACE_CONTROL_STOP);
            continue;
        }
        break;
    } while(true);
```

The `EVENT_TRACE_SYSTEM_LOGGER_MODE` flag is required to indicate using the kernel provider. The provider's GUID is automatically set by the call to `StartTrace`. In fact, if you specify it, `StartTrace` fails. On the other hand, if the classic session name is used (mandatory for Windows 7 or earlier), specifying the GUID is allowed.

The keywords flag is built by the `GetKeywords` function like so:

```
DWORD GetKeywords(const std::vector<PCWSTR>& names) {
    // always use process notifications
    DWORD flags = EVENT_TRACE_FLAG_PROCESS;

    const struct {
        PCWSTR name;
        DWORD flag;
    } keywords[] = {
      { L"process", EVENT_TRACE_FLAG_PROCESS },
      { L"image", EVENT_TRACE_FLAG_IMAGE_LOAD },
      { L"thread", EVENT_TRACE_FLAG_THREAD },
      { L"registry", EVENT_TRACE_FLAG_REGISTRY },
      { L"file",
        EVENT_TRACE_FLAG_FILE_IO | EVENT_TRACE_FLAG_FILE_IO_INIT },
      { L"disk", EVENT_TRACE_FLAG_DISK_IO |
        EVENT_TRACE_FLAG_DISK_IO_INIT | EVENT_TRACE_FLAG_DISK_FILE_IO },
      { L"network", EVENT_TRACE_FLAG_NETWORK_TCPIP },
      { L"memory", EVENT_TRACE_FLAG_VAMAP },
      { L"alloc", EVENT_TRACE_FLAG_VIRTUAL_ALLOC},
    };

    using namespace std;
    for(auto name : names) {
        auto it = find_if(begin(keywords), end(keywords),
            [&](auto& k) { return ::_wcsicmp(k.name, name) == 0; });
        if(it == end(keywords)) {
            printf("Unknown flag: %ws\n", name);
            continue;
        }
        flags |= it->flag;
    }

    return flags;
}
```

Notice the usage of the global `std::begin` and `std::end` functions. This allows treating regular C arrays as containers, since a C array does not (and cannot) have methods.

The function uses a subset of keywords supported by the kernel provider. The following example would show events in real-time involving processes and threads:

```
c:\>KernelETW -r process thread
```

The `main` function calls `GetKeywords` to obtain the final value to pass to `RunSession`. See the source code for the full details.

More ETW

The next subsections provide more details on ETW that may be of interest.

ETW Filters

Originally, ETW providers worked on a system wide basis - there was no way to filter by process, for example. Starting with Windows 7, providers can support several types of filters. This is more efficient than receving all events and filtering on the consumer side. For manifest based providers, these filters can be enumerated by calling `TdhEnumerateProviderFilters`:

```
TDHSTATUS TdhEnumerateProviderFilters(
    _In_ LPGUID Guid,
    _In_ ULONG TdhContextCount,         // unused
    _In_opt_ PTDH_CONTEXT TdhContext,   // unused
    _Out_ ULONG* FilterCount,
    _Out_opt_ PPROVIDER_FILTER_INFO* Buffer,
    _Inout_ ULONG* BufferSize);
```

The paramters should be almost self-evident, as well as the usage of the function. Given a provider's GUID, the number of filters is returned in `*FilterCount` as an array of `PROVIDER_FILTER_INFO` structures defined like so:

```
typedef struct _PROVIDER_FILTER_INFO {
    UCHAR Id;
    UCHAR Version;
    ULONG MessageOffset;
    ULONG Reserved;
    ULONG PropertyCount;
    _Field_size_(PropertyCount)
    EVENT_PROPERTY_INFO EventPropertyInfoArray[ANYSIZE_ARRAY];
} PROVIDER_FILTER_INFO, *PPROVIDER_FILTER_INFO;
```

Although, technically a good idea, no registered provider supports filtering at the time of this writing. I have added a function to the *ETWMeta* project to display filter information. Given these potential filters, calling `EnableTraceEx` or `EnableTraceEx2` allow specifying one filter to the provider (if supported).

Session Information

Once a session is open with `StartTrace`, it's possible to query or set various aspects of the session with `TraceQueryInformation` (Windows 8+) and `TraceSetInformation` (Windows 7+):

```
ULONG TraceQueryInformation (
    _In_ TRACEHANDLE SessionHandle,
    _In_ TRACE_INFO_CLASS InformationClass,
    _Out_writes_bytes_(InformationLength) PVOID TraceInformation,
    _In_ ULONG InformationLength,
    _Out_opt_ PULONG ReturnLength);
ULONG TraceSetInformation (
    _In_ TRACEHANDLE SessionHandle,
    _In_ TRACE_INFO_CLASS InformationClass,
    _In_reads_bytes_(InformationLength) PVOID TraceInformation,
    _In_ ULONG InformationLength);
```

These functions are generic, where the buffers supplied depend on the "information class" to query or set, defined by the `TRACE_(QUERY_)INFO_CLASS` enumeration:

```
typedef enum _TRACE_QUERY_INFO_CLASS {
    TraceGuidQueryList = 0,
    TraceGuidQueryInfo = 1,
    TraceGuidQueryProcess = 2,
    TraceStackTracingInfo = 3,
    TraceSystemTraceEnableFlagsInfo = 4,
    TraceSampledProfileIntervalInfo = 5,
    TraceProfileSourceConfigInfo = 6,
    TraceProfileSourceListInfo = 7,
    TracePmcEventListInfo = 8,
    TracePmcCounterListInfo = 9,
    TraceSetDisallowList = 10,
    TraceVersionInfo = 11,
    TraceGroupQueryList = 12,
    TraceGroupQueryInfo = 13,
    TraceDisallowListQuery = 14,
    TraceInfoReserved15 = 15,
    TracePeriodicCaptureStateListInfo = 16,
    TracePeriodicCaptureStateInfo = 17,
    TraceProviderBinaryTracking = 18,
    TraceMaxLoggersQuery = 19,
    TraceLbrConfigurationInfo = 20,
    TraceLbrEventListInfo = 21,
    TraceMaxPmcCounterQuery = 22,
    MaxTraceSetInfoClass
} TRACE_QUERY_INFO_CLASS, TRACE_INFO_CLASS;
```

Another function that uses the above enumeration is `EnumerateTraceGuidsEx`, which does not require a session handle, and thus is about the system as a whole rather than a specific session:

```
ULONG EnumerateTraceGuidsEx (
    _In_ TRACE_QUERY_INFO_CLASS TraceQueryInfoClass,
    _In_reads_bytes_opt_(InBufferSize) PVOID InBuffer,
    _In_ ULONG InBufferSize,
    _Out_writes_bytes_opt_(OutBufferSize) PVOID OutBuffer,
    _In_ ULONG OutBufferSize,
    _Out_ PULONG ReturnLength);
```

The enumeration is heaviliy commented in the *evntrace.h* header where it's defined, providing the expected format of the buffer passed to the above functions (not all functions support all the enumeration's values).
The following example enumerates all the provider GUIDs known to the system (not just those registered):

```
ULONG len;
// first call with lebgth zero returns required length
::EnumerateTraceGuidsEx(TraceGuidQueryList, nullptr, 0,
    nullptr, 0, &len);
auto cguids = len / sizeof(GUID);
auto guids = std::make_unique<GUID[]>(cguids);
status = ::EnumerateTraceGuidsEx(TraceGuidQueryList, nullptr, 0,
    guids.get(), len, &len);
// do something with guids...
```

Continue the above code snippet by displaying the details for each provider by calling `EnumerateTraceGuidsEx` for each provider with the `TraceGuidQueryInfo` trace information class.

Active Sessions

You may be wondering how to get the list of active ETW sessions, as seen in figure 8. This is where `QueryAllTraces` comes in:

```
ULONG QueryAllTraces (
    _Out_ PEVENT_TRACE_PROPERTIES *PropertyArray,
    _In_  ULONG PropertyArrayCount,
    _Out_ PULONG LoggerCount);
```

Using this function is not intuitive, however. The following snippet enumerates all running sessions:

```
ULONG nsessions;
PEVENT_TRACE_PROPERTIES psessions[256]; // absolute maximum
// the following uses a large enough buffer per session info
DWORD sessionPropSize = 4 << 10; // 4KB
auto buffer = std::make_unique<BYTE[]>(
    _countof(psessions) * sessionPropSize);
::ZeroMemory(buffer.get(), _countof(psessions) * sessionPropSize);

for(int i = 0; i < _countof(psessions); i++) {
    psessions[i] = (PEVENT_TRACE_PROPERTIES)(buffer.get()
        + sessionPropSize * i);
    auto session = psessions[i];
    session->Wnode.BufferSize = sessionPropSize;
    session->LoggerNameOffset = sizeof(EVENT_TRACE_PROPERTIES);
    session->LogFileNameOffset =
        session->LoggerNameOffset + 1024 * sizeof(WCHAR);
}

status = ::QueryAllTraces(psessions, _countof(psessions),
    &nsessions);
for(int i = 0; i < nsessions; i++) {
    printf("Session name: %ws\n",
        (PCWSTR)((BYTE*)psessions[i]
            + psessions[i]->LoggerNameOffset));
}
```

Trace Logging

Up until now, we've seen how to consume ETW events. It's time to examine how an application can report ETW events of its own.

ETW supports four type of providers, summarized in table 20-4.

Table 20-4: ETW Provider types

Provider Type	Provider APIs	Remarks
Classic (MOF)	`RegisterTraceGuids`, `TraceEvent`	Can be enabled on a single trace at a time. Use WMI APIs to query metadata.
WPP (Windows Preprocessor Provider)	`RegisterTraceGuids`, `TraceEvent`	Can be enabled on a single trace at a time. Used mostly by kernel drivers.

Table 20-4: ETW Provider types

Provider Type	Provider APIs	Remarks
Manifest-based (XML)	`EventRegister`, `EventWrite`	Can be enabled on 8 traces at a time. Require global registration to consume metadata.
Trace Logging	`TraceLoggingRegister`, `TraceLoggingWrite`	Can be enabled on 8 traces at a time. Bundle metadata with event data.

The providers we used in previous sections were manifest-based or classic. Applications that wish to expose ETW events should not use the classic or WPP providers because of their obvious limitations. Starting with Windows Vista, manifest-based providers were created to lift the "one provider per session" limit and have a more flexible schema model.

Using manifest-based provider for publishing events is possible, but requires manifest registration, so that consumers have access to the manifest for event decoding. Fortunately, Windows 10 added another provider type, known as *Trace Logging*, that allow publishing ETW events in an easier fashion that does not require global registration, as the metadata is bundled with the events themselves, making them self-describing. Although Windows 10 introduced these new APIs, they can be used on any Windows version from Vista, as they use the underlying ETW infrastructure just like manifest-based publishing.

Publishing Events with Trace Logging

Working with Trace Logging is surprisingly simple. To demonstrate publishing such events (and consuming them), we'll create a slightly simpler variant of the *MD5Calculator* project from chapter 7. That application used ETW as a consumer to get notifications for loaded DLLs (using the kernel provider), and calculating the MD5 hash of each DLL. As a bonus, a cache of calculated hashes was used as well, so that calculations don't have to repeat.

The *MD5Calc* project is a simplified version in the following ways:

- It's a console application, rather than a GUI one.
- It does not use ETW to get notified when a DLL is loaded. Instead, the application enumerates processes every second, and for every new process, enumerates the loaded DLLs in the new processes and performs a similar calculation.
- No locking is used, as everything is single threaded.

The `HashCache` is a simplified version compared to the one in chapter 7, that has no locking at all:

```
// HashCache.h

using Hash = std::vector<uint8_t>;

#include <unordered_map>

class HashCache {
public:
    HashCache();

    bool Add(PCWSTR path, const Hash& hash);
    const Hash Get(PCWSTR path) const;
    bool Remove(PCWSTR path);
    void Clear();

private:
    std::unordered_map<std::wstring, Hash> _cache;
};

// HashCache.cpp
#include "HashCache.h"

HashCache::HashCache() {
    _cache.reserve(512);
}

bool HashCache::Add(PCWSTR path, const Hash& hash) {
    auto it = _cache.find(path);
    if (it == _cache.end()) {
        _cache.insert({ path, hash });
        return true;
    }
    return false;
}

const Hash HashCache::Get(PCWSTR path) const {
    auto it = _cache.find(path);
    return it == _cache.end() ? Hash() : it->second;
}

bool HashCache::Remove(PCWSTR path) {
    auto it = _cache.find(path);
    if (it != _cache.end()) {
```

```
        _cache.erase(it);
        return true;
    }
    return false;
}

void HashCache::Clear() {
    _cache.clear();
}
```

In order to work with Trace Logging, the following headers are needed (located in *pch.h*):

```
#include <Windows.h>
#include <tdh.h>
#include <TraceLoggingProvider.h>
```

The last include is the new one. It provides functions and macros for trace logging. The application must generate a unique GUID and name, using the `TRACELOGGING_DEFINE_PROVIDER` macro. The following, taken from *MD5Calc.cpp*, defines our provider:

```
// {C44DF504-CA51-4CA6-BEA6-F20B6AECAED9}
TRACELOGGING_DEFINE_PROVIDER(g_Provider, "MD5CalcProvider",
    (0xc44df504, 0xca51, 0x4ca6, 0xbe, 0xa6, 0xf2,
    0xb, 0x6a, 0xec, 0xae, 0xd9));
```

The GUID was generated with the *GUIDGen* tool, accessible from the *Tools* Visual Studio menu (see chapter 21 for more on GUIDs and this tool). The second radio button was selected before clicking *Copy* to copy the text to the clipboard (figure 20-15).

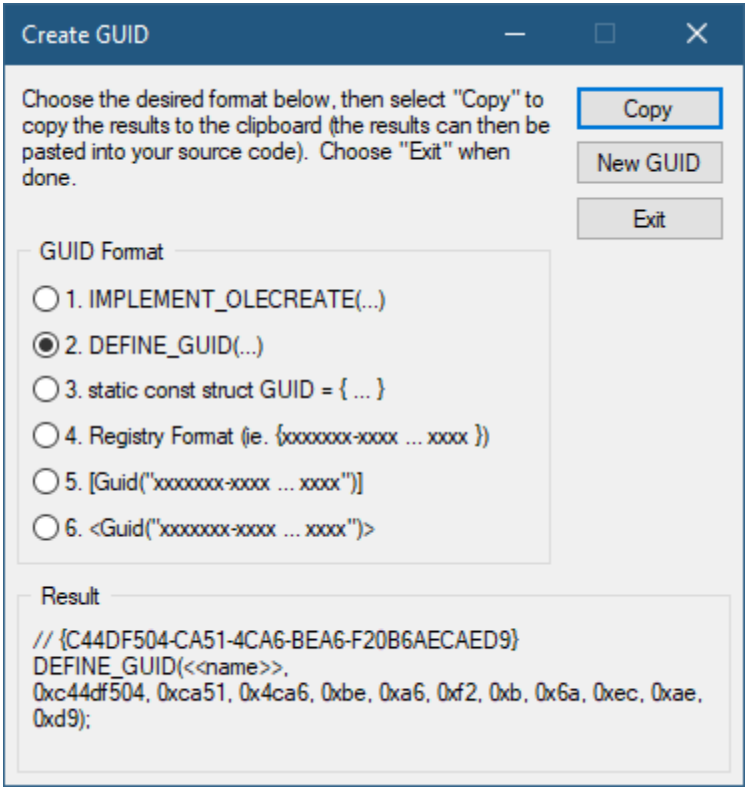


Figure 20-15: *GUIDGen*

The `main` function starts by registering the provider with ETW:

```
int main() {
    ::TraceLoggingRegister(g_Provider);
```

To publish an event, use the `TraceLoggingWrite` macro, that accepts any number of parameters, specified with other macros, each representing some property of the event. Here is one example, published right after initial registration:

```
TraceLoggingWrite(g_Provider, "Init",
    TraceLoggingLevel(TRACE_LEVEL_INFORMATION),
    TraceLoggingValue(L"Process started", "Text"),
    TraceLoggingValue(::GetCurrentProcessId(), "PID"));
```

The event name is "Init", its level is Information, and it has two properties: One is a Unicode string with the value "Process started" (property name is "Text"), and the second is a `DWORD` with the process ID as the value, and the property name is "PID". `TraceLoggingValue` attempts to infer the property type given the value (the first argument). Other macros exist that are more specific, such as `TraceLoggingInt32`,

where the property type is specified explicitly. It doesn't matter which you use, it's a matter of convenience and/or taste.

Besides the cache, we need to manage the processes on the system and keep track of new ones since the previous enumeration. This is the task for the `ProcessManager` class, defined like so:

```
#include <unordered_set>

class ProcessManager {
public:
    ProcessManager();

    bool Refresh();
    const std::vector<DWORD>& GetNewProcesses();

    std::vector<std::wstring> EnumModules(DWORD pid) const;

private:
    std::unordered_set<DWORD> _pids;
    std::vector<DWORD> _newPids;
};
```

A call to `Refresh` would enumerate the processeses, comparing with the existing list (`_pids`), reporting back the new PIDs when `GetNewProcesses` is called. The implementation of the class is shown below, but will not be discussed further, as there is nothing new we have not seen before.

```
ProcessManager::ProcessManager() {
    _pids.reserve(500);
    _newPids.reserve(500);
    Refresh();
}

bool ProcessManager::Refresh() {
    auto old = _pids;

    _newPids.clear();

    wil::unique_handle hSnapshot(
        ::CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0));
    if(!hSnapshot)
        return false;

    PROCESSENTRY32 pe;
    pe.dwSize = sizeof(pe);
```

```
    if(!::Process32First(hSnapshot.get(), &pe))
        return false;

    while(::Process32Next(hSnapshot.get(), &pe)) {
        auto pid = pe.th32ProcessID;
        if(old.find(pid) == old.end()) {
            _newPids.push_back(pid);
            _pids.insert(pid);
        }
        else {
            old.erase(pid);
        }
    }

    for(auto pid : old)
        _pids.erase(pid);

    return true;
}

const std::vector<DWORD>& ProcessManager::GetNewProcesses() {
    return _newPids;
}

std::vector<std::wstring>
ProcessManager::EnumModules(DWORD pid) const {
    std::vector<std::wstring> modules;

    wil::unique_handle hSnapshot(::CreateToolhelp32Snapshot(
        TH32CS_SNAPMODULE | TH32CS_SNAPMODULE32, pid));
    if(!hSnapshot)
        return modules;

    MODULEENTRY32 me;
    me.dwSize = sizeof(me);

    if(!::Module32First(hSnapshot.get(), &me))
        return modules;

    modules.reserve(50);
    do {
        modules.push_back(me.szExePath);
```

```
    } while(::Module32Next(hSnapshot.get(), &me));

    return modules;
}
```

The MD5 calculation itself is performed by a static function in the `MD5Calculator` class, essentially identical to the one from chapter 7. The only difference, it publishes ETW events as part of the calculation. Here is the code, with `TraceLoggingWrite` sprinkled throughout the function:

```
TRACELOGGING_DECLARE_PROVIDER(g_Provider);

std::vector<uint8_t> MD5Calculator::Calculate(PCWSTR path) {
    TraceLoggingWrite(g_Provider, "CalculatingMD5",
        TraceLoggingLevel(TRACE_LEVEL_INFORMATION),
        TraceLoggingValue(path, "Path"));

    std::vector<uint8_t> md5;

    wil::unique_hfile hFile(::CreateFile(path,
        GENERIC_READ, FILE_SHARE_READ, nullptr,
        OPEN_EXISTING, FILE_FLAG_SEQUENTIAL_SCAN, nullptr));
    if (!hFile)
        return md5;

    wil::unique_handle hMemMap(::CreateFileMapping(
        hFile.get(), nullptr, PAGE_READONLY, 0, 0, nullptr));
    if (!hMemMap)
        return md5;

    wil::unique_hcryptprov hProvider;
    if (!::CryptAcquireContext(hProvider.addressof(), nullptr, nullptr,
        PROV_RSA_FULL, CRYPT_VERIFYCONTEXT))
        return md5;

    wil::unique_hcrypthash hHash;
    if(!::CryptCreateHash(hProvider.get(), CALG_MD5, 0, 0,
        hHash.addressof()))
        return md5;

    wil::unique_mapview_ptr<BYTE> buffer((BYTE*)::MapViewOfFile(
        hMemMap.get(), FILE_MAP_READ, 0, 0, 0));
    if (!buffer)
        return md5;
```

```
    auto size = ::GetFileSize(hFile.get(), nullptr);
    TraceLoggingWrite(g_Provider, "CalculatingMD5Begin",
        TraceLoggingLevel(TRACE_LEVEL_VERBOSE),
        TraceLoggingOpcode(EVENT_TRACE_TYPE_START),
        TraceLoggingValue(size, "FileSize"));

    if (!::CryptHashData(hHash.get(), buffer.get(), size, 0))
        return md5;

    DWORD hashSize;
    DWORD len = sizeof(DWORD);
    if (!::CryptGetHashParam(hHash.get(), HP_HASHSIZE,
        (BYTE*)&hashSize, &len, 0))
        return md5;

    md5.resize(len = hashSize);
    ::CryptGetHashParam(hHash.get(), HP_HASHVAL, md5.data(),
        &len, 0);

    TraceLoggingWrite(g_Provider, "CalculatingMD5End",
        TraceLoggingLevel(TRACE_LEVEL_VERBOSE),
        TraceLoggingOpcode(EVENT_TRACE_TYPE_END),
        TraceLoggingBinary(md5.data(), len,
            "MD5", "MD5 hash result"));

    return md5;
}
```

The *Windows Implementation Library* (WIL) was added to the project using Nuget.

The `TRACELOGGING_DECLARE_PROVIDER` macro is used as an "extern" declaration so that the ETW provider is available for use. Notice the flexibility and ease of use when calling `TraceLoggingWrite`. The last call even publishes the MD5 hash itself as a binary blob.

Now that we have the main pieces, we can complete the `main` function, to work until *Ctrl+C* is hit:

```
HashCache cache;
ProcessManager pm;

::SetConsoleCtrlHandler([](auto code) {
    if(code == CTRL_C_EVENT) {
        HANDLE hEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE,
            GenerateEventName().c_str());
        _ASSERT(hEvent);
        ::SetEvent(hEvent);
        ::CloseHandle(hEvent);
        return TRUE;
    }
    return FALSE;
    }, TRUE);

HANDLE hEvent = CreateEvent(nullptr, FALSE, FALSE,
    GenerateEventName().c_str());
while(::WaitForSingleObject(hEvent, 1000) == WAIT_TIMEOUT) {
    pm.Refresh();
    printf("Received %u new processes\n",
        (uint32_t)pm.GetNewProcesses().size());
    for(auto pid : pm.GetNewProcesses()) {
        TraceLoggingWrite(g_Provider, "Processing",
            TraceLoggingLevel(TRACE_LEVEL_INFORMATION),
            TraceLoggingUInt32(pid, "PID"));

        for(auto& m : pm.EnumModules(pid)) {
            auto hash = cache.Get(m.c_str());
            bool cached = true;
            if(hash.empty()) {
                hash = MD5Calculator::Calculate(m.c_str());
                cached = false;
                cache.Add(m.c_str(), hash);
            }
            printf("MD5: %s Cached: %s Path: %ws\n",
                HashToString(hash), cached ? "YES" : "NO ", m.c_str());
        }
    }
    ::Sleep(1000);
}
::CloseHandle(hEvent);

TraceLoggingWrite(g_Provider, "Term",
```

```
    TraceLoggingLevel(TRACE_LEVEL_INFORMATION));

::TraceLoggingUnregister(g_Provider);
```

Two helper functions are used in the above snippet. The first generates a unique-enough event name based on the process ID:

```
std::wstring GenerateEventName() {
    return L"MD5CalcStopEvent"
        + std::to_wstring(::GetCurrentProcessId());
}
```

The second just convertes a hash (`vector` of bytes) to a string:

```
const char* HashToString(const std::vector<uint8_t>& v) {
    static std::string result;
    result.clear();
    char value[3];
    for(auto n : v) {
        sprintf_s(value, "%02X", n);
        result += value;
    }
    return result.c_str();
}
```

Consumning Trace Logging events is no different than what have done in the previous sections. The metadata is available automatically when using the `Tdh` APIs. One thing I had to do is enhance the `DisplayEventInfo` function from the *RunETW2* sample (which should also be copied for the other applications the decode events) is the case where the size of a property is not directly available as part of `EVENT_PROPERTY_INFO`, but rather available indirectly. Here is the addition required:

```
if(pi.Flags & PropertyParamLength) {
    // property length is stored elsewhere
    auto index = pi.lengthPropertyIndex;
    PROPERTY_DATA_DESCRIPTOR desc;
    desc.ArrayIndex = ULONG_MAX;
    desc.PropertyName = (ULONGLONG)propName;
    desc.Reserved = 0;
    // get the "real" length (stored in len)
    ::TdhGetPropertySize(rec, 0, nullptr, 1, &desc, &len);
}
```

Another enhancement we need to make for *RunETW2* is the ability to accept GUIDs as provider identifiers. Here is the first part of `GetProviders` that adds that capability:

```
std::vector<GUID> GetProviders(std::vector<PCWSTR> names) {
    std::vector<GUID> providers;
    providers.reserve(names.size());

    auto count = names.size();
    for(size_t i = 0; i < count; i++) {
        auto name = names[i];
        if(name[0] == L'{') {    // GUID rather than name
            GUID guid;
            if(S_OK == ::CLSIDFromString(name, &guid)) {
                providers.push_back(guid);
                names.erase(names.begin() + i);
                i--;
                count--;
            }
        }
    }

    if(names.empty())
        return providers;
```

Now we are ready to test the consumer while the *MD5Calc* application is running. Here is an example run (notice the provider's GUID provided in the command line):

```
C:\Dev\Win10SysProg\>runetw2 -r {C44DF504-CA51-4CA6-BEA6-F20B6AECAED9}
Provider: {C44DF504-CA51-4CA6-BEA6-F20B6AECAED9}
    Time: Sat May 15 23:01:04 2021 PID: 27464 TID: 17388
Task: Processing
Properties: 1
 Name: PID Value: 55700

Provider: {C44DF504-CA51-4CA6-BEA6-F20B6AECAED9}
    Time: Sat May 15 23:01:04 2021 PID: 27464 TID: 17388
Task: CalculatingMD5
Properties: 1
 Name: Path Value: C:\Dev\Win10SysProg\Chapter20\x64\Debug\RunETW2.exe

Provider: {C44DF504-CA51-4CA6-BEA6-F20B6AECAED9}
    Time: Sat May 15 23:01:04 2021 PID: 27464 TID: 17388
Task: CalculatingMD5Begin
Properties: 1
 Name: FileSize Value: 197120
```

```
Provider: {C44DF504-CA51-4CA6-BEA6-F20B6AECAED9}
    Time: Sat May 15 23:01:04 2021 PID: 27464 TID: 17388
Task: CalculatingMD5End
Properties: 2
 Name: MD5.Length Value: 16
 Name: MD5 Value: 0x86B471C98E57C9AF0F26B622D82EF591

Provider: {C44DF504-CA51-4CA6-BEA6-F20B6AECAED9}
    Time: Sat May 15 23:01:04 2021 PID: 27464 TID: 17388
Task: CalculatingMD5
Properties: 1
 Name: Path Value: C:\WINDOWS\SYSTEM32\ntdll.dll

Provider: {C44DF504-CA51-4CA6-BEA6-F20B6AECAED9}
    Time: Sat May 15 23:01:04 2021 PID: 27464 TID: 17388
Task: CalculatingMD5Begin
Properties: 1
 Name: FileSize Value: 2024728

Provider: {C44DF504-CA51-4CA6-BEA6-F20B6AECAED9}
    Time: Sat May 15 23:01:04 2021 PID: 27464 TID: 17388
Task: CalculatingMD5End
Properties: 2
 Name: MD5.Length Value: 16
 Name: MD5 Value: 0x02779F8274BED93C74F119B883AC9888
...
```

Debuggers

If you're writing any code, you are bound to use a debugger. Debuggers attach to processes, with the power of controlling threads in the debugged process, as well as having complete access to the process address space. Still, debuggers are programs just like any other, running just like other processes. A process becomes a debugger when it attaches to a process using certain APIs. In this section, we'll look at the basic debugger architecture, building a simple debugger along the way.

A process becomes a debugger in one of two ways:

- The to-be-debugged process is launched with a normal `CreateProcess` call, specifying the `DEBUG_PROCESS` or `DEBUG_ONLY_THIS_PROCESS` flag in the process flags parameter (the sixth). The calling process then becomes the newly created process debugger.
- A process attaches to an existing process by calling `DebugActiveProcess`. If successful, the calling process becomes the debugger of the target process.

> A process can be debugged by a single process only. In other words, a process can have at most one debugger attached to it. On the other hand, a single debugger may debug multiple processes at the same time.

Debugging processes that are part of other sessions require the *SeDebugPrivilege*, normally granted to administrators.

Calling `CreateProcess` with the `DEBUG_PROCESS` flag estanblishes the caller as the debugger of the new process and any child process created by that process. If debugging only the new process is desired, specify `DEBUG_ONLY_THIS_PROCESS` instead.

Attaching as a debugger to an existing process (assuming it's not already being debugged), is accomplished by calling `DebugActiveProcess`:

```
BOOL DebugActiveProcess(_In_ DWORD dwProcessId);
```

In both cases, the calling process becomes a debugger. This means that the process will start receiving debugger-related notifications when calling `WaitForDebugEvent` or `WaitForDebugEventEx` (Windows 10+):

```
BOOL WaitForDebugEvent(
    _Out_ LPDEBUG_EVENT lpDebugEvent,
    _In_ DWORD dwMilliseconds);
BOOL WaitForDebugEventEx(
    _Out_ LPDEBUG_EVENT lpDebugEvent,
    _In_ DWORD dwMilliseconds);
```

The extended function opts for receiving debug output events in Unicode directly (when the debugee calls `OutpoutDebugStringW` as described earlier in this chapter), rather than paying the cost of having the string converted to Unicode.

The function waits until the next debugging-related event is available, or the time out elapses. `lpDebugEvent` is the event data received once the function returns successfully. If the wait times out, the return value is `FALSE`.

The `DEBUG_EVENT` structure represents a single event using a union of structures, one of which is valid at a time. Here is its declaration:

```
typedef struct _DEBUG_EVENT {
    DWORD dwDebugEventCode;
    DWORD dwProcessId;
    DWORD dwThreadId;
    union {
        EXCEPTION_DEBUG_INFO Exception;
        CREATE_THREAD_DEBUG_INFO CreateThread;
        CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
        EXIT_THREAD_DEBUG_INFO ExitThread;
        EXIT_PROCESS_DEBUG_INFO ExitProcess;
        LOAD_DLL_DEBUG_INFO LoadDll;
        UNLOAD_DLL_DEBUG_INFO UnloadDll;
        OUTPUT_DEBUG_STRING_INFO DebugString;
        RIP_INFO RipInfo;
    } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;
```

`dwDebugEventCode` is the type of event reported. `dwProcessId` and `dwThreadId` are the process and thread IDs of the reporting process and thread, respectively. Remember that a debugger process may debug multiple processes at the same time. Table 20-5 describes the possible event types.

Table 20-5: Debugger events

Event Type	Data Member in union	Description
`CREATE_PROCESS_DEBUG_EVENT` (3)	`CreateProcessInfo`	Sent when the debugger attaches to a target process
`EXIT_PROCESS_DEBUG_EVENT` (5)	`ExitThread`	Sent when a target process exits
`CREATE_THREAD_DEBUG_EVENT` (2)	`CreateThread`	Sent when a thread is created in a target process
`EXIT_THREAD_DEBUG_EVENT` (4)	`ExitThread`	Sent when a thread exits in a target process
`EXCEPTION_DEBUG_EVENT` (1)	`Exception`	Sent when an exception occurs in a target process
`LOAD_DLL_DEBUG_EVENT` (6)	`LoadDll`	Sent when a DLL is loaded into a target process
`UNLOAD_DLL_DEBUG_EVENT` (7)	`UnloadDll`	Sent when a DLL is unloaded from a target process
`OUTPUT_DEBUG_STRING_EVENT` (8)	`DebugString`	Sent when a thread calls `OutputDebugString`
`RIP_EVENT` (9)	`RipInfo`	Sent when a target process is terminated outside the control of the debugger

Given the above definitions, a debugger creates a loop that consists of `WaitForDebugEvent(Ex)` call,

after which the event is dealt in some way. When the event is received, all threads in target processes are suspended, which means the debugger must release the targets after it's done with the current event. This requires calling `ContinueDebugEvent`:

```
BOOL ContinueDebugEvent(
    _In_ DWORD dwProcessId,
    _In_ DWORD dwThreadId,
    _In_ DWORD dwContinueStatus);
```

The first two parameters are the process and thread IDs, available from `WaitForDebugEvent(Ex)`. The last parameter (`dwContinueStatus`) indicates how the target thread should proceed. Table 20-6 shows the valid values with their meaning.

Table 20-6: Values for `ContinueDebugEvent`

Value	Description	Description
`DBG_CONTINUE`	If the event is an exception, continue as though it was handled. For any other event, continue normal execution.	
`DBG_EXCEPTION_NOT_HANDLED`	Continue normally if it's not an exception. If it's a first chance exception, give the thread a chance to handle it. If it's a second chance exception (meaning it wasn't handled), terminate the process.	
`DBG_TERMINATE_PROCESS`	Terminates the process.	
`DBG_TERMINATE_THREAD`	Terminates the thread.	
`DBG_REPLY_LATER`	(Windows 10+) Resumes all threads but causes the thread to replay the event after all other threads continue normally.	

In the case of an exit process or thread events, `ContinueDebugEvent` closes the handles handed to the caller.

A Simple Debugger

To demonstrate the `WaitForDebugEvent` and `ContinueDebugEvent` APIs, we'll build a simple debugger, that gets interesting events from the target. The complete code is in the *SimpleDebug* project.

SimpleDebug is a console based application that attached to an existing process or lanuches and executable and attaches to the new new process. The `main` function accepts either a process ID or an executable and arguments:

```
int wmain(int argc, const wchar_t* argv[]) {
    if (argc < 2) {
        printf("Usage: SimpleDebug <pid | executable [args...]>\n");
        return 1;
    }
```

If the first argument is a number, we would treat it as a process ID. Otherwise, let's assume it's an executable. In the process ID case, attach to the process:

```
DWORD pid = _wtoi(argv[1]);
if (pid != 0) {
    if (!::DebugActiveProcess(pid))
        return Error("Failed to attach to process");
    printf("Attached to process %u\n", pid);
}
```

If an executbale should be launched, then a `CreateProcess` call is required:

```
else {
    PROCESS_INFORMATION pi;
    STARTUPINFO si = { sizeof(si) };

    // build command line
    std::wstring path;
    for (int i = 1; i < argc; i++) {
        path += argv[i];
        path += L" ";
    }

    if (!::CreateProcess(nullptr, (PWSTR)path.data(),
        nullptr, nullptr, FALSE, DEBUG_PROCESS,
        nullptr, nullptr, &si, &pi))
        return Error("Failed to create and/or attach to process");
    printf("Process %u created\n", pi.dwProcessId);
    ::CloseHandle(pi.hProcess);
    ::CloseHandle(pi.hThread);
}
```

The code should be fairly familiar from chapter 3. The important part is specifying `DEBUG_PROCESS` (or `DEBUG_ONLY_THIS_PROCESS`), so that the calling process is registered as a debugger.

Now comes the debugger's loop:

```
DEBUG_EVENT evt;
while (::WaitForDebugEventEx(&evt, INFINITE)) {
    auto status = HandleEvent(evt);
    if (evt.dwDebugEventCode == EXIT_PROCESS_DEBUG_EVENT)
        break;
    ::ContinueDebugEvent(evt.dwProcessId, evt.dwThreadId, status);
}
```

If the event is a process exit event, the debugger loop is exited. This is technically incorrect, since we specified `DEBUG_PROCESS`, meaning multiple processes may be debugged at a time.

Fix the code so that if multiple processes are debugged, only the last exiting process would cause the debugger loop to exit.

The `HandleEvent` function is our function to handle events reported by `WaitForDebugEventEx`. It can be implemented as a list of cases like so:

```
DWORD HandleEvent(const DEBUG_EVENT& evt) {
    printf("Event PID: %u TID: %u %s (%u)\n",
        evt.dwProcessId, evt.dwThreadId,
        EventCodeToString(evt.dwDebugEventCode), evt.dwDebugEventCode);

    switch (evt.dwDebugEventCode) {
        case CREATE_PROCESS_DEBUG_EVENT:
            DisplayCreateProcessInfo(evt.u.CreateProcessInfo);
            hProcess = evt.u.CreateProcessInfo.hProcess;
            break;

        case CREATE_THREAD_DEBUG_EVENT:
            DisplayCreateThreadInfo(evt.u.CreateThread);
            break;

        case EXIT_PROCESS_DEBUG_EVENT:
            DisplayExitProcessInfo(evt.u.ExitProcess);
            break;

        case EXIT_THREAD_DEBUG_EVENT:
            DisplayExitThreadInfo(evt.u.ExitThread);
            break;

        case LOAD_DLL_DEBUG_EVENT:
            DisplayLoadDllInfo(evt.u.LoadDll);
```

```
            break;

        case UNLOAD_DLL_DEBUG_EVENT:
            DisplayUnloadDllInfo(evt.u.UnloadDll);
            break;

        case OUTPUT_DEBUG_STRING_EVENT:
            DisplayOutputDebugString(evt.u.DebugString);
            break;

        case EXCEPTION_DEBUG_EVENT:
            DisplayExceptionInfo(evt.u.Exception);
            return DBG_EXCEPTION_NOT_HANDLED;

        case RIP_EVENT:
            DisplayRipInfo(evt.u.RipInfo);
            break;
    }
    return DBG_CONTINUE;
}
```

Let's look at a few examples for certain events. You can find the rest in the project's source code. First, the process create event - always sent upon attach to the process (regardless of the method):

```
void DisplayCreateProcessInfo(
    const CREATE_PROCESS_DEBUG_INFO& info) {
    printf("\tImage Base: 0x%p\n", info.lpBaseOfImage);
    printf("\tStart address: 0x%p\n", info.lpStartAddress);
    printf("\tTEB: 0x%p\n", info.lpThreadLocalBase);
    BYTE buffer[1024];
    if (ReadImagePath(hProcess, info.lpImageName,
        buffer, sizeof(buffer)) > 0) {
        if (info.fUnicode)
            printf("\tName: %ws\n", (PCWSTR)buffer);
        else
            printf("\tName: %s\n", (PCSTR)buffer);
    }
}
```

The `CREATE_PROCESS_DEBUG_INFO` structure contains information the debugger can use:

```
typedef struct _CREATE_PROCESS_DEBUG_INFO {
    HANDLE hFile;
    HANDLE hProcess;
    HANDLE hThread;
    LPVOID lpBaseOfImage;
    DWORD dwDebugInfoFileOffset;
    DWORD nDebugInfoSize;
    LPVOID lpThreadLocalBase;
    LPTHREAD_START_ROUTINE lpStartAddress;
    LPVOID lpImageName;
    WORD fUnicode;
} CREATE_PROCESS_DEBUG_INFO, *LPCREATE_PROCESS_DEBUG_INFO;
```

You'll notice three handles provided to the debugger (image file handle, process and thread handles). The process and thread handles in particular have full access masks, so that the debugger can do anything with the debugee. These handles should be copied, so that the debugger can use them later (they are not closed automatically).

`lpImageName` is the image name of the target process in the target's process address space. To read that data, the debugger must use the `ReadProcessMemory` function - this is one example where an all-powerful handle is useful. The `ReadImagePath` helper function called by `DisplayCreateProcessInfo` does just that:

```
DWORD ReadImagePath(HANDLE hProcess, void* address,
    void* buffer, DWORD size) {
    SIZE_T read = 0;
    if (address) {
        void* p = nullptr;
        ::ReadProcessMemory(hProcess, address, &p, sizeof(p), nullptr);
        if (p) {
            ::ReadProcessMemory(hProcess, p, buffer, size, &read);
        }
    }
    return (DWORD)read;
}
```

You'll find many cases where the `lpImageName` field is `NULL`, so make sure the code checks for that.

Notice that two read operations take place: the first reads the address where the image path is stored. The second read operation reads the path itself.

Here is another example, handling the DLL load event:

```
void DisplayLoadDllInfo(const LOAD_DLL_DEBUG_INFO& info) {
    printf("\tBase address: 0x%p\n", info.lpBaseOfDll);
    printf("\tImage Name: 0x%p\n", info.lpImageName);
    BYTE buffer[1 << 10];
    if (ReadImagePath(hProcess, info.lpImageName,
        buffer, sizeof(buffer)) > 0) {
        if (info.fUnicode)
            printf("\tName: %ws\n", (PCWSTR)buffer);
        else
            printf("\tName: %s\n", (PCSTR)buffer);
    }
}
```

A typical debugger keeps track of processes, threads, and DLLs. Since it has full access to the process, it can read any memory location, query any information, and change things, such as insert breakpoints, write over memory, etc.

Running *SimpleDebug* with *notepad* shows results in the following (truncated) output:

```
Process 44956 created
Event PID: 44956 TID: 61092 Create Process (3)
        Image Base: 0x00007FF67FDA0000
        Start address: 0x00007FF67FDC3DB0
        TEB: 0x0000002898222000
Event PID: 44956 TID: 61092 Load DLL (6)
        Base address: 0x00007FF8B75F0000
        Image Name: 0x0000000000000000
Event PID: 44956 TID: 61092 Load DLL (6)
        Base address: 0x00007FF8B6410000
        Image Name: 0x0000002898222028
        Name: C:\WINDOWS\System32\KERNEL32.DLL
Event PID: 44956 TID: 61092 Load DLL (6)
        Base address: 0x00007FF8B5100000
        Image Name: 0x0000002898222028
        Name: C:\WINDOWS\System32\KERNELBASE.dll
...
Event PID: 44956 TID: 36412 Create Thread (2)
        TEB: 0x0000002898226000
        Start address: 0x00007FF8B7642AD0
Event PID: 44956 TID: 61092 Load DLL (6)
        Base address: 0x00007FF8B64E0000
        Image Name: 0x0000002898222028
        Name: C:\WINDOWS\System32\USER32.dll
Event PID: 44956 TID: 61808 Create Thread (2)
```

```
        TEB: 0x0000002898228000
        Start address: 0x00007FF8B7642AD0
Event PID: 44956 TID: 61092 Load DLL (6)
        Base address: 0x00007FF8B6AF0000
        Image Name: 0x0000002898222028
        Name: C:\WINDOWS\System32\combase.dll
...
Event PID: 44956 TID: 61092 Load DLL (6)
        Base address: 0x00007FF896B70000
        Image Name: 0x0000002898222028
        Name: C:\Windows\System32\MrmCoreR.dll
Event PID: 44956 TID: 61092 Output Debug String (8)
        onecoreuap\base\mrt\runtime\src\cresourcemanagerinternal.cpp(726\
)
        \MrmCoreR.dll!00007FF896B8D81B: (caller: 00007FF896B8CEAA) Retur\
nHr(1)
        tid(eea4) 80070002 The system cannot find the file specified.

Event PID: 44956 TID: 61092 Output Debug String (8)
        onecoreuap\base\mrt\runtime\src\cresourcemanagerinternal.cpp(577\
)
        \MrmCoreR.dll!00007FF896B8CFF4: (caller: 00007FF896B99301) Retur\
nHr(2)
        tid(eea4) 80070002 The system cannot find the file specified.

Event PID: 44956 TID: 61092 Output Debug String (8)
        onecoreuap\base\mrt\runtime\src\cresourcemanagerinternal.cpp(317\
)
        \MrmCoreR.dll!00007FF896B993C0: (caller: 00007FF896B9E516) Retur\
nHr(3)
        tid(eea4) 80070002 The system cannot find the file specified.
...
Event PID: 44956 TID: 61092 Load DLL (6)
        Base address: 0x00007FF8B5850000
        Image Name: 0x0000002898222028
        Name: C:\WINDOWS\System32\SHELL32.dll
...
Event PID: 44956 TID: 61808 Exit Thread (4)
        Exit code: 0x0
Event PID: 44956 TID: 48088 Exit Thread (4)
        Exit code: 0x0
...
```

More Debugging APIs

The `IsDebuggerPresent` returns `TRUE` if the current process is being debugged. A more generic function, `CheckRemoteDebuggerPresent` performs a similar query on any process given a handle with an access mask of `PROCESS_QUERY_INFORMATION`:

```
BOOL CheckRemoteDebuggerPresent(
    _In_ HANDLE hProcess,
    _Out_ PBOOL pbDebuggerPresent);
```

`IsDebuggerPresent` looks at a flag inside the *Process Environment Block* (PEB), which is a user space data structure. A malicious process might turn this bit off if it has sufficient access to the process, so that `IsDebuggerPresent` might return `FALSE` even if the process is being debugged. `CheckRemoteDebuggerPresent` is safer, since it asks the kernel if there is a debug port object associated with the process in question.

Once a debugger is attached to a process, it stays that way unless it detaches by calling `DebugActiveProcessStop`:

```
BOOL DebugActiveProcessStop(_In_ DWORD dwProcessId);
```

By default, the target process is terminated, unless the debugger calls `DebugSetProcessKillOnExit` and specifies otherwise:

```
BOOL DebugSetProcessKillOnExit(_In_ BOOL KillOnExit);
```

A process can set a breakpoint by inserting a call to `DebugBreak()`. This will cause a breakpoint exception notification to be sent to the connected debugger. If no debugger is connected, the process terminates.

An external break into a process is possible with `DebugBreakProcess`. This is typically used by a debugger to forcefully generate a breakpoint in a target process. It does so by injecting a thread into the target process that then calls `DebugBreak`.

```
BOOL DebugBreakProcess(_In_ HANDLE Process);
```

A debugger has full power over its targets, including the ability to read and change threads' context with `SetThreadContext` and `GetThreadContext`:

```
BOOL SetThreadContext(
    _In_ HANDLE hThread,
    _In_ CONST CONTEXT* lpContext);
BOOL GetThreadContext(
    _In_ HANDLE hThread,
    _Inout_ LPCONTEXT lpContext);
```

These advanced functions allow reading and changing the platform-dependent `CONTEXT` structure that contains CPU registers used by a thread. A typical usage would be suspending a thread, reading its context, making any required changes (such as pointing the thread to a different function), and then resuming the thread. See these functions documentation for more details.

Similar functions exist for manipulating WOW64 threads (when a 64-bit debugger process debugs a 32-bit process).

A debugger can also write or change code in the target process. if it soes so, calling `FlushInstructionCache` may be necessary, so that the CPU's cache is no longer used, forcing the CPU to read the updated code:

```
BOOL FlushInstructionCache(
    _In_ HANDLE hProcess,
    _In_reads_bytes_opt_(dwSize) LPCVOID lpBaseAddress,
    _In_ SIZE_T dwSize);
```

Writing a Real Debugger

The simple debugger from the previous section can hardly be called a *debugger*. It does not allow setting breakpoints, viewing memory, viewing source code (if available), and many other activities we associate with a debugger. Writing a full-fledged debugger is no mean feat, and is a huge undertaking.

The basics of any debugger are the ones shown - the debugger loop exists in any debugger, from simple to complex.

A typical debugger allows setting breakpoints. How would that work? A breakpoint is one type of exception. The debugger inserts a break instruction (such as `int 3` in x86/x64 assembly), overwriting some byte in the target's memory (the debugger saves that overwritten byte). Once the breakpoint hits, the debugger restores the previous byte, and handles the breakpoint.

Another big thing is symbols. How does a debugger correlate addresses with symbols and even source code? The *DebugHelp* API can be used to perform the required operations, which are out of scope for this chapter.

So what if you still want to write a debugger? Microsoft provides a debugger engine (implemented in *DbgEng.Dll*), that powers the fairly well-known *WinDbg* debugger (and other debuggers that are part of the *Debugging Tools for Windows* package). The debugger engine handles most of the tough parts of

a debugger, like setting breakpoints, working with symbols and source code, and more. The debugger engine API is documented, and can form the basis of a debugger (as it does *WinDbg*). You can find more information at https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-engine-and-extension-apis.

Even without writing a full featured debugger, a process can act as a debugger to analyze the way a target is behaving, taking advantage of the debugging events and the power it has over the target process(es).

Summary

In this chapter, we examined several mechanisms related to debugging and diagnostics. From a simple text output, to performance counters, to Event Tracing for Windows - there are quite a few ways to see what is hapenning in a process of interest or more holistically in a system.

Chapter 21: The Component Object Model

The *Component Object Model* (COM) technology made its official debut around 1992/93, initially called *OLE 2.0* and later renamed to *COM*, to better underline the fact that its use cases go much beyond an enhanced implementation for *Object Linking and Embedding* (OLE), most often used within the Microsoft Office suite of applications.

COM became the de-facto standard for object communication based on a binary protocol, which means COM servers and clients could be written in any programming language that adheres to the COM specification. COM is one of the most influential technology in the Microsoft technology stack, but it is also one of the most misunderstood.

In this chapter, we'll take a look at the fundamentals of COM, hopefully demystifying it, and see how to use it from a client and server perspective. At the end of the chapter, you should have the knowledge required to work with existing COM components and create your own when needed.

> This chapter is by no means exhaustive, as complete books have been written about COM.

In this chapter:

What is COM?

Whenever a new technology is created, it's because there are certain problems it aims to fix. COM is no different; what was COM trying to fix? Here are a few scenarios encountered while developing applications that are not easy to handle.

A developer wants to write a library, packaged as a DLL, to provide some functionality. What language should the library be written in? If it's written in C, it can be (relatively) easily consumed by a C or C++ client for sure, but what about other languages/platforms? C is considered the lowest common denominator, and facilities to consume it are available in most languages/platforms, such as .NET, Java, Python, Rust, and many others. But what if the developer writes the library in C++ that exposes a set of C++ classes? It's easy enough to consume in C++, but what about other languages/platforms? It's virtually impossible because C++ works on the source level rather than the binary level.

Even with C++, life is not ideal. The way to expose a C++ class in a DLL is to provide a LIB file that would be linked by the client. That forces the DLL to be loaded when the client process launches, rather than when it's needed. And if the DLL cannot be located, the process terminates, rather than having the opportunity to handle the failure gracefully. As we saw in chapter 15, it's practically impossible to dynamically link to a C++ DLL with calls to `LoadLibrary` and `GetProcAddress`.

it's possible to circumvent the dynamic loading problem by using Delay-Load DLLs, as described in chapter 15.

Here is another example: a developer wants to implement some C++ classes that are hosted in their own process. In this case, the client and server are different processes. How would the client gain access to the server's library functionality? Clearly, some form of inter-process communication (IPC) is required here, but how is that to be implemented so that it's easy for the client to consume the library, and easy for the server to expose that functionality?

Another issue is related to component evolution. Suppose that a C++ class is exposed by some library implemented in a DLL. At some point, a developer wants to extend the class with new functionality or even fix a bug or enhance a feature. Standard C++ rules dictate that if you don't change the public methods of a class, then you're good to go, as clients of your class do not need to make any source code changes. Here is a simple C++ class to serve as an example:

```
// RPNCalculator.h

class RPNCalculator {
public:
    RPNCalculator();
    void Push(double value);
    double Pop();
    bool Add();
    bool Subtract();
```

```
private:
    std::stack<double> _stack;
};
```

This class implements a *Reverse Polish Notation* (RPN) calculator, where values are pushed onto a stack, and calculations are performed by calling the relevant method (e.g. `Add`). The method pops two values off the stack, performs the calculation, and then pushes the result onto the stack. The stack is implemented in this example using the C++ standard library `std::stack<>` adapter class.

Suppose this class is properly implemented and packaged in a DLL. A client working with the calculator can do so with code like the following:

```
#include "RPNCalculator.h"

void SimpleCalc() {
    RPNCalculator calc;
    calc.Push(10);
    calc.Push(20);
    calc.Add();

    // should output 30 (and the stack is empty)
    printf("Result: %lf\n", calc.Pop());
}
```

So far, so good. Now suppose the developer of the `RPNCalculator` class realizes that working with any instance is not thread-safe, as the `std::stack<>` class is not thread-safe. The developer decides to add synchronization support by adding a `CRITICAL_SECTION` object as a private member, like so:

```
class RPNCalculator {
public:
    RPNCalculator();
    void Push(double value);
    double Pop();
    bool Add();
    bool Subtract();

private:
    std::stack<double> _stack;
    CRITICAL_SECTION _cs;
};
```

The relevant method implementations are updated to use the critical section internally. The developer recompiles the DLL and hands it off to the client developer. The client developer replaces his copy of the DLL (the old one) with the new updated version. From a C++ perspective, everything should be fine, since

the public methods in the class have not changed. Now the client application runs (without recompiling the code, as the public stuff has not changed). Can you guess what happens when the client app runs and the `SimpleCalc` function executes?

If you guessed "something bad" or "memory corruption", then you're absolutely correct. Can you spot the problem? Suppose the constructor is implemented in this way:

```
RPNCalculator::RPNCalculator() {
    ::InitializeCriticalSection(&_cs);
}
```

Fairly simple, but this causes a memory corruption. This is because the client code was **not** recompiled, so the size of an `RPNCalculator` object is the size of `std::stack<double>` - the client does not know of the `CRITICAL_SECTION` member, which causes the updated class code to trample stack memory. (if the allocation was dynamic, this would cause a heap corruption). This all happens because the C++ language has no binary compatibility, only source compatibility. We cannot use a "plug & play" approach, where a DLL can simply be replaced by a newer version, with the new functionality available without a hitch if the public surface of the class remains intact.

> There are other issues when using C++ for evolving components, but the above list should suffice as motivation for developing something better.

COM aims to solve these problems, by providing a binary standard of communication. This means the communication protocol between client and server is not based on the semantics of any specific programming language or platform; instead, communication is facilitated by defining some binary object layout that can be implemented by (theoretically) any language or platform, by adhering to the COM specification.

COM has many features and aspects, but it's built on top of two fundamental principles:

- Clients communicate with server objects using (abstract) interfaces, not concrete classes.
- Location transparency - the client does not need to know where a COM class implementation resides. Once the client obtains an interface pointer, it just makes calls and that's it. The server object may be in the same process, a different process on the same machine, or even a process on another machine (this is known as *Distributed COM* or DCOM, although it's not really different from standard COM).

Some of the terms used above need a more precise definition:

- *COM Interface* - a binary contract consisting of a set of methods with a well-defined binary layout.
- *COM Class* - an implementation of one or more COM interfaces.
- *COM Object* - an instance of a COM class.
- *COM Component/Server* - a deployment binary, consisting of one or more COM classes.

A COM Server can be *in-process* or *out-of-process*, depending on whether it's a DLL (loaded into the client's process) or an executable (launched in a separate process). Clients should be able to communicate with COM objects in either way. The in-process scenario is easier to understand, and is depicted in figure 21-1.

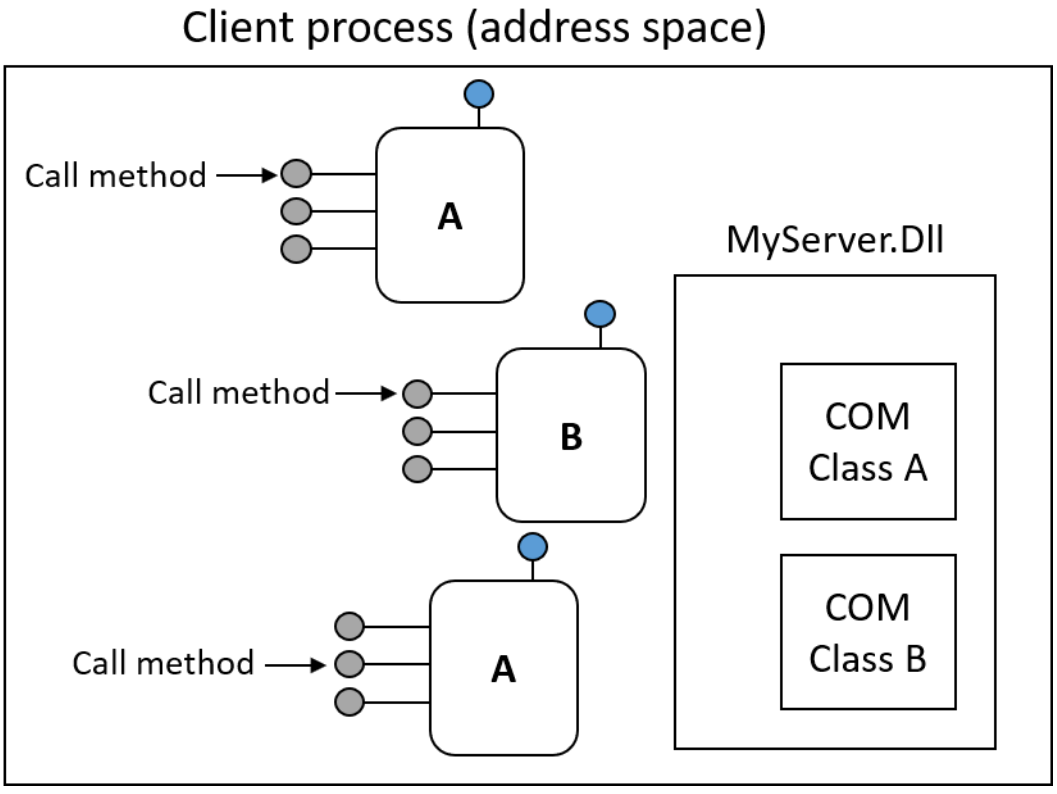


Figure 21-1: DLL (in process) COM Server

The *MyServer.Dll* implements two COM classes, A and B. From these implementations, three objects are created by the client: two instances of A and one instance of B. Then the client calls methods (invoking functionality) on one or more interfaces exposed by these objects. We have yet to discuss how the DLL is loaded, how objects are created, etc. We'll explain all these details as the chapter progresses.

In the out-of-process scenario, the COM infrastructure uses a *proxy* and *stub* object pair to facilitate transparent communication between client and server. The proxy is the server object's representation in the client's address space, while the stub is listening on the object's (server) side for method invocation requests. This layout is depicted in figure 21-2.

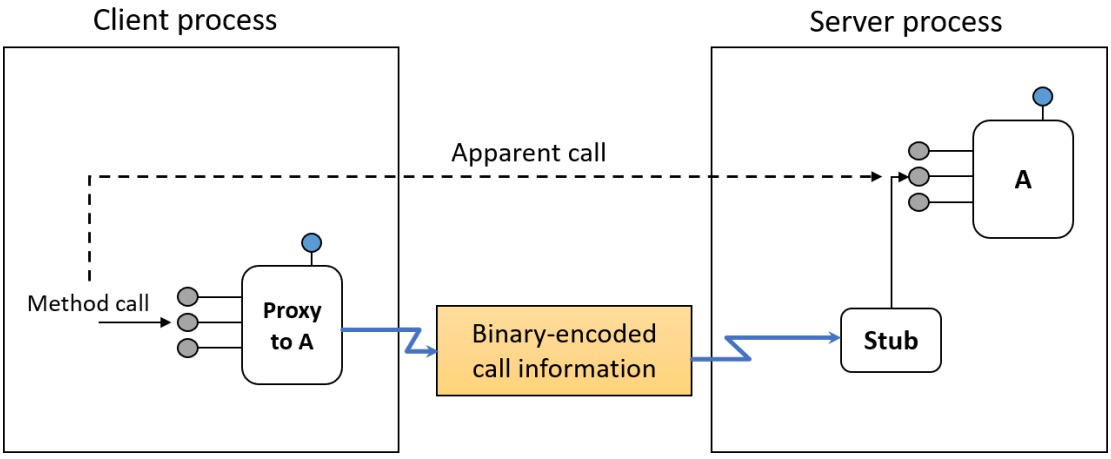


Figure 21-2: EXE (out of process) COM Server

Figure 21-2 demonstrates the location transparency property of COM. The client gets an interface pointer that looks like the real object - the proxy implements all the interfaces implemented by the real object - the client cannot (easily) and need not distinguish between a proxy and a real object. The client invokes methods normally (on the proxy). The proxy's job is to *marshal* the arguments of the method to the other side, so that the receiving end (the stub) can *unmarshal* the arguments and call the real object. This *marshaling* procedure is also performed in reverse for values that need to be returned back to the client. All this marshaling behavior is provided by the "magic" of COM out-of-process invocation.

Interfaces and Implementations

The most fundamental entity in COM is the interface. A COM interface is a binary contract, meaning that once defined and exposed to the clients, an interface should be immutable. This means it should never again change, as existing clients depend on the binary layout and semantics of the interface. If a change or extension is to be introduced, it must be done by defining a new interface (it may inherit all the methods from the original interface or by creating a completely new one).

In C++, a COM interface definition consists of a class with all functions defined as pure virtual. Here is an example of an interface that could be used with an RPN calculator implementation:

```
struct IRPNCalculator {
    virtual void Push(double value) = 0;
    virtual double Pop() = 0;
    virtual bool Add() = 0;
    virtual bool Subtract() = 0;
};
```

It's customary to name interfaces starting with a capital `I`.

With the above definition, the class implementation can be changed in terms of the interface:

```
class RPNCalculator : public IRPNCalculator {
public:
    void Push(double value) override;
    double Pop() override;
    bool Add() override;
    bool Subtract() override;

private:
    std::stack<double> _stack;
    CRITICAL_SECTION _cs;
};
```

Does the above change solve the crashing problem we observed earlier? Not yet. The next step is to hide the `RPNCalculator` implementation from the client entirely. The header exposed to clients should contain two pieces: the interface definition, and a factory function to create an instance. Here's what that may look like:

```
// RPNCalculatorClient.h

struct IRPNCalculator {
    virtual void Push(double value) = 0;
    virtual double Pop() = 0;
    virtual bool Add() = 0;
    virtual bool Subtract() = 0;
};

extern "C" IRPNCalculator* CreateCalculator();
```

The client has no idea where the interface is implemented or how. Creating an instance is now turned over to the server. This is ideal, since the server knows which implementation to create, and it's always going to have the correct size. Here is a revised client code:

```
void SimpleCalc() {
    IRPNCalculator* calc = CreateCalculator();
    if (calc) {
        calc->Push(10);
        calc->Push(20);
        calc->Add();
        printf("Result: %f\n", calc->Pop());

        // not ideal (see later)
```

```
        delete calc;
    }
}
```

The factory is defined with C linkage, because as established earlier, C is the lowest common denominator that is supported for exporting functions by literally all languages/platforms.

This is almost perfect. If the implementation changes, for example by adding or changing data members in the behind-the-scenes implementation, the client should not be affected. This is because the binary layout of the interface remains the same. With a C++ implementation, the virtual table mechanism is used to implement virtual functions, which provide the exact layout defined by a COM interface, making C++ a natural choice for COM class implementations. This layout is depicted in figure 21-3.

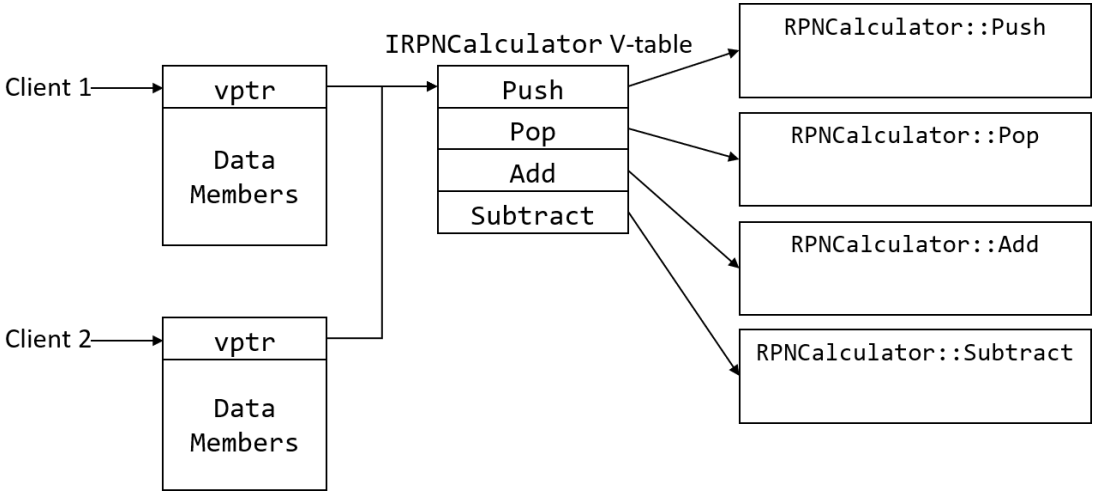


Figure 21-3: Virtual dispatch mechanism

All the client sees is the virtual table pointer (`vptr`) that is always the first member of any instance with virtual functions. The data members of the implementation are unknown (hidden) to the client. In fact, there is no way for the client to query any implementation details, such as the size of the object. This is great, as the implementation can be changed freely without any need for the client to recompile anything, so long as the interface remains the same - the same function order and the same parameters (the function names themselves mean nothing as the invocation is based on the offset of the function pointer within the v-table).

The above `SimpleCalc` still has one snag. To free the object, it calls the C++ `delete` operator. This assumes the object's memory was allocated with the C++ `new` operator.

> Using `delete` in this way has even more hidden assumptions: the client and server must use the same compiler. Both use the same C++ runtime library (dynamic vs.static), and that there is no operator overloading for `new` and `delete` in the server's class implementation).

This assumption (or assumptions) are too problematic. The solution is to transfer the responsibility of freeing the object back to the server (just as was done with object creation). This could be done by adding another method to every COM interface like so:

```
struct IRPNCalculator {
    virtual void Release() = 0;
    virtual void Push(double value) = 0;
    //...
};
```

All the client needs to do is call the `Release` method and not have to know anything about how the object's memory was allocated.

The `IUnknown` Interface

The previous section looked at some problematic details of the C++ language, and how separating interface from implementation could solve these issues. Now it's time to introduce the "official" COM definitions that are based on the principles outlined in the previous section.

COM defines a base interface, from which all interfaces must derive (extend). This ensures certain functionality is always available given any COM interface. This interface is called `IUnknown` and defined like so:

```
struct IUnknown {
    virtual HRESULT __stdcall
        QueryInterface(const IID& riid, void** ppv) = 0;
    virtual ULONG   __stdcall AddRef() = 0;
    virtual ULONG   __stdcall Release() = 0;
};
```

COM defines semantics for managing an object's lifetime. Instead of just providing a way to create and destroy an object, two methods are defined on `IUnknown` - `AddRef` and `Release` to manage the object's reference count (and indirectly - its lifetime). Whether the class implementation actually uses reference counting or not is no concern of COM, but the rules are clear: if the client receives an interface pointer, it must eventually call `Release`. `AddRef` may be used to artificially increment the object's reference count before passing the interface pointer to an independent entity (such as a separate thread of execution). In this way, each client works with the object safely until the pointer is no longer needed. Then the client calls `Release` on its pointer, and from that point on the pointer should be considered poison. The object may or may not be destroyed, but that should not matter to the particular client.

> All COM interface methods must use the standard calling convention (`__stdcall`). This is required as part of the binary interface - the choice seems arbitrary, but nevertheless some choice must be made so that clients and servers are in sync.

The first function in `IUnknown`, `QueryInterface` is concerned with querying the object for another interface that may or may not be supported by the object. Identifying interfaces, like most other COM entities, is done with Globally Unique Identifiers (GUIDs, also called Universally Unique Identifiers - UUIDs). These 128-bit numbers are generated by an algorithm that statistically guarantees uniqueness across time and space. Since 128-bit numbers cannot yet be represented in C/C++ as simple types, the `GUID` structure is defined to hold such a value. It has several alternative `typedef`s such as `IID` and `CLSID`, that have slightly different semantic meanings, but are otherwise identical from the binary perspective:

```
typedef struct _GUID {
    unsigned long  Data1;
    unsigned short Data2;
    unsigned short Data3;
    unsigned char  Data4[ 8 ];
} GUID;
```

GUIDs can be generated programmatically as needed by calling `CoCreateGuid`:

```
HRESULT CoCreateGuid(_Out_ GUID* pguid);
```

You might encounter some strange macros in the Windows header files, such as `FAR` that expand to nothing. This is a relic from 16-bit Windows, where there were near and far pointers.

Visual Studio provides a tool called *Create GUID*, normally accessible from the *Tools* menu, that calls `CoCreateGuid` and provides several formatting options (figure 21-4).
Each time you click *New GUID*, a new GUID is generated anf formatted. Clicking *Copy* copies the selected format to the clipboard. We'll use this tool later to generate GUIDs for COM components we'll author.

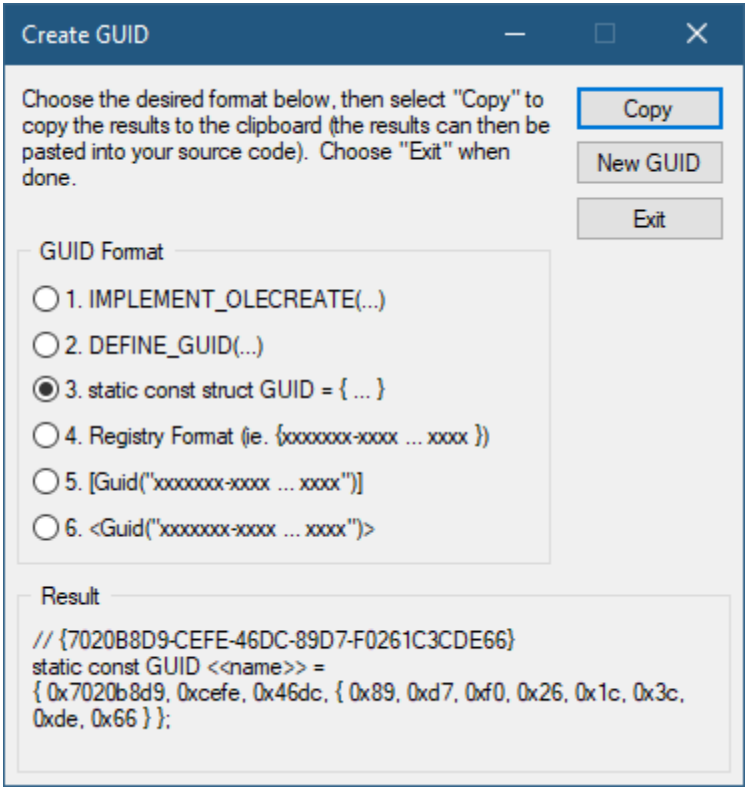


Figure 21-4: The Create GUID tool

GUIDs are used everywhere in COM. For example, interfaces are identified by GUIDs, as using strings such as "IRPNCalculator" are not globally unique. This means that the `IUnknown` interface has its own GUID, defined in the earliest days of COM. It's identified in the Windows headers with the variable `IID_IUnknown`.

This brings us back to the `QueryInterface` method from `IUnknown`. Its first argument is the interface ID to query for (interface names don't mean anything and can be projected differently by other languages/platforms). The result of the query is stored in the second argument (a pointer to a void pointer). The method returns an `HRESULT` - the standard COM return value, indicating success or failure of the operation.

HRESULTs

Most COM methods encountered return an `HRESULT` type. An `HRESULT` is just a signed 32-bit integer that conveys an error or success code. If the most significant bit (MSB) is set, this indicates an error. Looking back at the `IUnknown` interface, it's clear `AddRef` and `Release` do not return `HRESULT`s. These methods should be the only exceptions.

The COM standard specifies that `AddRef` and `Release` should return zero if the object is destroyed, and non-zero value otherwise. The return value is not required to be the actual reference count of the object, but often is. It follows that `AddRef` should never return zero, as it's only possible call `AddRef` on a live object.

The standard `HRESULT` success code is `S_OK` (0). `QueryInterface` returns `S_OK` if the requested interface is supported (and its pointer returned in `*ppv`, otherwise it returns `E_NOINTERFACE`, indicating the interface is not supported. Some of the common error codes are shown in table 21-1.

Table 21-1: Common failure `HRESULT`s

`HRESULT` symbol	Description
`E_NOINTERFACE`	Interface is not supported
`E_POINTER`	Pointer is not valid (typically `NULL` when it shouldn't be)
`E_UNEXPECTED`	Unexpected call to this method at this time
`E_NOTIMPL`	Functionality is not implemented
`E_OUTOFMEMORY`	Not enough memory to complete the operation
`E_INVALIDARG`	One or more invalid arguments to the call
`E_FAIL`	Generic failure

The general layout of an `HRESULT` contains three parts: the *facility* that identifies the category of result, the error or success, and the actual code. The `HRESULT` macros are built based on these three parts in the following way: `FACILITY_S/E_CODE`. The facility for all the values in table 21-1 is `FACILITY_NULL`, that does not show up in the named constants.

Checking for an `HRESULT` is typically done using the `FAILED` or `SUCCEEDED` macros. These return `true` on a failed or successful `HRESULT`, respectively.

More specific `HRESULT` values are defined in the SDK headers, and new ones may be defined by components as needed. If a new `HRESULT` is to be defined, bit 29 should be set for custom `HRESULT` values. Here is a ficticious example for querying for some custom interface and handling an error:

```
void DoWork(IUnknown* p) {
    ICalculator* pCalc;
    HRESULT hr = p->QueryInterface(IID_ICalculator,
        reinterpret_cast<void**>(&pCalc));
    if(SUCCEEDED(hr)) {
        // successful call to QueryInterface increments
        // the object's reference count
        int result;
        hr = pCalc->Add(4, 6, &result);
        if(FAILED(hr)) {
            printf("Failed in call to Add (0x%08X)\n", hr);
```

```
        }
        // done working with the interface
        pCalc->Release();
    }
    else {
        printf("Interface not supported\n");
    }
}
```

You may wonder why `reinterpret_cast` is needed in the above code. Can't we cast `ICalculator**` to `void**` implicitly? The answer is no as these types have no inheritance relationship, so the cast is necessary. If you feel lazy, you can use a C-style cast `(void**)` instead of `reinterpret_cast`.

COM Rules (pun intended)

There are several rules concerning reference counting and `IUnknown` that are worth listing explicitly:

- Whenever an interface pointer is returned via `QueryInterface`, the client should assume the reference count of the object was incremented, meaning the client must eventually call `Release` on that interface pointer, or else that object will leak.
- `QueryInterface` implementations must be symmetric: if it's possible to query for interface `IX` from `IY`, it should be possible to go the other way around.
- It should always be possible to query for the same interface calling `QueryInterface`.
- If one can get from `IX` to `IY` and from `IY` to `IZ`, then it should also be possible to directly request `IZ` from an `IX` interface pointer.
- The `IUnknown` pointer serves as the object's identity. This means an object should always return the same `IUnknown` pointer, no matter from which interface it's requested.
- All COM interfaces must inherit from `IUnknown`, directly or indirectly. This means that the first 3 methods of every interface are `QueryInterface`, `AddRef`, and `Release`, in that order.

Our early `IRPNCalculator` interface was not COM compliant. Let's turn it into a proper COM interface:

```
#include <Unknwn.h>  // including <Windows.h> also includes this one

struct IRPNCalculator : IUnknown {
    virtual HRESULT __stdcall Push(double value) = 0;
    virtual HRESULT __stdcall Pop(double* value) = 0;
    virtual HRESULT __stdcall Add() = 0;
    virtual HRESULT __stdcall Subtract() = 0;
};
```

The interface has seven methods, including those inherited from `IUnknown`. Ordinary return values (such as from `Pop`) turned into parameters with an extra level of indirection (`double` turned into `double*`) because of the requirement to return `HRESULT` from every method.

There is still something missing from the interface - its GUID. We'll get to that in the section "COM Servers", later in this chapter.

COM Clients

We now have enough information to start working with COM as clients, which is considerably easier than creating a COM server. The following example will use the *Background Intelligent Transfer Service* (BITS) API to download a file in the background. BITS is implemented as a Windows Service (see chapter 20 for more on services), but that fact is not really relevant to the way a BITS client invokes BITS functionality thanks to the location transparency property of COM, as we shall see.

We'll start with a standard C++ console application (named *BitsDemo* in the accompanying source code). First, we'll add the usual includes and an extra one for the BITS API:

```
#include <Windows.h>
#include <stdio.h>
#include <Bits.h>       // BITS API
```

The steps we need to take to download a file using the BITS service are outlined below:

1. Initialize COM for this thread.
2. Create an instance of the BITS manager.
3. Create a BITS job for download.
4. Add a URL to download from and local file to store the resulting file.
5. Initiate the transfer.
6. Wait for the transfer to complete.
7. Display final results.
8. Release the various objects created in the previous steps.

Most of the above steps are specific to the way the BITS API is to be used, but some of the steps are generic and applicable to all COM clients.

Step 1: Initialize COM

Before making any COM-related API call from a thread (there are very few exceptions to this rule), COM must be initialized for that thread. This can be done with a call to one of the following functions: `CoInitialize` or the extended `CoInitializeEx`:

```
HRESULT CoInitialize(_In_opt_ LPVOID pvReserved);
HRESULT CoInitializeEx(
    _In_opt_ LPVOID pvReserved,
    _In_ DWORD dwCoInit);
```

`CoInitialize` can only accept `NULL`, and in fact is just special case of `CoInitializeEx`. It's equivalent to calling `CoInitializeEx(nullptr, COINIT_APARTMENTTHREADED)`.

Although the term "initializing COM" may be good enough to proceed to the next step, it would be more accurate to say that the function puts the calling thread into an apartment whose type depends on the `dwCoInit` parameter (and is always a single-threaded apartment in the `Coinitialize` case). A comprehensive discussion of apartments is reserved for the section "Threads and Apartments". For now, we'll just call one of the functions as the first line inside `main` and move on to step 2:

```
::CoInitialize(nullptr);
```

Step 2: Create the BITS Manager

As with any API, proper usage requires reading the API's documentation. The next steps are based on the BITS API official documentation. Accessing BITS functionality requires creating an instance of a COM class called the *Background Copy Manager*. In general, creating instances of COM classes is achieved (in most cases) by calling `CoCreateInstance`:

```
HRESULT CoCreateInstance(
    _In_ REFCLSID rclsid,
    _In_opt_ LPUNKNOWN pUnkOuter,
    _In_ DWORD dwClsContext,
    _In_ REFIID riid,
    _COM_Outptr_ LPVOID* ppv);
```

> COM object creation is also referred to as *Activation*.

The prefix `Co` that most COM APIs use stands for "Component Object".

The purpose of `CoCreateInstance` is to create an instance of a given class and return a requested interface pointer to the new object. The first parameter identifies the class itself with a GUID, referred to here as class ID (CLSID), to make it easier to understand, but it's a GUID like any other. COM classes - implementations of COM interfaces - are identified by GUIDs, giving them unique names. The CLSID is looked up in the Windows Registry, as we shall see in the next section. The interface ID (supplied as the fourth parameter) is not good enough, as one interface can have any number of implementations.

The second parameter to `CoCreateInstance` is an `IUnknown` pointer called "outer IUnknown". This parameter is related to a COM extensibility mechanism called *Aggregation*. Aggregation is beyond the scope of this chapter, and if not used (the typical case), `NULL` is specified.

The next parameter (`dwClsContext`) indicates (mostly) which context the server should be allowed to run in. The most common value is `CLSCTX_ALL`, which means the client doesn't care, and would accept any implementation. Here are some Common alternatives:

- `CLSCTX_INPROC_SERVER` - in-process (DLL) implementation.
- `CLSCTX_LOCAL_SERVER` - out-of-process (EXE) implementation on the same machine as the client.
- `CLSCTX_REMOTE_SERVER` - out-of-process (EXE) implementation on another machine (used with `CoCreateInstanceEx`).

Multiple flags can be specified by using the or (`|`) operator. The function will attempt to get the "closest" implementation (i.e. DLL preferred over EXE, local EXE preferred over a remote server).

The next parameter is the interface ID requested from the new object. This can be `IUnknown` (`IID_IUknown`), which must always be supported, or a more specific interface that is known to be implemented by the class. If `IUnknown` is requested, another interface can later be obtained by calling `QueryInterface`.

The last parameter is the resulting pointer, if the call succeeds. As usual, the return value is an `HRESULT`, that is `S_OK` with a successful call.

Given this information and the BITS docs, we need to make the following call:

```
IBackgroundCopyManager* mgr;
HRESULT hr = ::CoCreateInstance(CLSID_BackgroundCopyManager,
    nullptr, CLSCTX_ALL,
    IID_IBackgroundCopyManager, reinterpret_cast<void**>(&mgr));
if (FAILED(hr))
    return Error(hr);
```

CLSIDs are traditionally prefixed with `CLSID_` and interface IDs with `IID_`. The above GUIDs are defined in the *bits.h* header.

> The last two arguments in the above `CoCreateInstance` call can be shortened by using the `IID_PPV_ARGS` macro like so:
>
> ```
> HRESULT hr = ::CoCreateInstance(CLSID_BackgroundCopyManager,
> nullptr, CLSCTX_ALL, IID_PPV_ARGS(&mgr));
> ```

```
```

`Error` is just a simple function that displays the `HRESULT`:

```
int Error(HRESULT hr) {
    printf("COM error (hr=%08X)\n", hr);
    return 1;
}
```

The process of locating the server is described in the next section. For the purposes of this demo, if we receive back a proper interface, we can move on to step 3.

Step 3: Create a BITS Job

The `IBackgroundCopyManager` interface is defined like so (copied from *bits.h*):

```
MIDL_INTERFACE("5ce34c0d-0dc9-4c1f-897c-daa1b78cee7c")
IBackgroundCopyManager : public IUnknown {
public:
 virtual HRESULT STDMETHODCALLTYPE CreateJob(
  /* [in] */ __RPC__in LPCWSTR DisplayName,
  /* [in] */ BG_JOB_TYPE Type,
  /* [out] */ __RPC__out GUID *pJobId,
  /* [out] */ __RPC__deref_out_opt IBackgroundCopyJob **ppJob) = 0;
 virtual HRESULT STDMETHODCALLTYPE GetJob(
  /* [in] */ __RPC__in REFGUID jobID,
  /* [out] */ __RPC__deref_out_opt IBackgroundCopyJob **ppJob) = 0;
 virtual HRESULT STDMETHODCALLTYPE EnumJobs(
  /* [in] */ DWORD dwFlags,
  /* [out] */ __RPC__deref_out_opt IEnumBackgroundCopyJobs **ppEnum)
    = 0;
 virtual HRESULT STDMETHODCALLTYPE GetErrorDescription(
  /* [in] */ HRESULT hResult,
  /* [in] */ DWORD LanguageId,
  /* [out] */ __RPC__deref_out_opt LPWSTR *pErrorDescription) = 0;
};
```

At first glance, this doesn't appear to be human-written code - it looks machine-generated. And indeed it is, as can be seen from reading the first line in *bits.h*:

```
/* this ALWAYS GENERATED file contains the definitions for the interface\
s */
```

The file was generated from an *Interface Definition Language* (IDL) file that was compiled by the Microsoft IDL (MIDL) compiler. The reason for using yet another file format for generating interfaces is discussed in the section "The Interface Definition Language", later in this chapter. For now, let's make sure we understand the above interface definition.

The `STDMETHODCALLTYPE` macro expands to `__stdcall`, as is required by all COM methods. All the `__RPC`-something macros are SAL annotations. The `MIDL_INTERFACE` is defined like so:

```
#define MIDL_INTERFACE(x)   struct DECLSPEC_UUID(x) DECLSPEC_NOVTABLE
```

Continuing with the remaining macros:

```
#define DECLSPEC_UUID(x)    __declspec(uuid(x))
#define DECLSPEC_NOVTABLE   __declspec(novtable)
```

`MIDL_INTERFACE` defines an interface by using the `struct` keyword (as C++ has no special keyword for interfaces), and uses two Visual C++ specific attributes. `uuid(x)` associates a GUID with the definition, which helps to simplify code by not requiring usage of an `IID_IX` variable, instead opting for the more elegant `__uuidof(IX)` operator that uses the attached GUID. This can also be used by classes, by the way, as it is with the Background Copy Manager:

```
class DECLSPEC_UUID("4991d34b-80a1-4291-83b6-3328366b9097") BackgroundCo\
pyManager;
```

All this means that the initial `CoCreateInstance` code can be simplified like so:

```
HRESULT hr = ::CoCreateInstance(__uuidof(BackgroundCopyManager),
    nullptr, CLSCTX_ALL,
    __uuidof(IBackgroundCopyManager), reinterpret_cast<void**>(&mgr));
```

All this is just compiler trickery and does not have any effect at runtime.

The macro `IID_PPV_ARGS` uses the `__uuidof` operator, without which it cannot "get" the interface ID.

A new BITS job is created by calling `IBackgroundCopyManager::CreateJob`, rather than requiring its own `CoCreateInstance` call. This is a very common pattern, as it allows the creation function to accept any extra parameters needed and provides control over instance creation (and does not require any Registry settings).

```
GUID jobId;
IBackgroundCopyJob* pJob;
hr = mgr->CreateJob(L"My job", BG_JOB_TYPE_DOWNLOAD, &jobId, &pJob);
if (FAILED(hr))
    return Error(hr);
```

The call to `CreateJob` returns a new interface pointer on a BITS job object. It also returns a GUID identifying this job. This GUID has nothing to do with COM, and is used internally by BITS to uniquely identify transfers. The inputs are a display name, which can be anything, and whether the job is a download or upload.

We can now technically release the interface pointer to `IBackgroundCopyManager` - we won't be needing it again:

```
mgr->Release();
```

Remember, this doesn't necessarily mean the object is destroyed. It's possible (and in fact probable) that the job object holds another interface pointer to the BITS manager. Regardless, as clients, we don't care. Once we don't need an interface pointer we received earlier, we release it.

Step 4: Add a Download

The next step is to add at least one file to download:

```
hr = pJob->AddFile(
    L"https://www.fnordware.com/superpng/pnggrad16rgb.png",
    L"c:\\temp\\image.png");
if (FAILED(hr))
    return Error(hr);
```

The first parameter is the remote URL to download from (or upload to if this was an upload job). The second parameter is the local file to which the remote is to be downloaded.

I've selected some fairly random image URL, that may or may not continue to work by the time you read this. If this does not work, find some other file to download from the web.

We can add more files if we want for the same job, but in this example we'll continue with just one.

Step 5: Initiate the Transfer

Starting the transfer is fairly easy with a call to `IBackgroundCopyJob::Resume`:

```
hr = pJob->Resume();
```

BITS works asynchronously, so `Resume` should start the download and return immediately. We need to know when the transfer is complete (or there is some error). BITS provides two ways to do this: synchronously and asynchronously. We'll use the synchronous option here, and discuss the asynchronous option later in this chapter, as it requires us to implement a callback interface.

Step 6: Wait for Transfer to Complete

We'll have a loop that polls the job object every some interval and query the state of the transfer, exiting the loop when it's done:

```
if (SUCCEEDED(hr)) {
    printf("Downloading... ");
    BG_JOB_STATE state;
    for (;;) {
        pJob->GetState(&state);    // assume it cannot fail
        if (state == BG_JOB_STATE_ERROR
            || state == BG_JOB_STATE_TRANSFERRED)
            break;
        printf(".");
        ::Sleep(300);
    }
```

`IBackgroundCopyJob::GetState` returns the state of the transfer. The code waits for a successful completion or some error to break out of the loop. It uses a `Sleep` call to prevent CPU hogging, as this is a network transfer.

Step 7: Display Results

Once out of the loop, we can show the results:

```
    if (state == BG_JOB_STATE_ERROR) {
        printf("\nError in transfer!\n");
    }
    else {
        pJob->Complete();
        printf("\nTransfer successful!\n");
    }
}
pJob->Release();
```

The call to `Complete` is required, as BITS stores the downloaded file with a temporary name generated by BITS. The call to `Complete` flushes any remaining bytes and renames the file to the client-provided name. Finally, the job interface pointer is released.

Step 8: Clean Up

We already released the manager and job interfaces. There is very little left to do - just uninitialize COM before the thread exits:

```
    ::CoUninitialize();
    return 0;
}
```

Run the application and you should see an output like the following:

```
Downloading... ..............
Transfer successful!
```

If some error occurs, you should get an error output (one simple way to get this is to modify the URL to a non-existing one):

```
Downloading... ...
Error in transfer!
```

If an error occurs, BITS provides extended error information by calling `IBackgroundCopyJob::GetError`. Add code to display rich error information in case of a failure with the transfer. Remember to follow COM rules regarding interface pointers.

COM Smart Pointers

The rules concerning `AddRef` and `Release` calls are not complicated when viewed in isolation, but in practice it's difficult to keep track of all interface pointers going around. As a consequence, it's easy to miss calling `Release` especially when an interface pointer is no longer needed in all code paths using that pointer. This is where COM smart pointers come in, automating calls to `AddRef` and `Release` so that explicit calls to these methods is rarely needed.

As a bonus, these smart pointers also provide easier access to `QueryInterface` by overloading constructors and the assignment operators. Several COM smart pointers are available in the Windows SDK today:

- *<comdef.h>* has definitions for smart pointers based on a class named `_com_ptr_t,.`. These throw C++ exceptions for failed `QueryInterface` calls.
- ATL provides two smart pointer classes that don't ever throw exceptions.
- The newer *Windows Runtime Library* (WRL) provide its own version of smart pointers (`ComPtr<>`), that are more verbose than the ATL ones.

Which smart pointer class you use is mostly a matter of taste. As this book uses ATL and WTL, I will demonstrate the usage of the ATL smart pointers, which I also personally prefer for their convenience and simplicity.

The two classes most often used are `CComPtr<>` and `CComQIPtr<>`, the latter having extra constructors that call `QueryInterface` when faced with a different typed interface.

What makes a pointer "smart"? In the COM case, it's about convenient constructors and destructor, that call `AddRef`, `QueryInterface` and `Release` as appropriate, and operator overloading for dereferencing (`*`, `->`) and address-of (`&`), as we shall see in the next code snippet.

The previous section used raw pointers (sometimes dubbed "stupid pointers") to access BITS functionality. Here is the equivalent code using the ATL smart pointers:

```
#include <atlcomcli.h>

HRESULT DoBITSWork() {
    // assume CoInitialize has already been called for this thread

    CComPtr<IBackgroundCopyManager> spMgr;
    HRESULT hr = spMgr.CoCreateInstance(
        __uuidof(BackgroundCopyManager));
    if (FAILED(hr))
        return hr;

    CComPtr<IBackgroundCopyJob> spJob;
    GUID guid;
    hr = spMgr->CreateJob(L"My Job", BG_JOB_TYPE_DOWNLOAD,
        &guid, &spJob);
    if (FAILED(hr))
        return hr;

    hr = spJob->AddFile(
        L"https://www.fnordware.com/superpng/pnggrad16rgb.png",
        L"c:\\temp\\image.png");
    if (FAILED(hr))
        return hr;

    hr = spJob->Resume();
    if (SUCCEEDED(hr)) {
        printf("Downloading... ");
        BG_JOB_STATE state;
        for (;;) {
            spJob->GetState(&state);
            if (state == BG_JOB_STATE_ERROR
```

```
                || state == BG_JOB_STATE_TRANSFERRED)
                break;
            printf(".");
            ::Sleep(300);
        }
        if (state == BG_JOB_STATE_ERROR) {
            printf("\nError in transfer!\n");
        }
        else {
            spJob->Complete();
            printf("\nTransfer successful!\n");
        }
    }
    return hr;
}
```

The function starts by including *<atlcomcli.h>* where the ATL smart pointers are defined. Alternatively, you can include *<atlbase.h>*, which has some useful extras and also includes *<atlcomcli.h>*.

The various interface methods are exposed directly by `CComPtr<>` because the `->` operator is overloaded and returns the internal interface pointer. Notice there are no `Release` calls in the above code. In fact, trying to call `Release` as the following example shows fails to compile:

```
spMgr->Release();
```

This is intentional, as allowing this call to `Release` will likely cause a crash, since the destructor, unaware that `Release` has been called, will attempt another `Release` call, which is one too many. The class *does* provide a `Release` method that can be called so that the interface is released early (before the destructor runs):

```
spMgr.Release();
```

This call invokes the internal interface's `Release` and sets the interface pointer to `NULL` so that the destructor does not call `Release` again (seeing the interface pointer is `NULL`). An equivalent call is setting the object to `NULL` (operator overloading at work):

```
spMgr = nullptr;
```

The original call to `CoCreateInstance` is replaced in the above code by the `CoCreateInstance` method exposed by the `CComPtr<>` class. This is just a helpful shortcut that calls `CoCreateInstance`, but provides the defaults of `NULL` for the outer `IUnknown` and `CLSCTX_ALL` for the class context.

Write a console application that lists all currently active BITS jobs by calling `IBackgroundCopyManager::EnumJobs`. Display each job's display name, state, description, priority, GUID, creation time, and progress. Use smart pointers.

My *BITSManager* tool shows how to accomplish this (https://github.com/zodiacon/BITSManager).

Querying for Interfaces

The BITS job object implements more than the `IBackgroundCopyJob` interface - it also supports the extended interfaces `IBackgroundCopyJob2` and `IBackgroundCopyJob3`. However, these interfaces may or may not be supported based on the BITS version on the machine. This extended functionality was added after the first BITS version was out, so new interfaces had to be defined. The new interfaces may inherit (extend) an existing interface, or be unrelated (inheriting directly from `IUnknown`). This is left to the discretion of those defining the new interface(s).

A client that wants to gain access to the new functionality must query for it and be ready to handle failure gracefully. Here is an example for working with one of the extended job interfaces:

```
// after the job is created (in spJob)

// leading space for SetNotifyCmdLine
WCHAR localPath[] = L" c:\\temp\\image.png";

CComPtr<IBackgroundCopyJob2> spJob2;
hr = spJob->QueryInterface(__uuidof(IBackgroundCopyJob2),
    reinterpret_cast<void**>(&spJob2));
if (spJob2) {    // checking HR is ok too
    hr = spJob2->SetNotifyCmdLine(
        L"c:\\windows\\system32\\mspaint.exe", localPath);
    // interface pointer released here
}

hr = spJob->AddFile(
    L"https://www.fnordware.com/superpng/pnggrad16rgb.png",
    localPath + 1);
//  rest of code is unchanged
```

If you try this out, *mspaint* should come up automatically when the file is downloaded successfully.

The caller must be prepared for the interface not being implemented and handle that appropriately. In some cases, the caller just won't use the new functionality. In other cases, it may be more appropriate to report an error and notify the user that a newer library is required for proper functionality.

The above `QueryInterface` (QI) call uses the real `QueryInterface` method to make the call, providing the interface ID and the address of a pointer to fill in on success. The ATL smart pointers, however, provide simplified ways to achieve the same thing. Here are more options for making the same QI call:

```
// leverages the __uuidof operator
hr = spJob.QueryInterface(&spJob2);

// uses CComQIPtr<>
CComQIPtr<IBackgroundCopyJob2> spJob2(spJob);
if(spJob2) {    // no HRESULT to examine
    // interface available
}
```

Make sure you release an interface pointer if you plan to reuse it for a future create or QI, otherwise you'll get an assertion failure from the ATL smart pointers.

`CoCreateInstance` Under the Hood

The location transparency of COM made working with the various BITS objects fairly easy. Once an interface pointer is obtained successfully, methods can be directly invoked regardless of whether the returned pointer is to a local object (in the same process) or a proxy (the real object may be in another process).

This is all good and well, but to gain a deeper understanding of COM, we need to look beneath the surface into the workings of `CoCreateInstance`. Such understanding also helps in troubleshooting COM-related issues.

`CoCreateInstance` is in fact a wrapper around a more fundamental function, `CoGetClassObject`, whose purpose is to obtain a class object (class factory) for creating instances of the requested type:

```
HRESULT CoGetClassObject(
    _In_      REFCLSID rclsid,
    _In_      DWORD dwClsContext,
    _In_opt_  COSERVERINFO* pServerInfo,
    _In_      REFIID riid,
    _Outptr_  LPVOID* ppv);
```

The first parameter is the CLSID identifying the type of object to create, the same one we find in `CoCreateInstance`. `dwClsContext` is the class context, having the same meaning as for `CoCreateInstance`. `pServerInfo` is an optional pointer to a `COSERVERINFO` structure which allows specifying a computer name on which the activation is attempted (DCOM). If `NULL` is specified, the object is to be created on the local machine (unless a Registry value redirects that to another machine). We'll be dealing with `NULL` values for this argument in this chapter (this is what is sent by `CoCreateInstance`). The last two parameters are the requested class object interface and (as usual) the address of a pointer to be filled on successful invocation.

The COM infrastructure supports using a class factory to create possibly multiple objects with the factory indirection. Although it's a nice idea, most COM activations use the standard `IClassFactory` interface to obtain a class factory. Here is its definition:

```
MIDL_INTERFACE("00000001-0000-0000-C000-000000000046")
IClassFactory : public IUnknown {
public:
    virtual HRESULT STDMETHODCALLTYPE CreateInstance(
        _In_opt_  IUnknown *pUnkOuter,
        _In_  REFIID riid,
        _COM_Outptr_  void **ppvObject) = 0;
    virtual HRESULT STDMETHODCALLTYPE LockServer(BOOL fLock) = 0;
};
```

The important method is `CreateInstance` which a proper factory must implement. This call creates the instance and returns (on success) the requested initial interface implemented by the object. Since the `IClassFactory` interface is so common as a factory, `CoCreateInstance` always asks for it. This means, `CoCreateInstance` is implemented roughly like so:

```
HRESULT CoCreateInstance(REFCLSID rclsid, LPUNKNOWN pUnkOuter,
    DWORD dwClsContext, REFIID riid, void** ppv) {
    IClassFactory* pCF;
    HRESULT hr = ::CoGetClassObject(rclsid, dwClsContext, nullptr,
        __uuidof(IClassFactory), reinterpret_cast<void**>(&pCF));
    if(FAILED(hr))
        return hr;

    // ask the factory to create the object
    hr = pCF->CreateInstance(pUnkOuter, riid, ppv);
    pCF->Release();
    return hr;
}
```

`CoGetClassObject`

Much of the COM infrastructure is rooted in the Registry. `CoGetClassObject` first searches the CLSID given in the Registry under *HKEY_CLASSES_ROOT\CLSID*. This is where all the class IDs are registered, essentially mapping between a GUID and an implementation file (DLL/EXE).

Let's look at the three common cases for activation: DLL, EXE, and service.

DLL Activation

DLL based activation is recognized in the Registry by having a subkey named *InProcServer32*, where the default value in that key points to the path of the DLL. Figure 21-5 shows an example - the *RDPSession*

COM class (“9B78F0E6-3E05-4A5B-B2E8-E743A8956B65”) that may be used for sharing a desktop with a viewer.

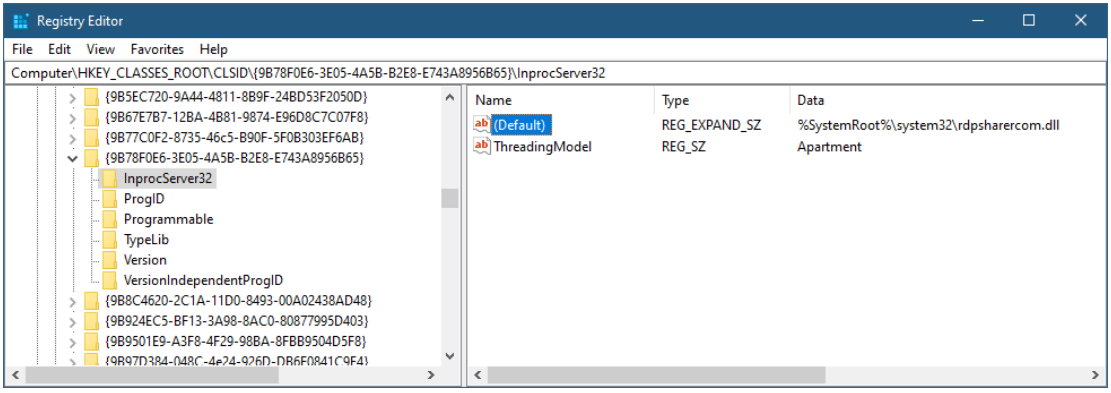


Figure 21-5: DLL activation in the Registry

`CoGetClassObject` performs the following operations once the DLL is located:

- Loads the DLL (`CoLoadLibrary`, similar to `LoadLibrary`).
- Calls `GetProcAddress` looking for an exported function named `DllGetClassObject`.

`DllGetClassObject` must have the following prototype:

```
HRESULT __stdcall
DllGetClassObject(REFCLSID rclsid, REFIID riid, void** ppv);
```

- `CoGetClassObject` then calls `DllGetClassObject` and passes along the CLSID and the interface ID of the class factory, expecting to get back a valid interface pointer. This pointer is returned to the caller.

These steps are summarized in figure 21-6.

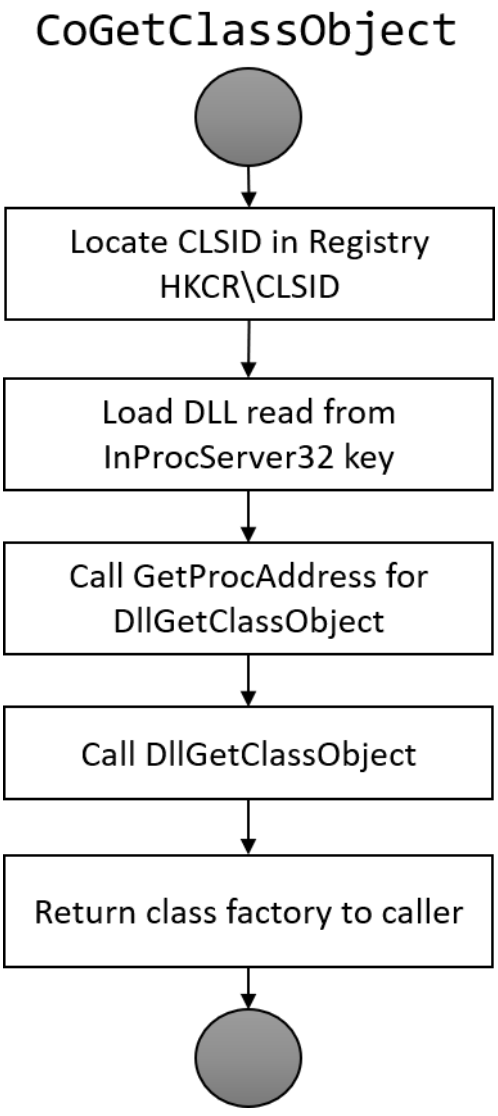


Figure 21-6: `CoGetClassObject` operation

`CoCreateInstance` simply calls `CoGetClassObject` and always requests the `IClassFactory` interface. Once received, it calls `IClassFactory::CreateInstance` and resports the results back to the client.

EXE Activation

COM out-of-process activation starts differently than DLL activation. The CLSID is first searched in a global table that maps a CLSID to its associated class factory. This map is managed by the *DCOMLaunch* service (hosted in a standard *SvcHost.exe*) that is responsible for launching COM EXE servers. The

first time the CLSID is requested, it's unlikely to be present in the CLSID/Class Factory map. In this case, `CoGetClassObject` turns to the Registry and attempts to locate the CLSID at *HKCR\CLSID*. For executable servers, the subkey *LocalServer32* must exist, where the default value of that key is the command line to use to launch the COM server. An example of such a COM class is shown in figure 21-7.

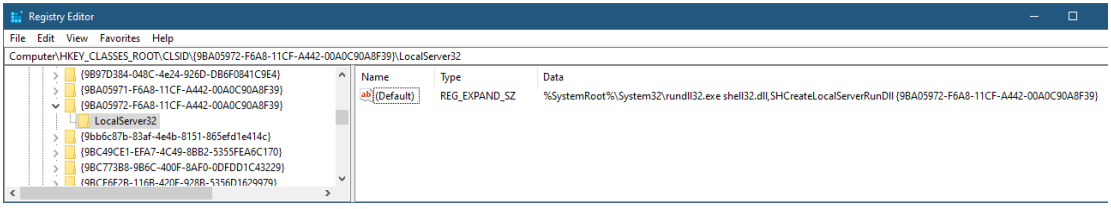


Figure 21-7: COM EXE registration

The above entry is for the *ShellWindows* COM class that allows manipulating *Explorer* windows.

Once located, `CreateProcess` is called by the *DCOMLaunch* service to launch the new process. At the same time, `CoGetClassObject` waits patiently for the appearance of the CLSID in the map managed by *DCOmLaunch* so that it can grab the class factory and return it to the client. To make it work, the newly created process must call the `CoRegisterClassObject` function (one or more times) as soon as possible to register its COM classes with *DCOMLaunch*:

```
HRESULT CoRegisterClassObject(
    _In_ REFCLSID rclsid,           // CLSID
    _In_ LPUNKNOWN pUnk,            // class factory
    _In_ DWORD dwClsContext,        // CLSCTX_ flags
    _In_ DWORD flags,               // REGCLS_ flags
    _Out_ LPDWORD lpdwRegister);    // registration cookie
```

The first parameter to `CoRegisterClassObject` is the CLSID to be registered, while the second parameter is a pointer to the class factory implementation (typically implementing `IClassFactory`, but can be any other interface the server exposes as a factory). `dwClsContext` identifies the context in which the registration is valid. The most common value is `CLSCTX_LOCAL_SERVER` to register the local process as the server.

The `flags` parameter is more interesting, providing customization options for the registration. The possible values are part of the `REGCLS` enumeration, copied here from the header file with some of the included comments (slightly modified):

```
typedef enum tagREGCLS {
    REGCLS_SINGLEUSE      = 0, // class object only generates one instan\
ce
    REGCLS_MULTIPLEUSE    = 1, // same class object genereates multiple \
instances
    REGCLS_MULTI_SEPARATE = 2, // multiple use, but separate control ove\
r each
                               // context
    REGCLS_SUSPENDED      = 4, // register is as suspended, will be acti\
vated
                               // when app calls CoResumeClassObjects
    REGCLS_SURROGATE      = 8, // must be used when a surrogate process
                               // is registering a class object that wil\
l be
                               // loaded in the surrogate
#if (NTDDI_VERSION >= NTDDI_WINTHRESHOLD)   // Windows 10+
    REGCLS_AGILE      = 0x10,  // Class object aggregates the free-threa\
ded
                               // marshaler (FTM) and will be made visib\
le to
                               // all inproc apartments
#endif
} REGCLS;
```

One of the first three values must be specified to indicate how instances are created with the registering class factory. `REGCLS_SINGLEUSE` means that every activation will be served from a different process. This provides robustness but might be wasteful in resources if many instances are created. This also prevents creating system-wide singletons, as each activation requires a new process.

The most common value is `REGCLS_MULTIPLEUSE`, where a single process is used to host all the instances created by clients. `REGCLS_MULTI_SEPARATE` is a combination of single and multiple-use: Out of process clients are treated as multiple-use, but clients creating instances from the server process itself (using this CLSID) get a separate process.

`REGCLS_SUSPENDED` is a flag that can be added to indicate the class factory should be invisible, until a later call to `CoResumeClassObjects`. This is useful if several classes need to be registered and the sever wants to make sure all registrations succeed (and perhaps perform other initialization) before unveiling the classes.

The `REGCLS_SURROGATE` and `REGCLS_AGILE` flags are beyond the scope of this chapter, although we will discuss the *Free-Threaded Marshaler* (FTM) in the section "Threads and Apartments", later in this chapter.

The final parameter to `CoRegisterClassObject` is a value marking this registration, to be passed later to a call to `CoRevokeClassObject`, typically before the server process exits:

```
HRESULT CoRevokeClassObject(_In_ DWORD dwRegister);
```

Once the registration appears in the table maintained by *DCOMLaunch*, the class factory is made available to the client that called `CoGetClassObejct`. Since the real class object is in the server process, the client receives a proxy object to the class factory. From the client's perspective it does not matter. If the original call is by `CoCreateInstance`, then the `IClassFactory` is returned and it calls `IClassFactory::CreateInstance` just like in the DLL server case.

Service Activation

COM activation with a service is no different than a non-service executable. The main difference is in the Registry values. Figure 21-8 shows the registration for the BITS service's *BackgroundCopyManager* (CLSID is {4991d34b-80a1-4291-83b6-3328366b9097}). The CLSID entry just points to something called an application ID (AppID), which are listed in yet another subkey under *HKCR - HKCR\AppID*.

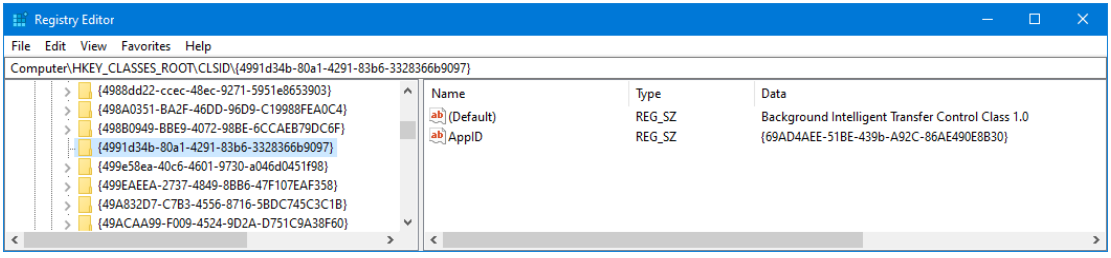


Figure 21-8: COM EXE service registration

Figure 21-9 shows where the AppID points to - the real service name (BITS).

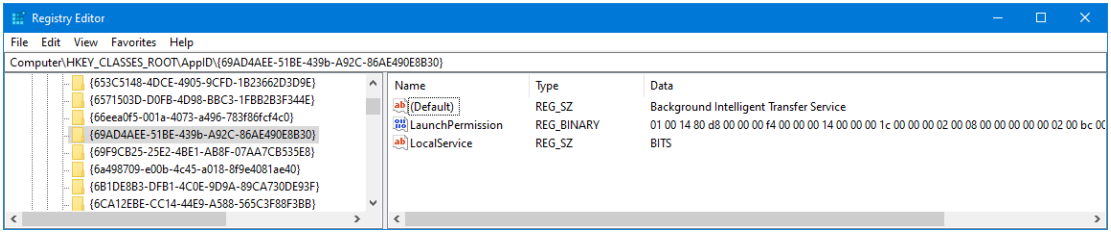


Figure 21-8: COM EXE service registration

In general, the role of the AppID is to provide common properties for multiple COM servers (mostly for out-of-process servers).

In a service-based registration, the *Service Control Manager* (SCM) is contacted to launch the service. The service process should register its class factories just like a non-service executable.

Implementing COM Interfaces

In the previous sections, we saw examples of client code that needs to access some COM objects. We have not yet implemented a COM class. Implementing COM classes is sometimes needed when working with

some Windows APIs. For example, the BITS APIs also support notifications of transfer state changes by having the client implementing a certain interface. The code we used earlier waited in a loop and polled for the status of the transfer by reading the job's state periodically, waiting for the operation to complete or fail. This is generally inefficient, and ties the initiating thread to the request. A better solution would be to have BITS notify the client when the BITS job's state changes asynchronously, allowing the initiating thread to perform other work.

This functionality is exposed through the `IBackgroundCopyJob::SetNotifyInterface` method:

```
virtual HRESULT __stdcall SetNotifyInterface(IUnknown *Val) = 0;
```

The function appears as if it expects any `IUnknown` pointer, but the documentation states that the object must implement the `IBackgroundCopyCallback` interface, defined like so:

```
MIDL_INTERFACE("97ea99c7-0186-4ad4-8df9-c5b4e0ed6b22")
IBackgroundCopyCallback : public IUnknown {
public:
    virtual HRESULT STDMETHODCALLTYPE JobTransferred(
        /* [in] */ __RPC__in_opt IBackgroundCopyJob *pJob) = 0;
    virtual HRESULT STDMETHODCALLTYPE JobError(
        /* [in] */ __RPC__in_opt IBackgroundCopyJob *pJob,
        /* [in] */ __RPC__in_opt IBackgroundCopyError *pError) = 0;
    virtual HRESULT STDMETHODCALLTYPE JobModification(
        /* [in] */ __RPC__in_opt IBackgroundCopyJob *pJob,
        /* [in] */ DWORD dwReserved) = 0;
};
```

This means the client must supply an implementation of `IBackgroundCopyCallback` (including the base `IUnknown` interface). Let's do that by creating a normal C++ class that inherits from `IBackgroundCopyCallback`:

```
struct JobNotifications : IBackgroundCopyCallback {
    HRESULT __stdcall QueryInterface(REFIID riid, void** ppvObject) over\
ride;
    ULONG __stdcall AddRef(void) override;
    ULONG __stdcall Release(void) override;
    HRESULT __stdcall JobTransferred(IBackgroundCopyJob* pJob) override;
    HRESULT __stdcall JobError(IBackgroundCopyJob* pJob,
        IBackgroundCopyError* pError) override;
    HRESULT __stdcall JobModification(IBackgroundCopyJob* pJob,
        DWORD dwReserved) override;
};
```

You can use Visual Studio to get the pure virtual methods to be completed for you, by right-clicking the interface or the class and selecting "Implement pure virtuals for base `IBackgroundCopyCallback`".

The class must implement the 3 `IUnknown` methods, as well as the specific methods of `IBackgroundCopyCallback` that should be called by BITS when changes to the job occur. Let's start with `IUnknown`. A reference count should be defined and managed properly with `AddRef` and `Release` calls. Here is the single member added to the class:

```
struct JobNotifications : IBackgroundCopyCallback {
//...
private:
    ULONG _refCount{ 0 };
};
```

It makes sense to initialize the reference count to zero when the object is initially created, as the first call made by BITS would be to query for some interface that would increment the reference count as the following code demonstrates:

```
ULONG __stdcall JobNotifications::AddRef(void) {
    return ++_refCount;
}

ULONG __stdcall JobNotifications::Release(void) {
    ULONG count = --_refCount;
    if (count == 0)
        delete this;

    return count;
}
```

`AddRef` simply returns the incremented reference count, but `Release` must destroy the object if the reference count drops to zero by calling `delete this`. This, of course, assumes the instance was initially created with the `new` operator. Since we will be implementing and using this class, we can fulfill that promise.

Notice that the code is careful not to touch the object's members if the object is deleted, otherwise an access violation exception is likely to occur. Using something like `return _refCount;` in the `Release` call is an error; returning a local variable is always safe since it's on the stack and the thread is very much alive.

Next, the class needs to implement the `QueryInterface` method by supporting two interfaces: `IUnknown` (mandatory for any COM object), and `IBackgroundCopyCallback`. Here is one simple way to accomplish this:

```
HRESULT __stdcall
JobNotifications::QueryInterface(REFIID riid, void** ppvObject) {
    if (ppvObject == nullptr)
        return E_POINTER;

    if (riid == __uuidof(IUnknown)
        || riid == __uuidof(IBackgroundCopyCallback)) {
        // interface supported
        AddRef();
        *ppvObject = static_cast<IBackgroundCopyCallback*>(this);
        return S_OK;
    }
    return E_NOINTERFACE;
}
```

A few points about the above implementation:

- The first line checks if the output pointer is `NULL`, and if so, returns the standard `E_POINTER` error, indicating a bad pointer was encountered.
- If any of the two interfaces supported is requested, we call `AddRef` to increment the reference count, expecting this object's client (the BITS job) to call `Release` at some point. This is the expected behavior with any `QueryInterface` call.
- The returned interface pointer is the result of a cast performed against

`IBackgroundCopyCallback*`, but `IUnknown*` is just as good. This is because they are part of the same vtable, so the value of the returned pointer is the same regardless of the requested interface. The use of `static_cast` ensures correct returned value in more complex cases, where multiple interfaces are implemented, not necessarily deriving from one another.

Now that we have `IUnknown` out of the way, it's time to implement the interface we really care about - `IBackgroundCopyCallback`. The simplest implementation would be to use console output to indicate what happened:

```
HRESULT __stdcall
JobNotifications::JobTransferred(IBackgroundCopyJob* pJob) {
    PWSTR name;
    pJob->GetDisplayName(&name);
    printf("Job %ws completed successfully!\n", name);
    ::CoTaskMemFree(name);
    pJob->Complete();
    pJob->SetNotifyInterface(nullptr);

    return S_OK;
}
```

```
HRESULT __stdcall JobNotifications::JobError(IBackgroundCopyJob* pJob,
    IBackgroundCopyError* pError) {
    PWSTR name, error, filename = nullptr;
    pJob->GetDisplayName(&name);
    pError->GetErrorDescription(
        LANGIDFROMLCID(::GetThreadLocale()), &error);
    CComPtr<IBackgroundCopyFile> spFile;
    pError->GetFile(&spFile);
    if (spFile)
        spFile->GetRemoteName(&filename);

    printf("Job %ws failed: %ws (file: %ws)\n", name, error,
        filename ? filename : L"(unknown)");
    ::CoTaskMemFree(name);
    ::CoTaskMemFree(error);
    if (filename)
        ::CoTaskMemFree(filename);

    pJob->SetNotifyInterface(nullptr);
    return S_OK;
}

HRESULT __stdcall JobNotifications::JobModification(
    IBackgroundCopyJob* pJob, DWORD /* dwReserved */) {
    BG_JOB_STATE state;
    pJob->GetState(&state);
    PWSTR name;
    pJob->GetDisplayName(&name);
    printf("Job %ws changed to state: %s (%u)\n", name,
        JobStateToString(state), state);
    ::CoTaskMemFree(name);
    if(state == BG_JOB_STATE_CANCELLED)
        pJob->SetNotifyInterface(nullptr);

    return S_OK;
}
```

The implementation queries various properties of the job passed in as an argument to all interface methods, such as its display name and state. Notice the calls to `CoTaskMemFree` to free strings returned by `IBackgroundCopyJob` methods. The documentation states that this function should be used to free the allocated memory. `CoTaskMemFree` (and its associated allocation function `CoTaskMemAlloc`) are wrappers around the *COM Task Memory Allocator*, whose significance is discussed in the section "Proxies

and Stubs", later in this chapter. For now, we can treat these functions as yet another way to allocate and free memory.

Once the operation completes (whether successfully, with an error or cancellation), `IBackgroundCopyJob::SetNotifyInterface` is called with a `NULL` pointer, indicating notifications are no longer needed. This should cause the job to release its interface pointer to our notification object, causing its reference count to drop to zero.

You can verify this and the other methods used by setting breakpoints and debugging.

`JobStateToString` is a simple function that converts the state (`BG_JOB_STATE` to a string). The source code is part of the *BITSDemo* project.

With the above implementation in place, we can create a function to test it like so:

```
HRESULT DoBitsWorkWithNotify(PCWSTR jobName, PCWSTR remoteUrl,
    PCWSTR localFile) {
    // assume CoInitializeEx has already been called
    CComPtr<IBackgroundCopyManager> spMgr;
    HRESULT hr = spMgr.CoCreateInstance(
        __uuidof(BackgroundCopyManager));
    if (FAILED(hr))
        return hr;

    CComPtr<IBackgroundCopyJob> spJob;
    GUID guid;
    hr = spMgr->CreateJob(jobName, BG_JOB_TYPE_DOWNLOAD, &guid, &spJob);
    if (FAILED(hr))
        return hr;

    hr = spJob->AddFile(remoteUrl, localFile);
    if (FAILED(hr))
        return hr;

    spJob->SetNotifyFlags(BG_NOTIFY_JOB_TRANSFERRED
        | BG_NOTIFY_JOB_MODIFICATION
        | BG_NOTIFY_FILE_RANGES_TRANSFERRED
        | BG_NOTIFY_JOB_ERROR);
    auto notify = new JobNotifications;
```

```
    hr = spJob->SetNotifyInterface(notify);
    if (FAILED(hr))
        return hr;

    return spJob->Resume();
}
```

The first part is identical to the way a BITS job was created and initialized in the previous demos. The new code includes calling `IBackgroundCopyJob::SetNotifyFlags` to let the job know what notifications we're interested in. By default, only completion and error are called, so the above code requests all possible notifications.

Next, an instance of the `JobNotifications` class we implemented as the target of notifications, and passing it in the call to `IBackgroundCopyJob::SetNotifyInterface`. Lastly, `IBackgroundCopyJob::Resume` is invoked to start the transfer.

Here is a `main` function example that uses the above function for an image to be downloaded:

```
int main() {
    ::CoInitializeEx(nullptr, COINIT_MULTITHREADED);

    DoBitsWorkWithNotify(L"My Job",
        L"https://www.fnordware.com/superpng/pnggrad16rgb.png",
        L"c:\\temp\\image2.png");

    // just for demonstration purposes
    ::Sleep(INFINITE);

    ::CoUninitialize();
    return 0;
}
```

After initiating the transfer, `main` calls `Sleep(INFINITE)` to prove that the notifications work regardless of what the initiating thread is doing. Running this code prints out the following on the console:

```
Job My Job changed to state: Connecting (1)
Job My Job changed to state: Transferring (2)
Job My Job changed to state: Transferred (6)
Job My Job completed successfully!
```

Note the call to `CoInitializeEx` uses `COINIT_MULTITHREADED`. This is required in this example. If you use `COINIT_APARTMENTTHREADED` or just `CoInitialize` (which are equivalent), you'll find that the application deadlocks and no notifications are received. The reason for this behavior is subtle, and is discussed in the section "Threads and Apartments", later in this chapter.

You can initiate multiple requests from the same thread and they will work flawlessly. Here is another `main` example:

```
int main() {
    ::CoInitializeEx(nullptr, COINIT_MULTITHREADED);

    DoBitsWorkWithNotify(L"My first Job",
        L"http://speedtest.ftp.otenet.gr/files/test10Mb.db",
        L"c:\\temp\\test1.db");
    DoBitsWorkWithNotify(L"My second Job",
        L"http://speedtest.ftp.otenet.gr/files/test1Mb.db",
        L"c:\\temp\\test2.db");

    // just for demonstration purposes
    ::Sleep(INFINITE);

    ::CoUninitialize();
    return 0;
}
```

Here is the output when run:

```
Job My first Job changed to state: Connecting (1)
Job My second Job changed to state: Queued (0)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job completed successfully!
Job My first Job changed to state: Transferred (6)
Job My second Job changed to state: Connecting (1)
```

```
Job My second Job changed to state: Transferring (2)
Job My second Job completed successfully!
Job My second Job changed to state: Transferred (6)
```

The call to `Sleep` in the `main` function is definitely not elegant, as we get no notification when all the transfers complete in one way or another. We can use event objects, and have the notification callbacks signal these events when an operation completes. Also, notice that currently each transfer creates its own notification object, but we could change that if we wanted to a single instance sharing notifications from multiple jobs.

Make the necessary changes so that a single notification object is shared by all initiated jobs.

We'll make a change where an operation accepts an additional (optional) event handle parameter, and signals that handle when the operation completes. The handle should be passed to the notification object, as it handles all notifications. To that end, we'll add an event member to the `JobNotifications` class and initialize it in the constructor:

```
// JobNotifications.h

struct JobNotifications : IBackgroundCopyCallback {
    explicit JobNotifications(HANDLE hEvent = nullptr);
//...
private:
    ULONG _refCount{ 0 };
    HANDLE _hEvent;
};

// JobNotifications.cpp

JobNotifications::JobNotifications(HANDLE hEvent)
    : _hEvent(hEvent) {}
```

Every method in `JobNotifications` that is notified of a final state (success, error, cancellation) needs to signal the event. Here is the extra code:

```
HRESULT __stdcall JobNotifications::JobTransferred(
    IBackgroundCopyJob* pJob) {
    // code unchanged, and then...
    if (_hEvent)
        ::SetEvent(_hEvent);

    return S_OK;
}

HRESULT __stdcall JobNotifications::JobError(IBackgroundCopyJob* pJob,
    IBackgroundCopyError* pError) {
    // code unchanged, and then...
    if (_hEvent)
        ::SetEvent(_hEvent);
    return S_OK;
}

HRESULT __stdcall
JobNotifications::JobModification(IBackgroundCopyJob* pJob, DWORD) {
    // code unchanged, and then...
    if (state == BG_JOB_STATE_CANCELLED) {
        pJob->SetNotifyInterface(nullptr);
        if (_hEvent)
            ::SetEvent(_hEvent);
    }
    return S_OK;
}
```

The code initiating the request accepts an optional handle and passes that along to the `JobNotifications` constructor:

```
HRESULT DoBitsWorkWithNotify(PCWSTR jobName, PCWSTR remoteUrl,
    PCWSTR localFile, HANDLE hEvent = nullptr) {
    // no code changes, until...
    auto notify = new JobNotifications(hEvent);
    hr = spJob->SetNotifyInterface(notify);
    if (FAILED(hr))
        return hr;

    return spJob->Resume();
}
```

Here is an example `main` function that uses two events for two operations:

```
int main() {
    ::CoInitializeEx(nullptr, COINIT_MULTITHREADED);

    HANDLE hEvent[] = {
        ::CreateEvent(nullptr, FALSE, FALSE, nullptr),
        ::CreateEvent(nullptr, FALSE, FALSE, nullptr),
    };
    DoBitsWorkWithNotify(L"My first Job",
        L"http://speedtest.ftp.otenet.gr/files/test10Mb.db",
        L"c:\\temp\\test1.db",
        hEvent[0]);
    DoBitsWorkWithNotify(L"My second Job",
        L"http://speedtest.ftp.otenet.gr/files/test1Mb.db",
        L"c:\\temp\\test2.db",
        hEvent[1]);

    ::WaitForMultipleObjects(_countof(hEvent), hEvent,
        TRUE, INFINITE);
    printf("All jobs completed!\n");

    ::CloseHandle(hEvent[0]);
    ::CloseHandle(hEvent[1]);

    ::CoUninitialize();
    return 0;
}
```

Here is an example run:

```
Job My first Job changed to state: Connecting (1)
Job My second Job changed to state: Queued (0)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job changed to state: Transferring (2)
Job My first Job completed successfully!
Job My first Job changed to state: Transferred (6)
Job My second Job changed to state: Connecting (1)
Job My second Job changed to state: Transferring (2)
Job My second Job completed successfully!
```

```
Job My second Job changed to state: Transferred (6)
All jobs completed!
```

COM Servers

In the previous section we implemented a COM class that was used to get notifications from a BITS job object. The class was "local" in the sense that it was not registered in the Windows Registry and could not be created externally as it had no CLSID and no supporting infrastructure. In this section, we'll see what it takes to create an independent COM server that can be activated from any client.

To demonstrate creating a simple, yet functional, COM component, we will complete the RPN Calculator example. The first thing to decide is whether we want an in-process component (DLL) or an out-of-process component (EXE). Since it's easier to understand how a COM DLL works, we'll build the component as a DLL.

The first step would be to create a standard Win32 DLL project using Visual Studio. This is no different than DLL projects created in chapter 15. I'll name the project *RPNCalcSvr*. I'll delete the *framework.h* file created by the wizard (if your Visual Studio version creates such a file), and put a single `#include <Windows.h>` in the precompiled header file (*pch.h*). The only other interesting file is *dllmain.cpp* where a barebones `DllMain` function is implemented.

The first order of business is to create a header file that would be common to the server (this project) and other C++ clients. This header should contain abstract definitions only - interfaces - never any implementation. This file should probably also include GUIDs the client needs to know about, such as the interface ID of `IRPNCalculator`.

I will name the file *RPNCalcInterfaces.h* (any reasonable name will do), and put the `IRPNCalculator` interface definition in that file. Then we can generate a GUID for the interface (since we're just defining it now and it's not yet known to clients) by running the *Create GUID* tool and copying the result from format number 2, and adding a proper name. Here is the final file (your generated GUID will be different):

```
// RPNCalcInterfaces.h

#pragma once

struct IRPNCalculator : IUnknown {
    virtual HRESULT __stdcall Push(double value) = 0;
    virtual HRESULT __stdcall Pop(double* value) = 0;
    virtual HRESULT __stdcall Add() = 0;
    virtual HRESULT __stdcall Subtract() = 0;
};

// {7020B8D9-CEFE-46DC-89D7-F0261C3CDE66}
DEFINE_GUID(IID_IRPNCalculator,
    0x7020b8d9, 0xcefe, 0x46dc, \
        0x89, 0xd7, 0xf0, 0x26, 0x1c, 0x3c, 0xde, 0x66);
```

The `DEFINE_GUID` macro builds the data structure instance to hold the 128-bit GUID and assigns it the given name. Technically, `DEFINE_GUID` is somewhat trickier than that, and its exact operation depends on another macro, `INIT_GUID`. If `INIT_GUID` is defined, the structure instance is defined and instantiated; otherwise, it's just declared as `extern`. This helps prevent an "unresolved external" linker error on the one hand and "multiple symbol definition" linker error, because the same file is to be used by the server and the client (as we'll see later). Another nice option would be to associate the interface GUID with the interface definition itself, so that the `__uuidof` operator can be used:

```
struct __declspec(uuid("7020B8D9-CEFE-46DC-89D7-F0261C3CDE66"))
    IRPNCalculator : IUnknown {
```

The macro `MIDL_INTERFACE` can be used as a shortcut for `__declspec(uuid(...))`.

Implementing the COM Class

The next step is to create a new class named `RPNCalculator` that implements the `IRPNCalculator` interface. We'll add any required members needed for the implementation - a reference counter and a stack of numbers:

```
// RPNCalculator.h
#include <stack>
#include "RPNCalcInterfaces.h"

class RPNCalculator : public IRPNCalculator {
public:
    // from IUnknown
    HRESULT __stdcall QueryInterface(REFIID riid,
        void** ppvObject) override;
    ULONG __stdcall AddRef(void) override;
    ULONG __stdcall Release(void) override;
    // from IRPNCalculator
    HRESULT __stdcall Push(double value) override;
    HRESULT __stdcall Pop(double* value) override;
    HRESULT __stdcall Add() override;
    HRESULT __stdcall Subtract() override;

private:
    ULONG _refCount{ 1 };
    std::stack<double> _data;
};
```

The reference counter is initialized to one, contrary to the `JobNotifications` class from the previous section. We'll see later when implementing the class factory why this is the case.
Implementing the `IUnknown` methods is similar to the `JobNotifications` class:

```
HRESULT __stdcall
RPNCalculator::QueryInterface(REFIID riid, void** ppvObject) {
    if(ppvObject == nullptr)
        return E_POINTER;

    if (riid == __uuidof(IUnknown)
        || riid == __uuidof(IRPNCalculator)) {
        AddRef();
        *ppvObject = static_cast<IRPNCalculator*>(this);
        return S_OK;
    }
    return E_NOINTERFACE;
}

ULONG __stdcall RPNCalculator::AddRef(void) {
    return ++_refCount;
}

ULONG __stdcall RPNCalculator::Release(void) {
    auto count = --_refCount;
    if (count == 0)
        delete this;
    return count;
}
```

Next, we implement the "real" stuff we want to expose from the class. It consists of managing the stack correctly, and performing the mathematical operations:

```
HRESULT __stdcall RPNCalculator::Push(double value) {
    // limit the stack size
    if (_data.size() > 256)
        return E_UNEXPECTED;

    _data.push(value);
    return S_OK;
}

HRESULT __stdcall RPNCalculator::Pop(double* value) {
    if (_data.empty())
```

```
        return E_UNEXPECTED;

    *value = _data.top();
    _data.pop();

    return S_OK;
}

HRESULT __stdcall RPNCalculator::Add() {
    double x, y;
    auto hr = Pop(&x);
    if (FAILED(hr))
        return hr;

    hr = Pop(&y);
    if (FAILED(hr))
        return hr;

    Push(x + y);
    return S_OK;
}

HRESULT __stdcall RPNCalculator::Subtract() {
    double x, y;
    auto hr = Pop(&x);
    if (FAILED(hr))
        return hr;

    hr = Pop(&y);
    if (FAILED(hr))
        return hr;

    Push(y - x);
    return S_OK;
}
```

The implementation details are not important as far as COM is concerned, but they are important for correct functionality. At this point, we have a class implementation, but COM requires instances to be created by a class factory. Since the class factory is just another COM class, we'll create another class named `RPNCalculatorFactory` and implement the most common class factory interface - `IClassFactory`.

Implementing the Class Object (Factory)

Here is the header file declaration for the class factory:

```
// RPNCalculatorFactory.h

struct RPNCalculatorFactory : IClassFactory {
    // methods from IUnknown
    HRESULT __stdcall QueryInterface(REFIID riid, void** ppvObject) over\
ride;
    ULONG __stdcall AddRef(void) override;
    ULONG __stdcall Release(void) override;
    // methods from IClassFactory
    HRESULT __stdcall CreateInstance(IUnknown* pUnkOuter,
        REFIID riid, void** ppvObject) override;
    HRESULT __stdcall LockServer(BOOL fLock) override;
    // no ref count !?
};
```

The `virtual` keyword is not required when overriding virtual functions from a base class.

The implementation of the `IUnknown` methods is similar to previous implementations with a twist:

```
HRESULT __stdcall
RPNCalculatorFactory::QueryInterface(REFIID riid, void** ppv) {
    if (ppv == nullptr)
        return E_POINTER;

    if (riid == __uuidof(IUnknown)
        || riid == __uuidof(IClassFactory)) {
        *ppv = static_cast<IClassFactory*>(this);
        return S_OK;
    }
    return E_NOINTERFACE;
}

ULONG __stdcall RPNCalculatorFactory::AddRef(void) {
    return 2;
}

ULONG __stdcall RPNCalculatorFactory::Release(void) {
    return 1;
}
```

The `AddRef` and `Release` methods do not manage any reference count and just return arbitrary non-zero numbers. This is because we will create the class factory as a static object (rather than being allocated dynamically), as a single class factory is able to create any number of object instances.

> The above approach is common with class factories, but certainly not the only one. It's perfectly fine to manage a class factory just like other COM objects with a proper reference count. Note that COM doesn't care either way, since COM is concerned with interfaces rather than the implementation. A COM object's `AddRef` and `Release` methods must return non-zero values if the object is still alive, but they do not have to return the real reference count.

Now comes the most important method in `IClassFactory`: `CreateInstance`. Here is one way to implement it:

```
HRESULT __stdcall
RPNCalculatorFactory::CreateInstance(IUnknown* pUnkOuter,
    REFIID riid, void** ppvObject) {
    if (pUnkOuter)
        return CLASS_E_NOAGGREGATION;

    auto calc = new RPNCalculator;
    auto hr = calc->QueryInterface(riid, ppvObject);
    calc->Release();

    return hr;
}
```

`IClassFactory::CreateInstance` accepts three parameters. The first is an optional `IUnknown` pointer to the *Controlling IUnknown*, in case the object is to be aggregated by the caller. We'll look at the basics of aggregation in the section "Threads and Apartments", later in this chapter. For now, we do not support aggregation, returning failure if the controlling `IUnknown` is non-`NULL`.

The next two parameters are the interface ID requested for the new object and the address of a pointer where the result should be, if successful. The requested interface could be `IUnknown` (which must always be supported), or some custom interface the client expects to be implemented.

The object is created with the `new` operator and its reference count initialized to one (see the header code in the previous section). Next, `QueryInterface` is called on the object to obtain the interface requested by the client. If that succeeds, `S_OK` is returned and the internal reference count is incremented to 2. If it fails, `E_NOINTERFACE` is returned, and the reference count is unchanged (1).
The next call is to `Release` - if `QueryInterface` succeeded, the reference count drops to 1 as it should before returning to the client. Otherwise, the reference count drops to zero and the object is deleted by the `Release` call.

The second method of `IClassFactory` - `LockServer` is used to keep the server alive even if no objects created by the server exist. A COM server can maintain a reference count, but a simple implementation can just return a not implemented status:

```
HRESULT __stdcall RPNCalculatorFactory::LockServer(BOOL fLock) {
    return E_NOTIMPL;
}
```

Implementing `DllGetClassObject`

Now that we have a class factory implementation, it's time to tie it to the `DllGetClassObject` function that must be implemented by any COM DLL (and exported). Refer to figure 21-6 for a high-level overview of the DLL activation process.

At this point, we have no CLSID for the `RPNCalculator` class, so let's generate one with the *Create GUID* tool and paste its definition into the *RPNCalcInterfaces.h* header file, where all the other abstractions are stored:

```
// {FA523D4E-DB35-4D0B-BD0A-002281FE3F31}
DEFINE_GUID(CLSID_RPNCalculator,
    0xfa523d4e, 0xdb35, 0x4d0b, \
        0xbd, 0xa, 0x0, 0x22, 0x81, 0xfe, 0x3f, 0x31);
```

The traditional prefix from GUIDs representing classes is `CLSID_`. We can optionally add a declaration that would allow usage of the `__uuidof` operator for the class:

```
class __declspec(uuid("FA523D4E-DB35-4D0B-BD0A-002281FE3F31"))
RPNCalculator;
```

Now we can implement `DllGetClassObject` (in *dllmain.cpp*):

```
#include "RPNCalculatorFactory.h"
#include "RPNCalcInterfaces.h"

STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, void** ppv) {
    if (rclsid == __uuidof(RPNCalculator)) {
        static RPNCalculatorFactory factory;
        return factory.QueryInterface(riid, ppv);
    }
    return CLASS_E_CLASSNOTAVAILABLE;
}
```

The `STDAPI` macro is just a shorthand for `extern "C" HRESULT __stdcall`, meaning the function must be exported as a C function (rather than C++) with the standard calling convention. (Refer to chapter 15 for more on DLLs and exports).

This is where the class factory is created as a singleton object, so the implementation of `Release` does not attempt to delete the object. If the requested class is the one we implemented, then the factory is queried for the requested interface (hopefully `IClassFactory`). Otherwise, the standard error `CLASS_E_CLASSNOTAVAILABLE` is returned, indicating the particular CLSID is not available in this COM server.

Next, we'll add a module definition file (*RPNCalcSvr.def*) and include the currently single exported function:

```
LIBRARY
EXPORTS
    DllGetClassObject PRIVATE
```

The `PRIVATE` directive indicates the export will be invisible in the generated LIB file (if generated). This is to enforce the notion that COM DLLs are always loaded dynamically, and never linked against statically.

Implementing Self Registration

Our current COM DLL implementation is complete, as far as functionality is concerned. Still, clients that call `CoCreateInstance` or `CoGetClassObject` will fail to locate the DLL. This is because the CLSID is not registered in the Registry where `CoGetClassObject` looks by default.

If you have worked with COM before, you might be familiar with the *RegSvr32.exe* tool provided by Windows that magically registers a COM DLL. This tool is not magical, of course, and expects the DLL to register itself. For this purpose, COM DLLs must implement and export two functions: `DllRegisterServer` and `DllUnregisterServer`. *RegSvr32* simply calls `LoadLibrary`, `GetProcAddress` and then `DllRegisterServer` to "perform" the registration. If `S_OK` is returned, it announces that registration has succeeded.

We'll implement the registration and unregistration functions by using the familiar Registry APIs from chapter 17. To make these operations more robust, we'll use a transaction object to wrap the Registry operations, so partial successful operations will be rolled back instead of leaving partial changes.

Before we get to the actual registration, we're going to need the DLL instance handle (its address) to retrieve its full path name. For that, we can extract the value from `DllMain`. Here is the updated `DllMain`:

```
HMODULE g_hInstDll;

BOOL WINAPI DllMain(HMODULE hModule, DWORD reason, PVOID reserved) {
    switch (reason) {
        case DLL_PROCESS_ATTACH:
            g_hInstDll = hModule;
            ::DisableThreadLibraryCalls(hModule);
            break;
```

```
        case DLL_PROCESS_DETACH:
            break;
    }
    return TRUE;
}
```

The global `g_hInstDll` is initialized once the DLL is loaded into a process. The call to `DisableThreadLibraryCalls` is an optimization that notifies the loader that thread creation and destruction should not cause `DllMain` to be called for this DLL. We'll also add a constant string with the relative CLSID path for the `RPNCalculator` class:

```
static const WCHAR clsidPath[] =
    L"CLSID\\{FA523D4E-DB35-4D0B-BD0A-002281FE3F31}";
```

Now we can start coding `DllRegisterServer` by creating a simple transaction:

```
STDAPI DllRegisterServer() {
    HANDLE hTransaction = ::CreateTransaction(nullptr, nullptr,
        TRANSACTION_DO_NOT_PROMOTE, 0, 0, INFINITE, nullptr);
    if(hTransaction == INVALID_HANDLE_VALUE)
        return HRESULT_FROM_WIN32(::GetLastError());
```

If the transaction creation fails, the returned value should reflect that, but it must be an `HRESULT`. The `HRESULT_FROM_WIN32` is a handy macro that converts a Windows error code to an `HRESULT`.

`HRESULT_FROM_WIN32` sets the `HRESULT` *facility* to `FACILITY_WIN32` and tucks the error code in the *code* part of the `HRESULT`.

Next, we need to create the key identified by `clsidPath` variable above and write a simple string for our class, such as "RPNCalculator". This is not strickly necessary, as `CoGetClassObject` will not be looking for that string, but it is easier for identification purposes by humans. Furthermore, some tools, such as the *OLE/COM Object Viewer* (*oleview.exe*), show this string in their output, making it easily recognizeable.

```
HKEY hKey;
auto error = ::RegCreateKeyTransacted(HKEY_CLASSES_ROOT, clsidPath,
    0, nullptr, REG_OPTION_NON_VOLATILE, KEY_WRITE,
    nullptr, &hKey, nullptr, hTransaction, nullptr);
if (error != ERROR_SUCCESS)
    return HRESULT_FROM_WIN32(error);

WCHAR value[] = L"RPNCalculator";
error = ::RegSetValueEx(hKey, L"", 0, REG_SZ,
    (const BYTE*)value, sizeof(value));
if (error != ERROR_SUCCESS)
    return HRESULT_FROM_WIN32(error);
```

`RegCreateKeyTransacted` is used to create the key (or open it if it exists) while attaching the call to the transaction.

The next step is to write the important registration part, which is the "InProcServer32" subkey pointing to the location of the DLL:

```
HKEY hInProcKey;
error = ::RegCreateKeyTransacted(hKey, L"InProcServer32", 0, nullptr,
    REG_OPTION_NON_VOLATILE, KEY_WRITE, nullptr, &hInProcKey, nullptr,
    hTransaction, nullptr);
if (error != ERROR_SUCCESS)
    return HRESULT_FROM_WIN32(error);

WCHAR path[MAX_PATH];
::GetModuleFileName(g_hInstDll, path, _countof(path));
error = ::RegSetValueEx(hInProcKey, L"", 0, REG_SZ, (const BYTE*)path,
    DWORD(::wcslen(path) + 1) * sizeof(WCHAR));
if (error != ERROR_SUCCESS)
    return HRESULT_FROM_WIN32(error);
```

The subkey is created/opened relative to the parent key and attached to the same transaction. Then, the DLL's path retrieved dynamically with `GetModuleFileName`, because we don't want to hard code a path here in case the DLL is later moved to some other location and registered from there.

Finally, we need to commit the transaction and close all handles for good measure:

```
    auto ok = ::CommitTransaction(hTransaction);
    auto hr = ok ? S_OK : HRESULT_FROM_WIN32(::GetLastError());

    ::RegCloseKey(hInProcKey);
    ::RegCloseKey(hKey);
    ::CloseHandle(hTransaction);

    return hr;
}
```

Unregistering is simpler, since we just need to delete the specific class Registry key and all its children:

```
STDAPI DllUnregisterServer() {
    HANDLE hTransaction = ::CreateTransaction(nullptr, nullptr,
        TRANSACTION_DO_NOT_PROMOTE, 0, 0, INFINITE, nullptr);
    if (hTransaction == INVALID_HANDLE_VALUE)
        return HRESULT_FROM_WIN32(::GetLastError());

    HKEY hKey;
    auto error = ::RegOpenKeyTransacted(HKEY_CLASSES_ROOT,
        clsidPath, 0,
        DELETE | KEY_ENUMERATE_SUB_KEYS
        | KEY_QUERY_VALUE | KEY_SET_VALUE,
        &hKey, hTransaction, nullptr);
    if (error != ERROR_SUCCESS)
        return HRESULT_FROM_WIN32(error);

    // delete tree (deletes all children)
    error = ::RegDeleteTree(hKey, nullptr);
    if (error != ERROR_SUCCESS)
        return HRESULT_FROM_WIN32(error);

    // delete the key itself
    error = ::RegDeleteKeyTransacted(HKEY_CLASSES_ROOT,
        clsidPath, 0, 0,
        hTransaction, nullptr);
    if (error != ERROR_SUCCESS)
        return HRESULT_FROM_WIN32(error);

    BOOL ok = ::CommitTransaction(hTransaction);
    auto hr = ok ? S_OK : HRESULT_FROM_WIN32(::GetLastError());

    ::RegCloseKey(hKey);
```

```
    ::CloseHandle(hTransaction);

    return hr;
}
```

Registering the Server

The registration functions perform their work under `HKEY_CLASSES_ROOT`, which is an aggregation of two Registry keys:

- *HKEY_LOCAL_MACHINE\Software\Classes*
- *HKEY_CURRENT_USER\Software\Classes*

The default behavior when writing to *HKEY_CLASSES_ROOT* is to modify the local machine hive, which means that such registration can only succeed if run by an elevated process. On the other hand, registering in `HKEY_CURRENT_USER` can be done from a process running with standard user rights, and takes precedence over machine wide registration.

To register the server, *RegSvr32* is invoked with the DLL full or relative path, for example:

```
c:\RPNCalcSvr\>regsvr32 RPNCalcSvr.dll
```

Unregistering is equally easy, by adding the `/u` command line switch:

```
c:\RPNCalcSvr\>regsvr32 /u RPNCalcSvr.dll
```

To provide more flexibility for registration, *Regsvr32* supports an additional `/i` switch that calls another export, `DllInstall` that must have the following prototype:

```
STDAPI DllInstall(BOOL install, _In_opt_ PCWSTR commandLine);
```

This export allows more flexibility as the "commandLine" parameter can be any string that makes sense to the DLL. For example, the string "user" can be used to tell the DLL to perform the registration to the user's hive rather than the machine's. Without a `/n` switch, the standard function is called first (e.g. `DllRegisterServer`), and if success is returned, `DllInstall` is called. If the `/n` switch is added, only `DllInstall` is called.

Here is an example for using *RegSvr32* and `DllInstall` only for registering and unregistering (assuming the DLL is in the working directory):

```
c:\RPNCalcSvr\>regsvr32 /n /i:user RPNCalcSvr.dll
```

```
c:\RPNCalcSvr\>regsvr32 /n /u /i:user RPNCalcSvr.dll
```

The first command calls `DllInstall(TRUE, L"user")` and the second calls `DllInstall(FALSE, L"user")`.

Make the necessary changes so that the registration can be done under the *HKEY_LOCAL_MACHINE* or *HKEY_CURRENT_USER* based on some value to `DllInstall`.

Debugging Registration

Debugging registration code is fairly easy to do by setting *RegSvr32* as the executable to launch and the DLL path as the command line argument. Adding `/u` will test unregistration. Figure 21-10 shows what the debug settings look like in Visual Studio. Note that *RegSvr32* is one of the suggested values if you click the dropdown button in the *Command* entry.

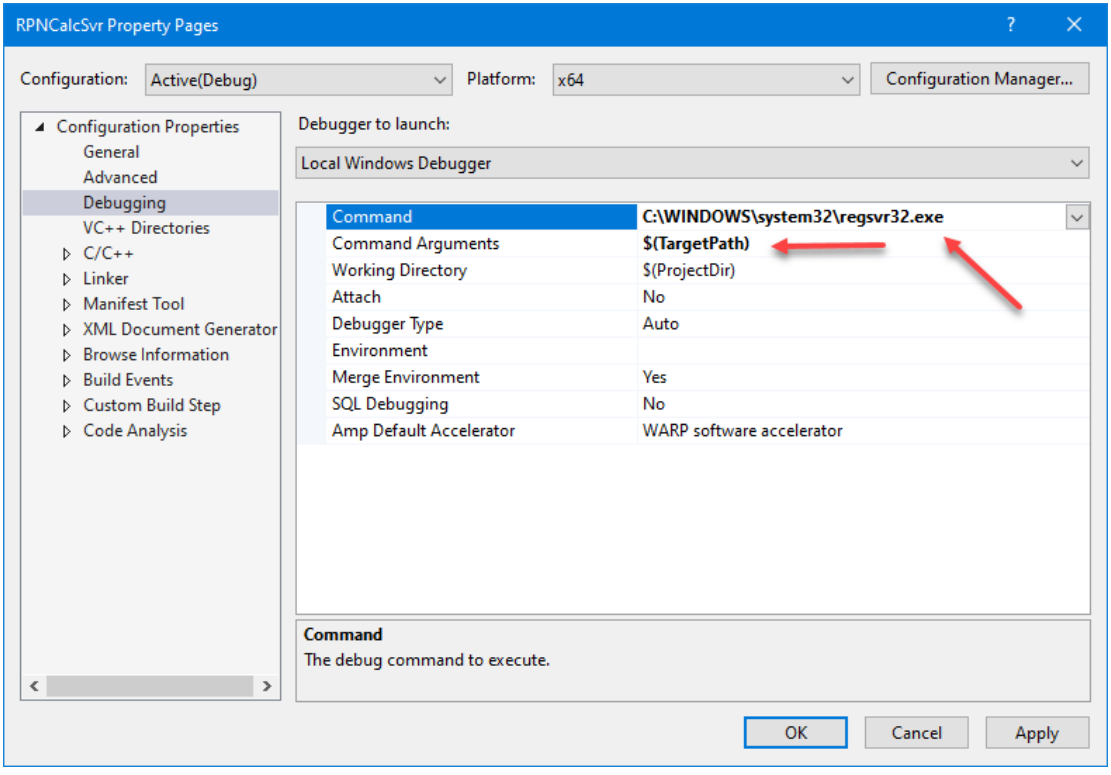


Figure 21-10: Debugging registration with Visual Studio

There is one snag, however. If you build the project for 32-bit (Platform shows Win32), your breakpoints won't hit. The reason is that the 64-bit *Regsvr32.exe* is launched, and it finds that the DLL is 32-bit, so launches its 32-bit version from the *SysWow64* directory, so a new child process is the one performing the loading of the DLL and calling your function(s). There are two ways to handle this:

- Change the *command* value to invoke the 32-bit version of *RegSvr32*

(*c:\Windows\SysWow64\RegSvr32.exe*).

- Use an extension to Visual Studio that allows child process debugging, and enable it (can be downloaded using the standard *Extensions* dialog in Visual Studio).

The first option is obviously easier, but it's good to know the second option exists, not just for this case, but more generally if child processes debugging is needed.

Testing the Server

Assuming the server has been registered successfully, it's time to test it. You can make sure registration succeeded by navigating to the correct location in the Registry and verifying. Add a new project to the existing solution named *Calculator* as a C++ Console application.

The following code snippet shows how to create an instance and invoke some methods:

```
#include <Windows.h>
#include <stdio.h>
#include <atlcomcli.h>
#include "..\RPNCalcSvr\RPNCalcInterfaces.h"

int main() {
    ::CoInitialize(nullptr);

    CComPtr<IRPNCalculator> spCalc;
    auto hr = spCalc.CoCreateInstance(__uuidof(RPNCalculator));
    if (SUCCEEDED(hr)) {
        spCalc->Push(10);
        spCalc->Push(6);
        spCalc->Push(12);
        spCalc->Add();
        double result;
        spCalc->Pop(&result);
        printf("Value popped: %lf\n", result);
        spCalc->Push(result);
        spCalc->Subtract();
        spCalc->Pop(&result);
        printf("Value popped: %lf\n", result);
    }

    ::CoUninitialize();
```

```
    return 0;
}
```

You can set breakpoints, step into method calls (such as `Push` or `Add`). The calls are just virtual method invocations once the object is created. COM is no longer involved - you can verify that by looking at the call stack (figure 21-11).

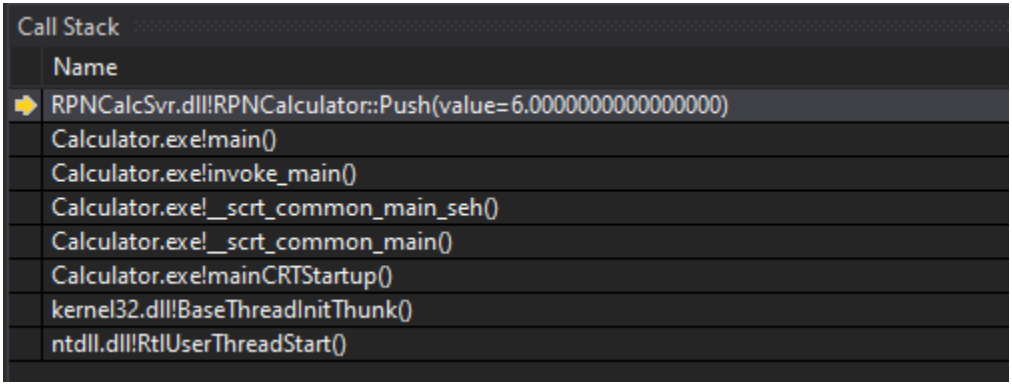


Figure 21-11: Call stack while inside a method

If you run the above code with a debugger and let it run to the end, you'll find that you get an access violation exception (figure 21-12). Can you figure out why given the code above?

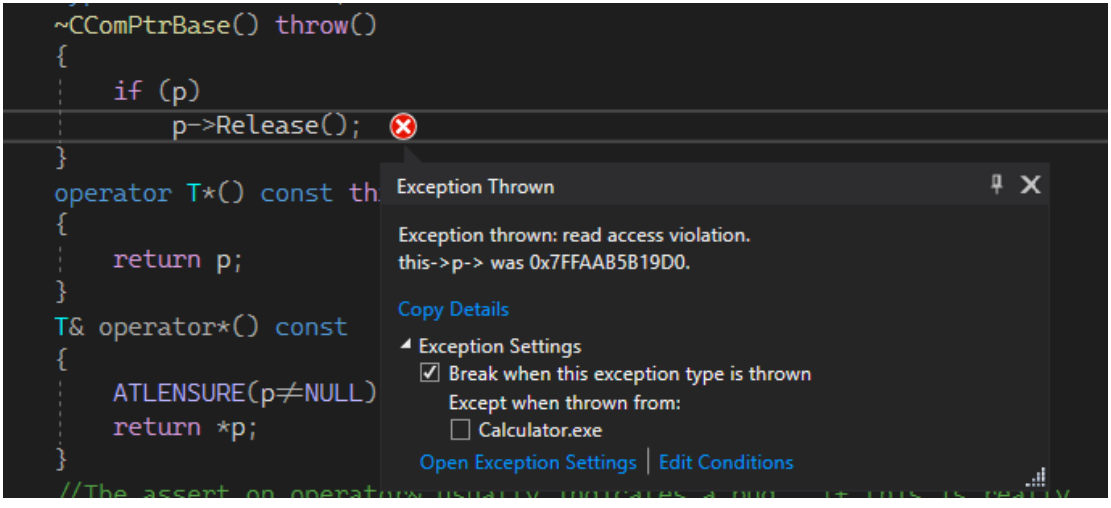


Figure 21-12: Access violation exception

The problem is that `CoUninitialize` is called before the smart pointer destructor. Since the thread no longer needs to use COM, the server DLL (*RPNCalcSvr.dll*) is unloaded from the process address space as there are no other threads using it. Then the destructor of the smart pointer is called, and the `Release` call caused an access violation exception because the code is no longer in memory.

This is not normally a problem in real applications, where COM work is done in functions other than

`main`. Here are a few ways to solve this:

- Call `Release` explicitly on the smart pointer (`spCalc.Release();`) or

`spCalc = nullptr;` before exiting the scope.

- Add an artificial scope from the after ‘CoInitialize `to just before` CoUninitialize‘.
- Put the entire code except the calls to `CoInitialize` / `CoUninitialize` in another function.
- Remove the `CoUninitialize` call altogether. This works in this case, but not a good idea in general.

Testing with non C/C++ Client

COM is a binary standard, so we should be able to use the same functionality from a non-C++ client. We’ll take C# and the .NET Framework as an example. In C++, we used an `#include` to get the interface definition and GUIDs, but how can we do that in C#? The C# compiler does not understand C/C++ header files.

To solve this, we’ll provide the definitions to the C# compiler in C#. In the section “IDL and Type Libraries”, we’ll use another way to more easily get those definitions to .NET clients.

> If you don’t care about C# much, you can safely skip this section. However, it would be valuable to see how the binary COM contract translates itself to .NET, as similar ideas apply to other platforms.

Add a new project to the solution, this time a C# .NET Console application, and name it *CalcCS* (figure 21-13).

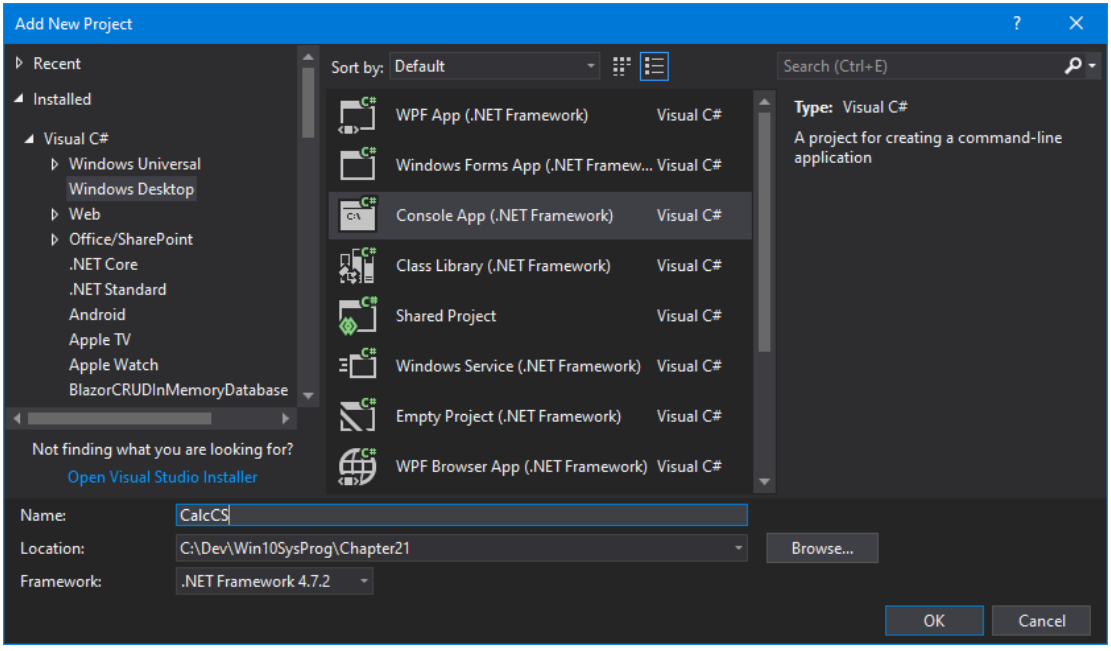


Figure 21-13: New C# Console Application project

We need to declare a C# interface but decorate it with certain attributes that mark it as a COM interface like so:

```
using System;
using System.Runtime.InteropServices;

namespace CalcCS {
    [ComImport, Guid("7020B8D9-CEFE-46DC-89D7-F0261C3CDE66")]
    [InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
    interface IRPNCalculator {
        void Push(double value);
        double Pop();
        void Add();
        void Subtract();
    };
```

The interface is decorated with an attribute that indicates it's a COM interface that derives from `IUnknown` (`InterfaceType` attribute), with the correct GUID. Then, the interface members are specified in v-table order, just as defined in the C++ header file. There are some anomalies here:

- All `HRESULT`s are removed, replaced with `void`. This is because by default if a failed `HRESULT` is returned, it's turned by .NET into a .NET exception

(`System.Runtime.InteropServices.COMException` class).

- The `Pop` method returns its value directly, rather than with a pointer/reference. This is another nice feature we get for free if the last parameter of a method is an output pointer marked with the `[retval]` attribute.

It's possible to remove all the above niceties from a method and retain the original signature by adding the `[PreserveSig]` attribute over the method.

The next step is to add a C# class declaration with the proper attributes so that it's recognized as a COM class:

```
[ComImport, Guid("FA523D4E-DB35-4D0B-BD0A-002281FE3F31")]
class RPNCalculator { }
```

Finally, we can write the `Main` method that creates an instance and invokes operations:

```
static class Program {
    [STAThread]
    static void Main(string[] args) {
        IRPNCalculator calc = (IRPNCalculator)new RPNCalculator();
        calc.Push(10);
        calc.Push(20);
        calc.Add();
        Console.WriteLine(calc.Pop());
    }
}
```

The `[STAThread]` attribute causes the main thread to call `CoInitialize(nullptr)` internally, which is critical in this case for apartment compatibility reasons (see the "Threads and Apartments" section later in this chapter for more details). The `new` operator calls `CoCreateInstance` thanks to the attributes specified for the `RPNCalculator` class. The cast to `IRPNCalculator` causes a `QueryInterface` method to be invoked behind the scenes.

The class name and interface name in the C# example don't mean anything; any names can be chosen. The important parts are the GUIDs and the order of functions in the interface.

Lastly, the project must be compiled with the same "bitness" as the DLL, so that the DLL can be loaded into the client process. By default, the *AnyCPU* configuration is used by Visual Studio, which compiles

for 32-bit by default. If the COM DLL is 64-bit, the *Prefer 32-bit* checkbox must be cleared in the project's properties *Build* tab (figure 21-14).

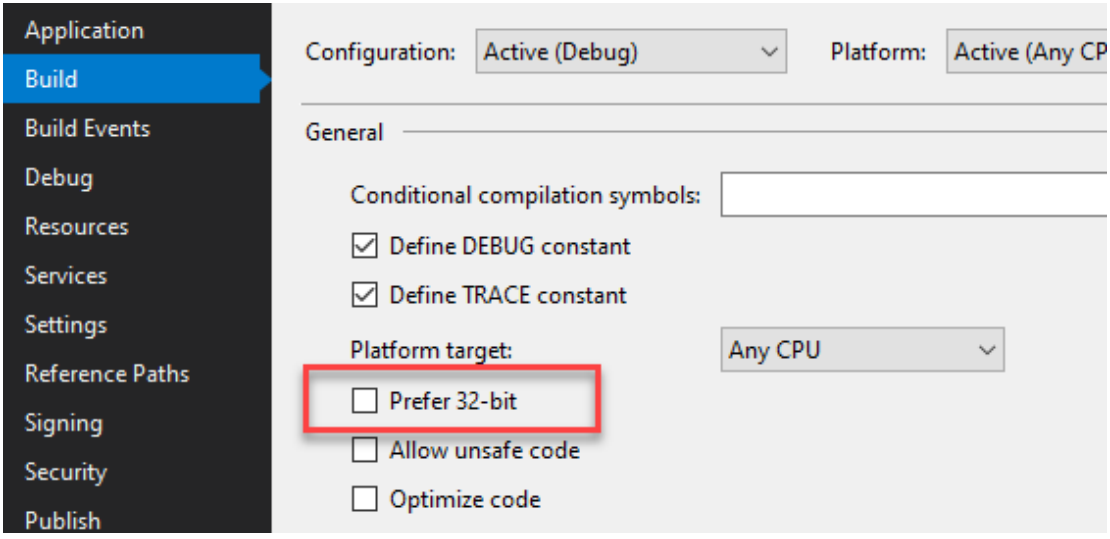


Figure 21-14: C# Project properties

Proxies and Stubs

The BITS example we looked at worked because the client got an interface pointer to a proxy object that looks like the real object. Any method call causes the proxy to marshal the arguments to the server process, where a stub object accepts the arguments, unmarshals them, builds a stack frame, and calls the real object. Any output parameters are then marshalled back to the client with the roles of proxy and stub reversed (see also figure 21-2).

The question is how did the proxy and stub got created, and why these particular proxy/stub types? Generally speaking, COM supports marshaling through the standard `IMarshal` interface. If the object supports `IMarshal`, the COM runtime works with that interface to figure out how to marshal the given interface. (`IMarshal` is beyond the scope of this chapter).

If `IMarshal` is not supported on the object, the Registry is searched for the interface ID to be marshaled, at *HKEY_CLASSES_ROOT\Interface*. Note that marshaling is based on interfaces rather than classes. This makes sense, since interfaces define the call semantics, not classes, which are just implementations.

Let's look at the `IBackgroundCopyManager` interface used in the BITS demos. Its interface ID is "". Here is the Registry data shown in the REG file format for simplicity:

```
[HKEY_CLASSES_ROOT\Interface\{5CE34C0D-0DC9-4C1F-897C-DAA1B78CEE7C}]
@="IBackgroundCopyManager"
[HKEY_CLASSES_ROOT\Interface\{5CE34C0D-0DC9-4C1F-897C-DAA1B78CEE7C}\
    NumMethods]
@="7"
[HKEY_CLASSES_ROOT\Interface\{5CE34C0D-0DC9-4C1F-897C-DAA1B78CEE7C}\
    ProxyStubClsid32]
@="{5CE34C0D-0DC9-4C1F-897C-DAA1B78CEE7C}"
```

The subkey *ProxyStubClsid32* points to the CLSID that implements the proxy and stub. Searching for that CLSID under the usual *HKCR\CLSID* finds this:

```
[HKEY_CLASSES_ROOT\CLSID\{5CE34C0D-0DC9-4C1F-897C-DAA1B78CEE7C}]
@="PSFactoryBuffer"
[HKEY_CLASSES_ROOT\CLSID\{5CE34C0D-0DC9-4C1F-897C-DAA1B78CEE7C}\
    InProcServer32]
@="C:\\Windows\\System32\\BitsProxy.dll"
"ThreadingModel"="Both"
```

Observant readers might spot the fact that the CLSID for the proxy/stub is the same as the interface ID for the `IBackgroundCopyManager`. This is not a coincidence nor a necessity, but the default selection made by the *MIDL* compiler (discussed in the next section).

The proxy/stub pair are COM classes just like any other, but they are created using a custom class factory implementing the `IPSFactoryBuffer`, shown here to get a "feel" for proxy/stub creation:

```
struct __declspec(uuid("D5F569D0-593B-101A-B569-08002B2DBF7A"))
IPSFactoryBuffer : public IUnknown {
public:
    virtual HRESULT __stdcall CreateProxy(
        _In_  IUnknown *pUnkOuter,
        _In_  REFIID riid,
        _Outptr_  IRpcProxyBuffer **ppProxy,
        _Outptr_  void **ppv) = 0;
    virtual HRESULT __stdcall CreateStub(
        _In_  REFIID riid,
        _In_opt_  IUnknown *pUnkServer,
        _Outptr_  IRpcStubBuffer **ppStub) = 0;
};
```

The proxy/stub DLL implementation is clearly in *BitsProxy.dll*. If you run the BITS samples, you'll find this DLL loaded into the client process as well as the server process (figures 21-15 and 21-16).

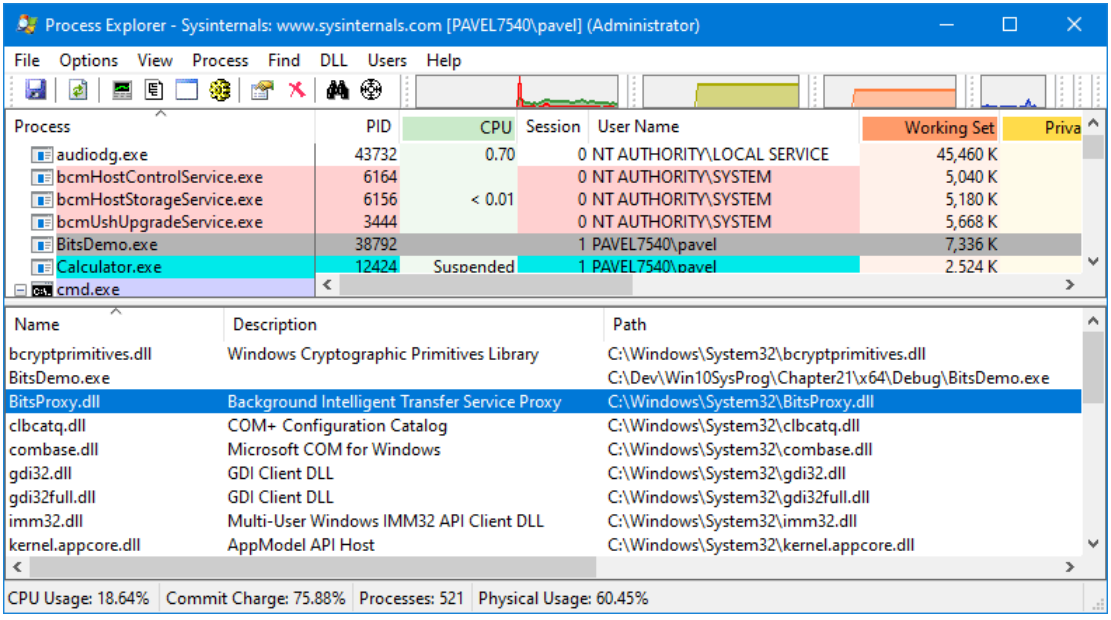


Figure 21-15: BITS proxy/stub DLL in a client

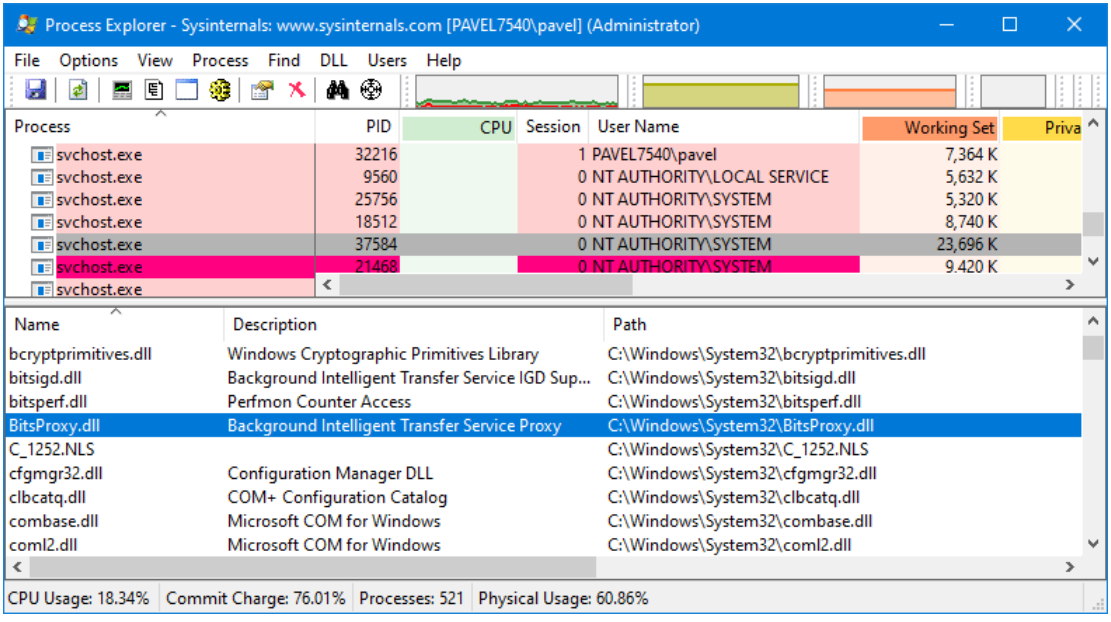


Figure 21-16: BITS proxy/stub DLL in the server (BITS service host)

Search for the interface `IBackgroundCopyJob` and locate its proxy/stub DLL.

IDL and Type Libraries

If you open the include file *bits.h* we used in the BITS demo, you'll find it to be "ugly", clearly stating at the beginning that it is a generated file rather written by a human:

```
/* this ALWAYS GENERATED file contains the definitions
    for the interfaces */
/* File created by MIDL compiler version 8.01.0623 */
```

This means the various BITS interfaces, enums, structs and other definitions were created using a different file called *Interface Definition Language* (IDL) and compiled with the associated compiler, the Microsoft IDL (MIDL) compiler. But why do we need yet another language for describing COM entities such as interfaces? isn't C++ enough?

The main issue is that C++ is not precise enough for generating correct proxy/stub code. For example, here is a simple interface method definition in C++:

```
HRESULT __stdcall DoWork(int* p);
```

From a C++ perspective, the pointer `p` can point to one integer, or a hundred. Or perhaps, the value is put inside that integer by the method implementation. The point is, there is no way to tell. This is fine if the client and object are connected directly in the same process. But if they are in different processes, the proxy needs to know how to deal with the pointer - how many integers to copy from the client to the server? Or perhaps just copy back?

This is where IDL comes in. It provides attributes that can be added (in square brackets) to make the definition precise and all the MIDL compiler to (among other things) generate a proper proxy/stub DLL implementation. Here is one way to write this method in IDL:

```
HRESULT DoWork([in, size_is(100)] int* p);
```

This example indicates this is an input parameter (copy from client to server but not back) and is an array of 100 integers.

In the BITS example, the original *bits.idl* is provided as part of the Windows SDK. Here is an excerpt showing the description of `IBackgroundCopyManager`:

```
[
    uuid(5ce34c0d-0dc9-4c1f-897c-daa1b78cee7c),
    helpstring("Background Copy interface"),
    odl
]
interface IBackgroundCopyManager : IUnknown {
    HRESULT CreateJob(
        [in]  LPCWSTR       DisplayName,
        [in]  BG_JOB_TYPE   Type,
        [out] GUID *        pJobId,
        [out] IBackgroundCopyJob **ppJob);

    HRESULT GetJob( [in] REFGUID jobID,
                    [out] IBackgroundCopyJob **ppJob );

    cpp_quote("#define    BG_JOB_ENUM_ALL_USERS  0x0001")

    HRESULT EnumJobs( [in] DWORD dwFlags,
                      [out] IEnumBackgroundCopyJobs **ppEnum );

    HRESULT GetErrorDescription(
        [in] HRESULT hResult,
        [in] DWORD LanguageId,
        [out] LPWSTR *pErrorDescription );
}
```

> IDL looks like C++ - it's similar in syntax, but definitely not the same. IDL is purely declarative, there is never any implementation. A full discussion of IDL is beyond the scope of this chapter.

Back in the C# example, we defined the `IRPNCalculator` interface explicitly and were careful not to make any errors. It would be nice if a "universal header" file would exist, that could be interpreted using a neutral API and translated to the client platform that can consume it.

This is the other benefit of using IDL. The MIDL compiler can produce a *type library*, which is a binary representation of the IDL, easily consumed by many platforms and languages, including C++ and .NET.

Type libraries can also be registered in the Registry under *HKCR\TypeLib*, and these can be easily consumed by .NET because .NET knows how to read type libraries and generate the wrapper .NET assembly with the correct interface definitions (and GUIDs, CLSIDs, etc.).

Type library registration is only necessary when using the *type library marshalar* (also called *Universal Marshalar*). This is out of scope for this chapter.

Unfortunately, there is no BITS type library provided with the Windows SDK, but since the IDL is provided, we can generate the type library by manually compiling the IDL file.

Copy *bits.idl* to some temporary directory. Open the Visual Studio Developer command window so that the environment variables are set to easily locate the tools we need. Then compile the IDL file:

```
c:\temp\>midl bits.idl
Microsoft (R) 32b/64b MIDL Compiler Version 8.01.0623
Copyright (c) Microsoft Corporation. All rights reserved.
Processing .\bits.idl
bits.idl
Processing C:\Program Files (x86)\
    Windows Kits\10\include\10.0.20175.0\um\unknwn.idl
unknwn.idl
Processing C:\Program Files (x86)\
    Windows Kits\10\include\10.0.20175.0\shared\wtypes.idl
...
Processing C:\Program Files (x86)\
    Windows Kits\10\include\10.0.20175.0\um\oaidl.acf
oaidl.acf
```

Now a *bits.tlb* file was created. You can verify that this file contains the same information as the IDL with the *OleView* tool. Run *oleview* from the developer command window. Select the *File* / *View type lib...* and browse to the *bits.tlb* file. You'll see the "decompiled" type library in IDL syntax (figure 21-17).

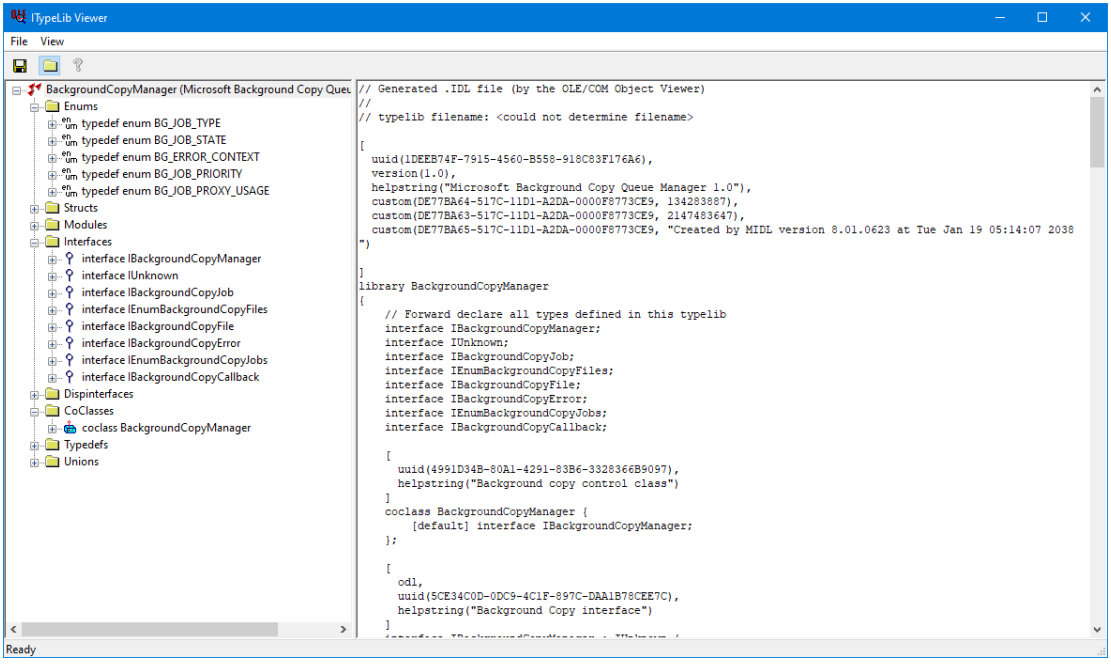


Figure 21-17: Type library decompilation with *OleView*

Now run the following to generate the COM wrapper assembly (this is more reliable than adding a reference from Visual Studio directly on the TLB file):

```
C:\temp>tlbimp bits.tlb /o:bits.dll
Microsoft (R) .NET Framework Type Library to Assembly Converter 4.8.4161\
.0
Copyright (C) Microsoft Corporation.  All rights reserved.

TlbImp : Type library imported to C:\temp\bits.dll
```

Now we can add the resulting DLL (.NET assembly) as a "normal" reference to a C# project. First, create a new C# project named *BitsDemoCS*. Copy the *Bits.Dll* file to the project's folder so it's within the bounds of the solution and not lying around in some random directory. Add a reference to the *Bits.dll* file, by browsing to its location.

Here is some simple C# code to test the managed DLL wrapper:

```
using bits;
using System;
using System.Threading;

namespace BitsDemoCS {
    static class Program {
        static void Main(string[] args) {
            var mgr = new BackgroundCopyManager();
            mgr.CreateJob("My managed Job",
                BG_JOB_TYPE.BG_JOB_TYPE_DOWNLOAD,
                out var guid, out var job);
            job.AddFile(
                "http://speedtest.ftp.otenet.gr/files/test10Mb.db",
                @"c:\temp\test.db");
            job.Resume();

            BG_JOB_STATE state;
            while(true) {
                job.GetState(out state);
                if (state == BG_JOB_STATE.BG_JOB_STATE_TRANSFERRED ||
                    state == BG_JOB_STATE.BG_JOB_STATE_ERROR)
                    break;
                Thread.Sleep(500);
                Console.Write(".");
            }
            Console.WriteLine();
            if (state == BG_JOB_STATE.BG_JOB_STATE_TRANSFERRED) {
                job.Complete();
                Console.WriteLine("Transferred successfully.");
            }
            else {
                Console.WriteLine("Error!");
            }
        }
    }
}
```

A type library file can also be used by Visual C++ to generate a C++ header. The Microsoft-specific `#import` keyword is the key. Here is an example:

```
#import "bits.tlb" raw_interfaces_only no_namespace

// use the BITS interfaces/classes normally...
```

The `#import` keyword reads a type library file (or a DLL/EXE with a type library embedded as a resource) and creates a file with a TLH extension (type library header) that is automatically fed to the compiler (no need to `#include` it explicitly). The attributes shown affect some aspects of the generated file(s). For example, `no_namespace` prevents the generation of a C++ namespace and `raw_interfaces_only` prevents creating wrappers that throw a C++ exception for failed `HRESULT`s (a TLI file is also generated if this attribute is not specified).

> `#import` supports are other useful attributes, refer to the documentation for the details.

Threads and Apartments

Different COM classes have different needs in terms of threading. For example, some COM class implementations are thread-safe, while others are not. Some COM objects need to work with UI-based clients, so being invoked on a single thread (the UI thread) is beneficial.

To that end, the COM infrastructure provided automatic protection and marshaling when needed when a client's threading model is different than the object it's communicating with. COM defines the concept of an *Apartment* as a logical grouping of objects that share threading requirements. Three types of apartments are supported:

- **Single Threaded Apartment** (STA) - an apartment where only a single thread lives. This thread is the one invoking methods on all objects that live in that apartment. Any number of STAs can exist in a process.
- **Multithreaded Apartment** (MTA) - an apartment where any number of threads may live. There can be at most one MTA in a process.
- **Thread Neutral Apartment** (TNA or NTA) - an apartment where threads cannot live in. Threads can, however, "rent" the apartment and invoke methods on objects with any thread.

For COM DLLs, the threading requirement of a class is provided in the Registry in a value called *ThreadingModel* set in the *InProcServ32* key (see figure 21-18). Table 21-2 shows the valid values for *ThreadingModel* and their meaning.

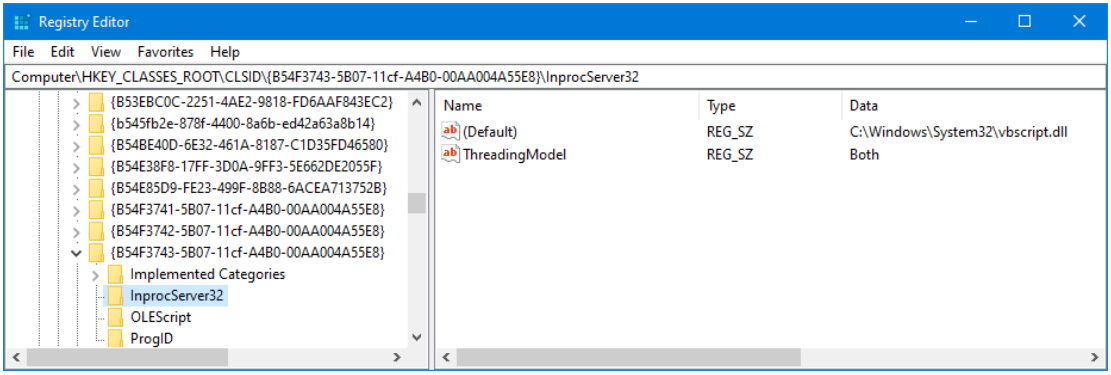


Figure 21-18: *ThreadingModel* value in the Registry

Table 21-2: *ThreadModel* values

Value	Description
Single (or none)	Main STA (first STA created in the process)
Apartment	STA
Free	MTA
Both	STA or MTA (same as the client's apartment)
Neutral	TNA

Figure 21-19 shows a schematic view of apartments in a process.

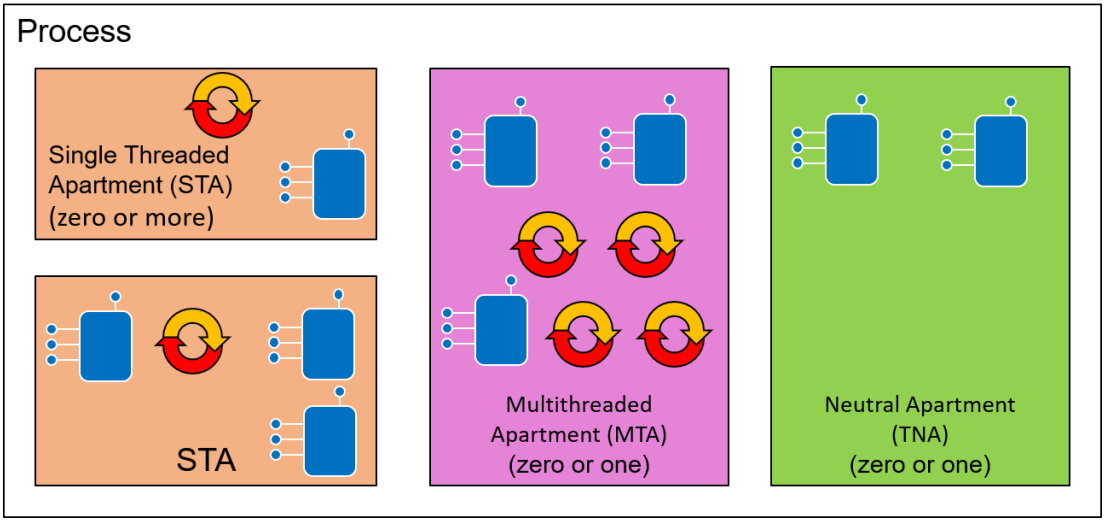


Figure 21-19: Apartments ina process

The other side of the coin is the client's thread that calls `CoCreateInstance` or

`CoGetClassObject`. COM clients must call `CoInitialize` or `CoInitializeEx` and that puts the calling thread into an apartment according to these rules:

- Calling `CoInitialize` puts the client into a new STA.
- Calling `CoInitializeEx` with the second argument set to `COINIT_APARTMENTTHREADED` is the same as `CoInitailzie`.
- Calling `CoInitializeEx` with the second argument set to `COINIT_MULTITHREADED` puts the thread in the process-wide MTA.

Whenever a client thread wants to be in an STA, a new STA must be created, where that thread is the sole resident thread. This is "by definition", as an STA is a *single thread* apartment. Conversely, if a thread wants to enter the MTA, there is only one such MTA in a process, since this is by definition shared by all threads that want to live in the MTA.

When a client attempts to create a COM class implemented in a DLL, COM must check the compatibility between the caller's apartment and the class' setting in the Registry. If they match, the client receives a direct pointer to the newly created object. Otherwise, a proxy is created and returned to the client to bridge the mismatch so the object gets the concurrency model specified in the Registry.

Figure 21-20 shows the simple case where two threads, in each its own STA creating objects where the *ThreadingModel* is equal to *Apartment*.

The name *Apartment* in the Registry is a bad name, since an MTA is an apartment as well. The term *Apartment* in the *ThreadModel* value means "STA".

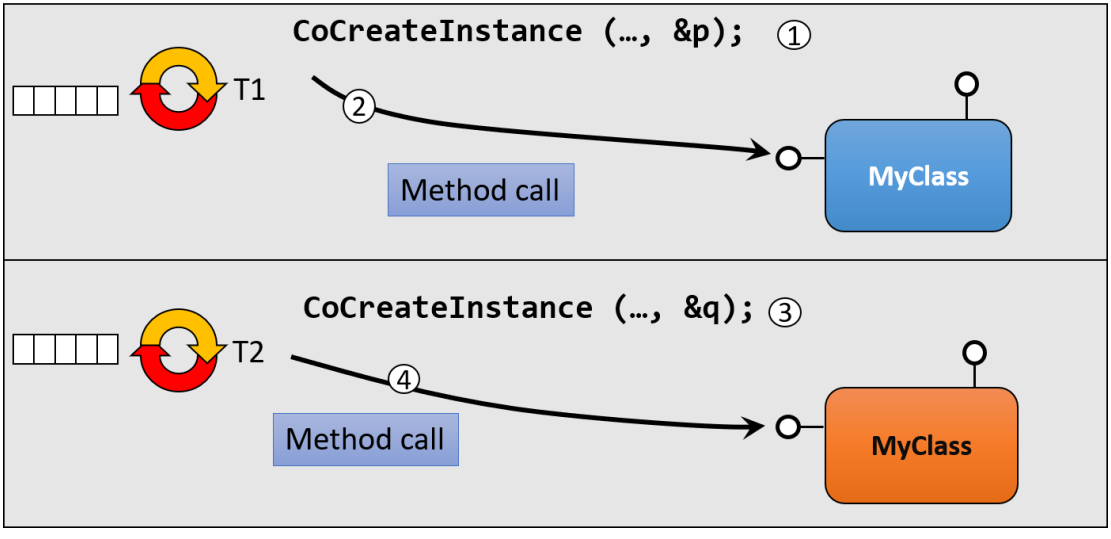


Figure 21-20: STA clients create STA objects

Threads T1 and T2 in figure 21-20 call `CoCreateInstance`, each creating an instance in each own's apartment because the objects' requirements in the Registry are compatible with the apartment the client thread is in.

A proxy is created when there is mismatch of apartments. Figure 21-21 shows such an example, where T1 enters the MTA and creates an instance of `MyClass`, where its Registry *ThreadingModel* value specifies *Apartment* (STA). The class is in control, so is created in an STA (COM will create that STA with a thread, called "Host STA"). The object will be created in the STA, and a proxy will be returned to the client.

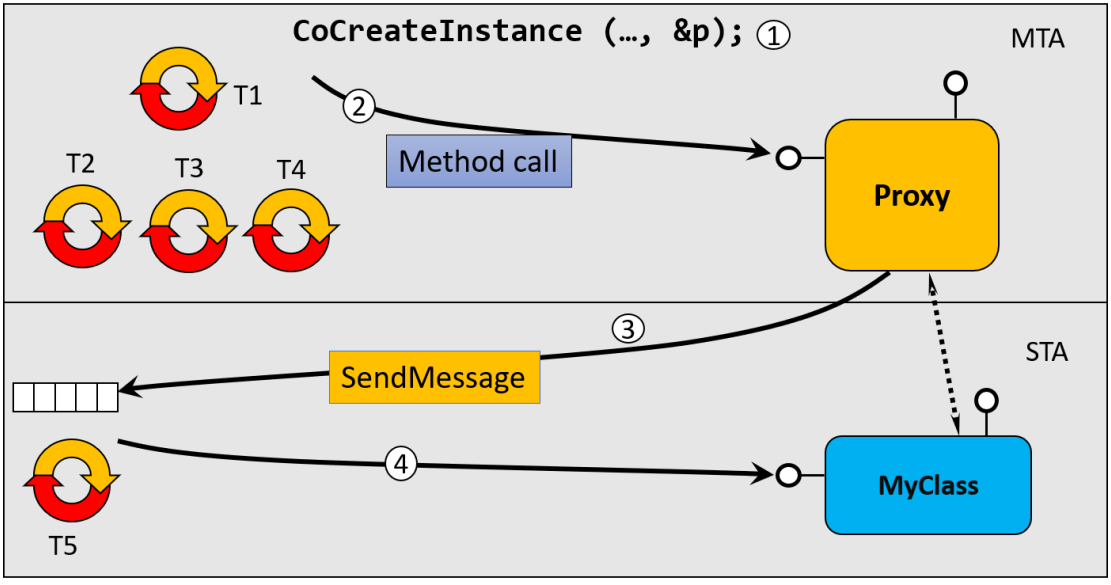


Figure 21-21: MTA client creating an STA object

Now the client thread wants to invoke a method. It does so on the proxy (although the client need not know that). The proxy does what is needed to make this work, which depends on the target apartment. If the target is an STA (as in figure 21-21), a message is sent to a hidden window created in every STA by the `CoInitialize` call. The thread in the STA (T5 in figure 21-21) must be processing messages for this to work, otherwise T1 will hang, waiting indefinitely for the call to complete.

> This requirement of STA threads to pump messages is the reason the callback BITS example in the section "Implementing COM Interfaces" will hang if the initial thread does not enter the MTA, because the last call of this thread is an infinite sleep.

Message pumping is done with the same code typically used to process UI messages:

```
MSG msg;
while(::GetMessage(&msg, nullptr, 0, 0)) {
    ::DispatchMessgae(&msg);
}
```

If the thread in question already has a message loop (already a GUI thread), then there is no need to add that code anywhere.

T1 in figure 21-21 can safely pass its proxy pointer to threads T2, T2 and T4 to call the same object. All calls will be serialized any way using the message queue handled by thread T5. In this case, T5 was created by the COM runtime as part of object creation and as such provides the correct message pumping logic.

The Free Threaded Marshalar (FTM)

Suppose an object wants to be accessed directly at all times, regardless of any apartment settings. First it specifies *Both* in the *ThreadingModel*, meaning it's always created in the apartment of the caller (no proxy). However, what if the calling thread wants to pass the thread to a different apartment to be used by another thread. This requires *marshaling* the pointer correctly to the other apartment, making a proxy appear in the other apartment. The COM API provies several functions to perform this marshaling explicitly. In most cases, however, this is done automatically if such an interface pointer is a parameter in a method.

If an object does not wish to incur the overhead of a proxy in all (in-process) cases, it can do so by aggregating an object implemented by COM called the *Free Threaded Marshalar* (FTM). This object provides a custom implementation of `IMarshal` that always returns a direct pointer to the object, rather than creating a proxy for incompatible apartments.

Aggregation in COM parlance is an object reuse mechanism that is closest to object-oriented inheritance, but does not involve any source code (as COM is a binary standard). Figure 21-22 shows an example of aggregation vs. the simpler reuse mechanism known as *Containment*.

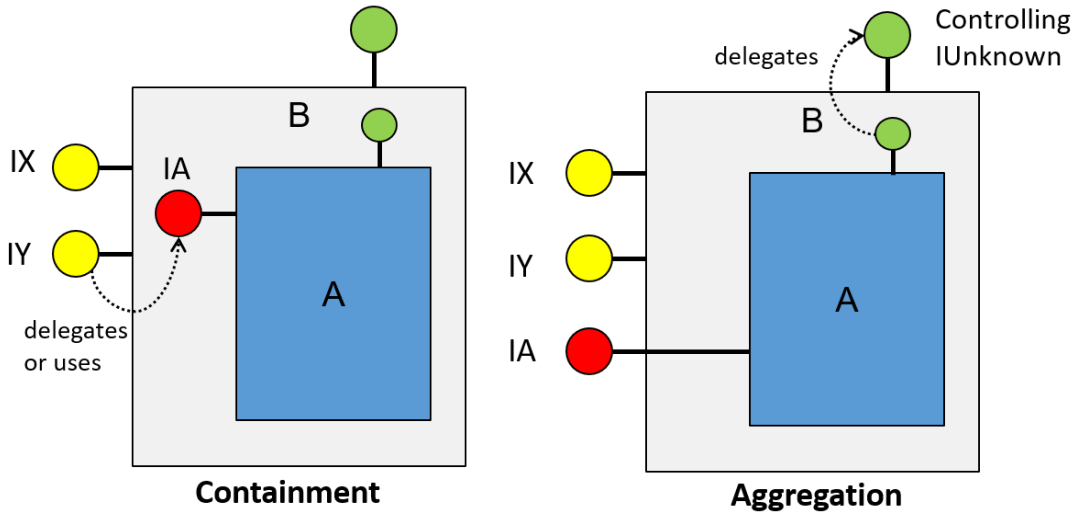


Figure 21-22: Containment (left) vs. Aggregation (right)

In figure 21-22, object B aggregates A (right), meaning interface IA is exposed directly outside B. As far as clients are concerned, the object B implements IX, IY and IA.

Figure 21-23 shows what an object aggregating the FTM looks like. In process, this object will be always be accessed directly with no proxy. This also means the object must have a thread-safe implementation.

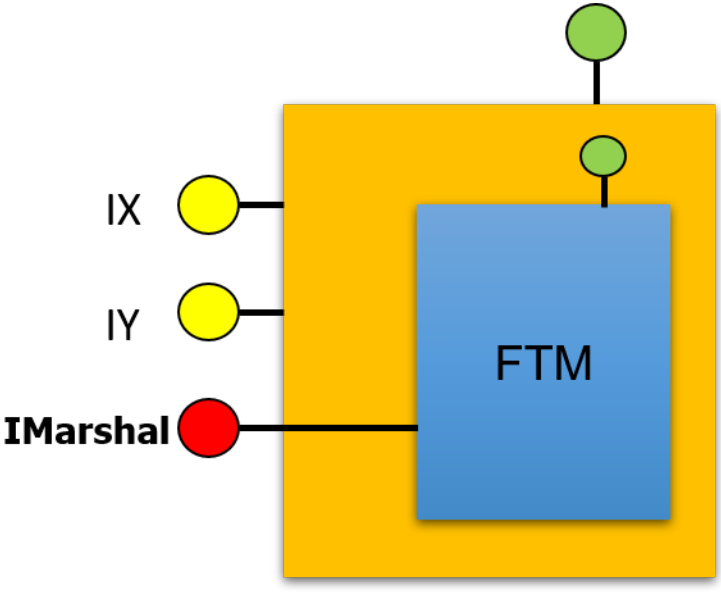


Figure 21-23: Aggregating the FTM

Creating and aggregating the FTM is done with `CoCreateFreeThreadedMarshalar`:

```
HRESULT CoCreateFreeThreadedMarshaler(
    _In_opt_ LPUNKNOWN punkOuter,
    _Outptr_ LPUNKNOWN* ppunkMarshal);
```

The input is the `IUnknown` of the outer object (also called *Controlling IUnknown*) that must be called by the inner object when `IUnknown` methods are called on it, since it cannot know what interfaces are supported by the other object. This means that an aggregated object must handle `IUnknown` differently than non-aggregatable object. The result (second parameter) is a pointer to the FTM.

> Full coverage of marshalling, the FTM and related topics such as the *Global Interface Table* (GIT) is beyond the scope of this chapter.

Odds and Ends

Implementing COM classes in the way that was done in the section *COM Servers* is possible, but it requires a lot of boilerplate code that is best implemented by a library (for example, `IUnknown` and class factory implementations as well as registering and unregistering).

The *Active Template Library* (ATL) is the most well-known library that can be used for such purpose. We did use some of its facilities for client-side COM development, such its smart pointers. ATL has a lot of infrastructure for building COM servers, as DLLs or EXEs. ATL is beyond the scope of this book, but there are many resources for using ATL on the web and in books.

A modern alternative for using COM is with the *Windows Runtime Library* (WRL) classes, that also provide client-side support with smart pointers, as well as classes for implementing COM servers. WRL is modern, using C++11 and later language facilities, while ATL is much older and does not use modern C++.

ATL is somewhat richer, providing support for more COM-related features, such as connection points, registration, ActiveX controls and more. WRL was created mostly to support the Windows Runtime (see next chapter), but still provides decent support for classic COM as well. Ultimately, the library you use is mostly a matter of taste, as you can accomplish everything COM-related with either.

Summary

The *Component Object Model* is rich in functionality. We just scratched the surface and presented the most important ideas and coding practices needed to work with existing implementations of COM classes and provide some of our own when needed.

Chapter 22: The Windows Runtime

The *Windows Runtime* (sometimes abbreviated *WinRT*) was introduced in Windows 8, primarily for the purpose of building the so-called *Universal Windows Platform* (UWP) applications. These types of applications had many past names: Metro, modern, store, universal - but all mean the same thing: applications that can be packaged in a standard way and uploaded to the Microsoft Store.

The Windows Runtime is built on top of COM, extending or modifying it in at least a couple of important ways:

- WinRT classes are identified by strings, rather than class IDs (GUIDs).
- The fundamental interface WinRT types must implement is called `IInspectable` (which derives from `IUnknown`).

In this chapter, I will introduce you to working with WinRT, focusing on the foundations. Coverage of UWP application development is beyond the scope of this book since it has little to do with System Programming.

In this chapter:

Introduction

The Windows Runtime is built on top of COM, extending and modifying some aspects. COM was chosen because of its battle-proven infrastructure and powerful concepts: interfaces, class factories, marshaling, and so on, most of which are covered in chapter 21.

Most of the COM concepts and ideas are carried forward to WinRT, including apartments, interfaces, v-tables, class factories (called activation factories in WinRT parlance), and `HRESULT`s.

The first enhancement WinRT introduced compared to COM is the addition of a base interface, which all WinRT types must implement, and all interfaces must derive from. Here is the IDL version (from *Inspectable.idl*):

```
typedef [v1_enum] enum TrustLevel {
    BaseTrust,
    PartialTrust,
    FullTrust
} TrustLevel;

[
  object,
  uuid(AF86E2E0-B12D-4c6a-9C5A-D7AA65101E90),
  pointer_default(unique)
]
interface IInspectable : IUnknown {
    HRESULT GetIids(
        [out] ULONG * iidCount,
        [out, size_is(,*iidCount)] IID** iids);
    HRESULT GetRuntimeClassName( [out] HSTRING* className);
    HRESULT GetTrustLevel([out] TrustLevel* trustLevel);
}
```

`IInspectable` adds a way to query an object for its list of interfaces with `GetIids`. With `IUnknown`, you can query for a specific interface, but there is no built-in way to get a list of supported interfaces. This capability is needed for some language projections, mainly JavaScript. `GetRuntimeClassName` returns the class name of the instance (using an `HSTRING`, discussed in the next section). (Remember that the same interface can be implemented by any number of types.) Finally, `GetTrustLevel` returns a `TrustLevel` enumeration for the object, which can be useful in some specialized scenarios.

The original proposed name for `IInspectable` was `IKnown`, to complement `IUnknown`. Eventually, it seemed too confusing, so `IInspectable` was chosen instead.

WinRT adds ideas from various programming languages, such as namespaces, properties, methods (instance and static), asynchronous operations, and constructors, not directly found in COM (technically, instance methods and properties do exist in COM). WinRT has full metadata describing the classes, interfaces, methods, etc., similar to COM type libraries. There are differences, however:

- WinRT metadata is complete and mandatory (COM type libraries are optional and may be incomplete).
- WinRT Metadata format is richer, using the same metadata format used by .NET. COM's type library format does not define certain concepts, like namespaces or constructors.

The metadata is required for some language projections at runtime, but in general can (and is) used to generate language projections for any language/platform that wishes to interact with WinRT APIs. Microsoft provides several of those projections: for C++ (C++/WinRT), C# (and other .NET languages),

JavaScript, and Rust. It's not difficult to add projections for other languages, because of the neutral format of the metadata (usually bundled in files with the extension *winmd*).

Because the WinRT metadata uses the .NET format, it can be viewed with any tool available for inspecting .NET metadata, such as *ILSasm*, *ILSpy*, *dotPeek*, and *Just Decompile*. Figure 22-1 shows a screenshot of Jetbrain's *dotPeek* tool showing the metadata for the `Calendar` class. Use the *File/Open…* menu item and navigate to the *System32\WinMetadata* directory, where you'll find the WinRT APIs metadata files.

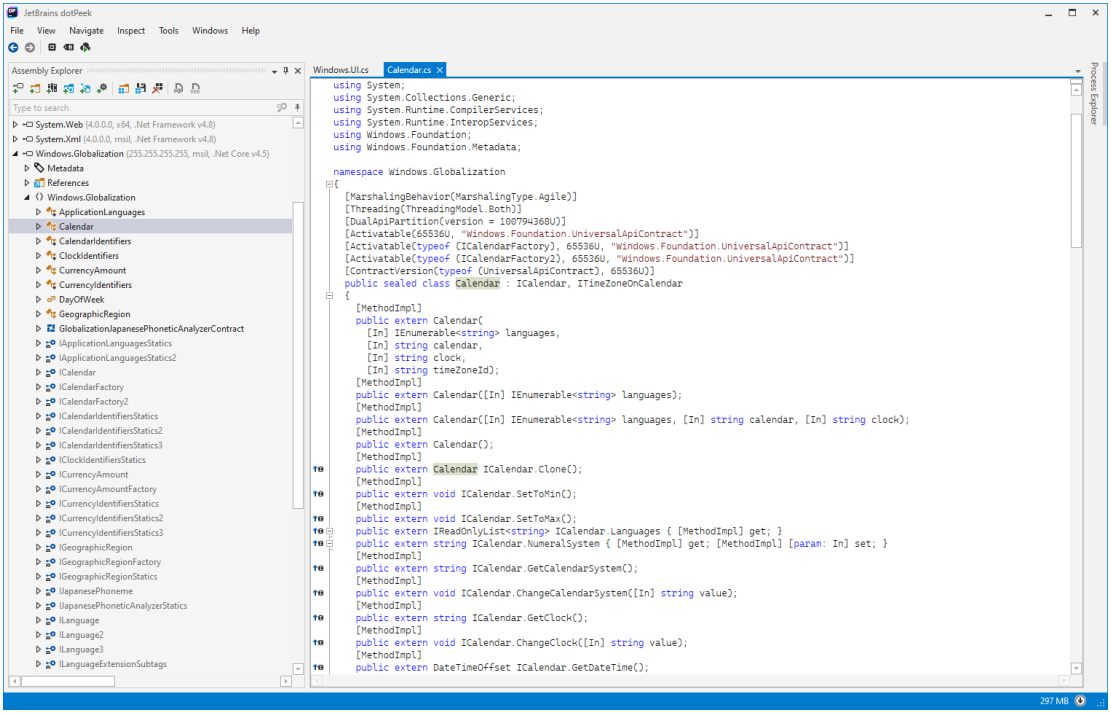


Figure 22-1: Viewing WinRT metadata in *dotPeek*

Working with WinRT

Using pure C++ to interact with WinRT is possible, but is fairly verbose, and gets tedious quickly. To get a sense of what that looks like, we'll create a simple application that creates an instance of the `Windows.Globalization.Calendar` class and use some of its members.

> I am using the WinRT metadata syntax for specifying classes/namespaces, separating their components with a dot. In C++, these would translate to the usual scope resolution operator (`::`).

The first thing to do is add proper includes:

```
#include <windows.globalization.h>
#include <roapi.h>
#include <atlbase.h>
#include <string>
#include <stdio.h>
```

Windows.globalization.h is the header file defining all types within the `Windows.Globalization` namespace. *roapi.h* defines many of the common WinRT C APIs, some of which we need to use. *atlbase.h* is included to get smart pointer support (the same ones used in chapter 21), although most WinRT examples you may come across use another smart pointer type called `ComPtr<>`.

The `ComPtr<>` smart pointer is more verbose, but explicit, than the ATL equivalent. Personally, I prefer the ATL ones. Either works, however, and performs the necessary reference counting. `ComPtr<>` is more explicit, in the sense that operator overloading is not used in some cases, requiring explicit methods calls instead. For example, specifying the address of the internal interface pointer must be done with `GetAddressOf()` rather than simply using the *address-of* operator & as is done with ATL. `ComPtr<>` is defined in *wrl.h* within the `Microsoft::WRL` namespace.

WRL is short for *Windows Runtime Library*, an early attempt to create convenient wrappers for WinRT usage with C++. It still exists and works, but there is a better way - C++/WinRT, discussed in the next section.

Use precompiled headers to speed compilation times when compiling WinRT headers, as they pull in lots of code from other headers.

Next, we need to add an import library for the WinRT C APIs we're about to use:

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

Another helpful statement would be a namespace `using` declaration to simplify names in the upcoming `main` function:

```
using namespace ABI::Windows::Globalization;
```

ABI stands for *Application Binary Interface*, a concept that defines the boundary between the WinRT APIs and any language projection used to access the underlying API. For a pure consumer C++ example, there is no real boundary, as we'll be using the WinRT types and functions directly. All the "pure" WinRT definitions (those generated by the enhanced MIDL compiler) are stored within the `ABI` top-level namespace.

> The term *ABI* is a generic one, defining the boundary between some API/framework and its consumers that may be using some arbitrary language/platform. The ABI defines the rules that any user of the API needs to follow if types or functions are exported for other consumers. These exposed types/functions must adhere to the ABI rules. For example, in WinRT, any method that uses a string, must use an `HSTRING` rather than some other string type that wouldn't be able to pass the ABI boundary. Imagine, for example, using `std::wstring` with C++ - a C# consumer wouldn't be able to use it, as it does not conform to the WinRT ABI.

The first thing to do in `main` is enter an apartment:

```
int main() {
    ::RoInitialize(RO_INIT_MULTITHREADED);
```

`RoInitialize` is roughly equivalent to `CoInitializeEx`, requiring any thread that wants to access WinRT to enter an aprtment - either an STA or the process-wide MTA - just like in classic COM. The values are different, but mean the same thing as their `CoInitializeEx` counterparts. Valid values (part of `RO_INIT_TYPE` enumeration) are `RO_INIT_SINGLETHREADED` (STA) and `RO_INIT_MULTITHREADED` (MTA).

There is another apartment variant in WinRT, called the *Application STA* (ASTA). This one is used by UWP applications, where the apartment is a regular STA, but does not allow reentrancy. Further discussion of the ASTA is beyond the scope of this chapter.

Creating instances in WinRT is done via the `RoActivateInstance`, a simplified version of COM's `CoCreateInstance` API:

```
HRESULT RoActivateInstance(
    _In_ HSTRING activatableClassId,
    _COM_Outptr_ IInspectable** instance);
```

The class identifier is a string - an `HSTRING` to be precise. An `HSTRING` is the standard string type used in the WinRT ABI. It's a Unicode, reference counted, immutable string. The *immutable* part is very important - this string cannot change once created. Any potential changes must return a new string. Of course, in C++ it's easy to get a pointer to the underlying string buffer, so changes cannot be technically prevented, but doing so would constitude a bug.

The .NET string type is immutable as well.

Immutability in general, has some desirable benefits:

- Immutable objects are thread safe. Multiple threads can access the object concurrently, without and risk of a data race. This is because concurrent reads are never a problem.
- Immutable objects gauarantee predictability. For example, suppose you're passing a pointer to an immutable object to some function. Whatever the function does with the object (like passing it to other functions), when the original function returns, you can be rest assured that the object has not been modified, since it's immutable.

Some languages provide immutability as part of their built-in mechanisms. Examples include Rust and F#.

The second parameter to `RoActivateInstance` is the resulting `IInspectable` pointer. Notice there is no way to request another interface directly - a further call to `IUnknown::QueryInterface` is required.

In order to create a WinRT object, we need its class name as an `HSTRING`. One way of creating an `HSTRING` from a normal C string, is with `WindowsCreateString`:

```
HRESULT WindowsCreateString(
    _In_reads_opt_(length) PCNZWCH sourceString,    // const WCHAR*
    UINT32 length,
    _Outptr_result_maybenull_ HSTRING* string);
```

`WindowsCreateString` allocated a buffer to store the string, so there is no dependency on the original C string. Any string created in this way must be dereferenced with `WindowsDeleteString`:

```
HRESULT WindowsDeleteString(_In_opt_ HSTRING string);
```

Alternatively, an `HSTRING` can be created to wrap an existing C string without any allocation if it just needs to be used temporarily while the C string is alive by calling `WindowsCreateStringReference`:

```
HRESULT WindowsCreateStringReference(
    _In_reads_opt_(length + 1) PCWSTR sourceString,
    UINT32 length,
    _Out_ HSTRING_HEADER* hstringHeader,    // opaque
    _Outptr_result_maybenull_ HSTRING* string);
```

For our example, we can use either approach, but clearly using `WindowsCreateStringReference` is faster as no string allocation is involved. We'll examine both ways. Here is the first option using `WindowsCreateString`:

```
HSTRING hClassName;
HRESULT hr = ::WindowsCreateString(
    RuntimeClass_Windows_Globalization_Calendar,
    _countof(RuntimeClass_Windows_Globalization_Calendar) - 1,
    &hClassName);

CComPtr<IInspectable> spInst;
hr = ::RoActivateInstance(hClassName, &spInst);
::WindowsDeleteString(hClassName);
```

The `RuntimeClass_Windows_Globalization_Calendar` name is the fully-qualified class name of the calendar class defined like so:

```
const WCHAR RuntimeClass_Windows_Globalization_Calendar[]
    = L"Windows.Globalization.Calendar";
```

The `_countof` macro calculates the string length at compile time, which is faster than calling `wcslen` (at runtime).

The alternative to `WindowsCreateString` is using an `HSTRING` "reference" before calling `RoActivateInstance`:

```
HSTRING_HEADER header;  // internal and opaque
HRESULT hr = ::WindowsCreateStringReference(
    RuntimeClass_Windows_Globalization_Calendar,
    _countof(RuntimeClass_Windows_Globalization_Calendar) - 1,
    &header, &hClassName);
```

Now that we have an object instance on our hands, we can work with it by calling `QueryInterface` to locate the `ICalendar` interface, and use it if implemented:

```
if(SUCCEEDED(hr)) {
    CComQIPtr<ICalendar> spCalendar(spInst);    // QueryInterface
    if(spCalendar) {
        spCalendar->SetToNow();
        INT32 hour, minute, second;
        spCalendar->get_Hour(&hour);
        spCalendar->get_Minute(&minute);
        spCalendar->get_Second(&second);

        printf("The time is %02d:%02d:%02d\n", hour, minute, second);
    }
    spInst = nullptr; // calls IUnknown::Release
}
```

The above code is no different than using any other COM interface. Finally, we need to call `RoUninitialize` as the counterpart to `RoInitialize` (very similar to `CoUninitialize` from classic COM):

```
    ::RoUninitialize();
}
```

> This example can be found in the *DateTime* project sample.

The `IInspectable` interface

We can use `IInspectable` to query for the implemented interface on an object and its class name. We'll create a helper function to display this information. Let's start by displaying the class name:

```
HRESULT DisplayInterfaces(IInspectable* pInst) {
    HSTRING hName;
    if (SUCCEEDED(pInst->GetRuntimeClassName(&hName))) {
        ULONG32 len;
        auto name = ::WindowsGetStringRawBuffer(hName, &len);
        if (name)
            printf("Class name: %ws\n", name);
        ::WindowsDeleteString(hName);
    }
```

`WindowsGetStringRawBuffer` provides convenient access to the underling string buffer, so we can use it just like any other character pointer pointer.

Next, we'll see if `GetIids` fails, and return if it does:

```
ULONG count;
IID* iid;
auto hr = pInst->GetIids(&count, &iid);
if(FAILED(hr))
    return hr;
```

`GetIids` returns an array of GUIDs and the count of GUIDs. These are allocated internally with the COM task memory allocator, so we must remember to free that array when we're done by calling `CoTaskMemFree`.

Now we can iterate over the interface IDs. As a bonus, we can consult the Registry at *HKCR\Interface* for the interface ID, and if located we can display its friendly name.

> Remember that interface IDs don't have to be registered. Interfaces are registered if they can be marshaled to other apartments/processes (and the object does not implement `IMarshal`).

```
    WCHAR siid[64];
    WCHAR name[256];
    for(ULONG i = 0; i < count; i++) {
        if(SUCCEEDED(::StringFromGUID2(iid[i], siid, _countof(siid)))) {
            printf("I: %ws", siid);
            CRegKey key;
            if(ERROR_SUCCESS == key.Open(HKEY_CLASSES_ROOT,
                (std::wstring(L"\\Interface\\") + siid).c_str(),
                KEY_QUERY_VALUE)) {
                ULONG chars = _countof(name);
                if(ERROR_SUCCESS == key.QueryStringValue(L"", name,
                    &chars)) {
                    printf(" %ws", name);
                }
            }
            printf("\n");
        }
    }
    ::CoTaskMemFree(iid);
    return S_OK;
}
```

`CRegKey` is an ATL class that provides convenient access to Registry APIs. Running the above function with an `ICalendar` interface pointer yields the following output:

```
Class name: Windows.Globalization.Calendar
I: {CA30221D-86D9-40FB-A26B-D44EB7CF08EA} __x_Windows_CGlobalization_CIC\
alendar
I: {00000038-0000-0000-C000-000000000046} IWeakReferenceSource
I: {BB3C25E5-46CF-4317-A3F5-02621AD54478} __x_Windows_CGlobalization_CIT\
imeZoneOnCalendar
I: {0CA51CC6-17CF-4642-B08E-473DCC3CA3EF}
```

As hinted from the above output, the Calendar object supports four interfaces (excluding `IUnknown` and `IInspectable`). The first one is the `ICalendar` interface, and the third one is `ITimeZoneOnCalendar`, which is documented for working with time zones. Here is some code to use that interface given an `ICalendar` pointer used before:

```
CComQIPtr<ITimeZoneOnCalendar> spTZ(spCalendar);
if (spTZ) {
    HSTRING hTimeZone;
    if (SUCCEEDED(spTZ->TimeZoneAsFullString(&hTimeZone))) {
        auto tzname = ::WindowsGetStringRawBuffer(
            hTimeZone, nullptr);
        printf("Time zone: %ws\n", tzname);
        ::WindowsDeleteString(hTimeZone);
    }
}
```

Language Projections

The Windows Runtime was built from the beginning with multiple consumer types in mind. It's not just C++, but other languages should have access to WinRT APIs, and not just any access - it must be convenient access. Otherwise, WinRT would be a hard sell. Developers want to be productive, to write simple and easy to understand code, and have a good debugging experience. These traits are essential for any successful framework.

Although COM can be consumed by almost any language/platform, that wasn't good enough for WinRT. There are several reasons for that. Here are a few:

- WinRT supports static methods, something COM knows nothing about. But how would that work? WinRT is built on top of COM, and with COM instance methods is the only thing there is. There is no notion of "static" members.
- WinRT supports constructors, including overloaded ones. How would that be modeled through COM?
- Many WinRT APIs are asynchronous, meaning a method starts some operation, where the return value is an object representing the operation, but it's not the final result. How would the result be returned once the operation completes in a convenient way?
- WinRT supports events. How would those be exposed through COM?
- WinRT supports method (and constructor) overloading, but COM does not.

Let's expand on static members to get a sense of the complexities involved. COM does not support any notion of statics (as languages like C++ and C# do). The trick used is to define another interface ending with the word "Statics" that represents an interface implemented by a singleton object that acts like "static" members.

For example, the `NetworkInformation` class (from the `Windows.Networking.Connectivity` namespace) has a bunch of static methods. These are exposed by the `INetworkInformationStatics` interface, which means that to gain access to the "static" members, we need to create some object that implements that interface. Only then would we be able to access the members. This is not too difficult but requires a lot of code to be written for something that should be very simple.

The following example shows how to get the network profiles using the above description, displaying the profile name for each:

```
// pch.h

#include <windows.networking.connectivity.h>
#include <windows.foundation.h>
#include <windows.foundation.collections.h>
#include <wrl.h>
#include <atlbase.h>
#include <roapi.h>
#include <stdio.h>

// main.cpp

using namespace ABI::Windows::Networking::Connectivity;
using namespace ABI::Windows::Foundation;
using namespace ABI::Windows::Foundation::Collections;
using namespace Microsoft::WRL::Wrappers;    // for HStringReference

#pragma comment(lib, "runtimeobject.lib")

int main() {
    Initialize(RO_INIT_MULTITHREADED);    // from ABI::Windows::Foundati\
on

    CComPtr<INetworkInformationStatics> spStatics;
    auto hr = RoGetActivationFactory(
        HStringReference(
        RuntimeClass_Windows_Networking_Connectivity_NetworkInformation)\
.Get(),
        __uuidof(INetworkInformationStatics),
        reinterpret_cast<void**>(&spStatics));

    if (SUCCEEDED(hr)) {
        CComPtr<IVectorView<ConnectionProfile*>> spProfiles;
        spStatics->GetConnectionProfiles(&spProfiles);
        if (spProfiles) {
            unsigned count;
            spProfiles->get_Size(&count);
            for (unsigned i = 0; i < count; i++) {
                CComPtr<IConnectionProfile> spProfile;
                spProfiles->GetAt(i, &spProfile);
```

```
                HSTRING hName;
                spProfile->get_ProfileName(&hName);
                printf("Name: %ws\n",
                    ::WindowsGetStringRawBuffer(hName, nullptr));
                ::WindowsDeleteString(hName);
            }
        }
    }

    return 0;
}
```

> The above code is part of the *NetworkInfoRaw* sample project.

The static members are implemented on the activation factory (called class factory in COM), which makes sense since the activation factory is typically a singleton - it can create any number of instances. `HStringReference` is a helper type provided by *roapi.h* to simplify working with an `HSTRING` reference.

`GetConnectionProfiles` returns a `IVectorView<>`, which is a WinRT interface representing a read only collection of some type. In the above example, these are network profiles, represented by `ConnectionProfile` objects. This may seem weird - where is the interface? The default interface (in this case `IConnectionProfile` is hidden by the class name). You can see these (and other) manipulations declared as part of the metadata in the definition of `ConnectionProfile` in IDL (from *Windows.Networking.Connectivity.idl* found in the Windows SDK):

```
[contract(Windows.Foundation.UniversalApiContract, 1.0)]
[marshaling_behavior(agile)]
runtimeclass ConnectionProfile {
    [default] // the default interface
    interface Windows.Networking.Connectivity.IConnectionProfile;
    [contract(Windows.Foundation.UniversalApiContract, 1.0)]
    interface Windows.Networking.Connectivity.IConnectionProfile2;
    [contract(Windows.Foundation.UniversalApiContract, 1.0)]
    interface Windows.Networking.Connectivity.IConnectionProfile3;
    [contract(Windows.Foundation.UniversalApiContract, 5.0)]
    interface Windows.Networking.Connectivity.IConnectionProfile4;
    [contract(Windows.Foundation.UniversalApiContract, 7.0)]
    interface Windows.Networking.Connectivity.IConnectionProfile5;
}
```

> A deep dive into IDL is beyond the scope of this chapter.

Accessing the static members is inconvenient, to say the least. Even accessing properties, like `ProfileName` is not much fun, not to mention the access to the `IVectorView<>` object. Things get even worse when asynchronous operations come into play.

This is where language projections come in. They provide more convenient and idiomatic access to WinRT APIs. Microsoft's first attempt at a C++ related language projection is called *C++/CX*. This projection is based on extending the C++ language with keywords and semantics that would make it easy to access WinRT APIs.

C++/WinRT

Here is a simple example using C++/CX with the `Calendar` type:

```
Calendar^ calendar = ref new Calendar;
calendar->SetToNow();
printf("Time: %02d:%02d:%02d\n",
    // look like member variables
    calendar->Hour, calendar->Minute, calendar->Second);
```

The `ref new` keyword is syntactic sugar for calling `RoActivateInstance`. The "hat" symbol represents smart pointers behind the scenes. It doesn't seem too bad, but there are a few potential issues. First, it's not standard C++, which makes this code only work with Microsoft's compiler; and in general, learning new keywords for a language dialect is not ideal. Second, some aspects of WinRT were still difficult to deal with, mainly asynchronous operations. These (and some other) reasons pushed Microsoft to create another projection for C++ developers called *C++/WinRT*.

C++/WinRT uses standard C++ only, and requires C++17 compiler support. It provides a similar syntactic sugar as C++/CX, but additionally supports easy asynchronous operation consumption. This projection is generated based on WinMD (metadata) files, so can work with third-party WinRT components as well.

To get the proper support for C++/WinRT, you should install the C++/WinRT Visual Studio extension that provides several project templates to start with, as well as extension to the debugger that shows WinRT properties conveniently. The other option is to simply add the C++/WinRT Nuget package to an existing C++ project of any type.

> The full Nuget package name is *Microsoft.Windows.CppWinRT*.

C++/WinRT exposes WinRT classes as "normal" C++ classes. No need for special keywords, not even `new`. Just instantiate a class normally on the stack, or part of some other class/struct and that's it. These objects

are smart pointers in disguise, performing the WinRT creation (`RoActivateInstance` and the like) as needed. Static methods are invoked just as they are in C++, as are overloaded constructors and methods.

Here is the Calendar example using C++/WinRT (found in the *DateTime2* sample project):

```
// pch.h

#include <winrt/Windows.Foundation.h>
#include <winrt/Windows.Globalization.h>
#include <stdio.h>

// DateTime2.cpp

#include "pch.h"

using namespace winrt::Windows::Foundation;
using namespace winrt::Windows::Globalization;

int main() {
    winrt::init_apartment();

    {
        Calendar calendar;
        calendar.SetToNow();
        printf("The time is %02d:%02d:%02d\n",
            calendar.Hour(), calendar.Minute(), calendar.Second());
        printf("Time zone: %ws\n",
            calendar.TimeZoneAsString().c_str());
    }
    winrt::uninit_apartment();
}
```

A few remarks are in order:

- The C++/WinRT types are part of the `winrt` namespace, followed by the subnamespaces as needed.
- The includes are under a *winrt* folder, followed by the full namespace name (like

`Windows.Globalization`). Note that you can use a forward slash or a backslash for separating directories.

- All the C++/WinRT code is in header files only. There are no cpp files anywhere, so there is no "runtime" for C+=/WinRT.
- `HRESULT`s are not exposed directly. Instead, any failed `HRESULT` is translated to a C++ exception (several types are defined and reported based on the error encountered).

- Objects are created directly (most of the time), no need to use interfaces explicitly in most cases. The metadata upon which these C++ types are generated contains the required information to synthesize the types correctly.
- `winrt::init_apartment` is used to initialize COM/WinRT on the thread (internally calls `CoInitializeEx`).

Note the `TimeZoneAsString` method of the `Calendar` class. In the original pure C++ code, that method was part of another interface implemented by the Calendar - `ITimeZoneOnCalendar`. Also, if you look back at the original example, the method name is `TimeZoneAsFullString`, rather than `TimeZoneAsString`. This is because the projection is using two overloaded methods - they have the same name but different parameters. In classic COM, this is not supported, and technically nor is that supported in WinRT. But because of the metadata indicating this is supposed to be an overload, the C++/WinRT projection can accomodate. Languages that don't support method overloading wouldn't use that capability (e.g. Rust).

Let's take the second example of looking at network information. Here is the equivalent C++/WinRT code (in the *NetworkInfo* sample project):

```
// pch.h

#include <winrt/Windows.Foundation.h>
#include <winrt/Windows.Foundation.Collections.h>
#include <winrt/Windows.Networking.Connectivity.h>
#include <stdio.h>

// main.cpp

#include "pch.h"

using namespace winrt;
using namespace winrt::Windows::Foundation;
using namespace winrt::Windows::Networking::Connectivity;

int main() {
    init_apartment();

    for (auto profile : NetworkInformation::GetConnectionProfiles()) {
        printf("Name: %ws\n", profile.ProfileName().c_str());
    }

    return 0;
}
```

Compare that with the *NetworkInfoRaw* project code. Notice the simple call to the static `NetworkInformation::GetConnectionProfiles` method. Iterating over the returned collection

does not require any special gymnastics - a standard C++ `for` loop can be used. No need to deal with `HSTRING`s directly - the `c_str` method on the `HSTRING` wrapper provided by C++/WinRT looks like a normal C pointer. This is truly easy and understandable.

Here is yet another example, found in the *AppList* sample project, that lists the UWP application packages available to the user:

```
// pch.h
#include <winrt/Windows.Foundation.h>
#include <winrt/Windows.Foundation.Collections.h>
#include <winrt/Windows.ApplicationModel.h>
#include <winrt/Windows.Management.Deployment.h>

// main.cpp

#include "pch.h"

using namespace winrt;
using namespace Windows::Foundation;
using namespace Windows::Management::Deployment;

int main() {
    init_apartment();

    PackageManager pm;
    auto packages = pm.FindPackagesForUser(L"");
    int count = 0;
    for (auto package : packages) {
        if (package.IsFramework() || package.IsResourcePackage()
            || package.IsBundle())
            continue;

        count++;
        printf("%-40ws (%ws) %ws\n",
            package.DisplayName().c_str(),
            package.Id().FullName().c_str(),
            package.Description().c_str());
    }
    printf("%d packages found.\n", count);
    return 0;
}
```

Properties are accessed by invoking a method with the property's name (e.g. `IsFramework()`, `Id()`). For writeable properties, the same name applies, but accepts a single parameter which is the value to set.

Asynchronous Operations

One of the unique aspects of many of WinRT's APIs compared to most classic COM APIs is that they are asynchronous. This means that calling a method does not perform the operation completely before returning to the caller; rather, the operation only begin, and control returns to the caller, so the caller's thread is free to do other things while the operation is in flight.

Don't confuse asynchrony with threading. One way to accomplish asynchrony is by using threads, but it's definitely not the only way. Another very common way is by leveraging asynchronous I/O, which does not require any thread to maintain the operation, as the hardware is doing the work while the CPU can run other code. Refer to chapters 5, 6 and 11 threads and I/O (in part 1) for more information.

The asynchronous method must return some object that represents the operation that has just started. This is sometimes refered to as a "promise" or a "future". Regardless of the term, the caller needs to know when the operation completes in order to retrieve any results - at least knowing if the operation was successful. The caller does not want to block - this could be because of any number of reasons. A common reason in the world of UI is to prevent the UI thread from being put in a wait state as that would freeze the UI, making it unresponsive.

WinRT has a model for asynchronous operations based on a number of interfaces, the most fundamental one being `Windows.Foundation.IAsyncAction` (with several derived interfaces). Here is the definition of `IAsyncAction` in IDL and pure C++ (namespaces removed for clarity):

```
[contract(Windows.Foundation.FoundationContract, 1.0)]
[uuid(5A648006-843A-4DA9-865B-9D26E5DFAD7B)]
interface IAsyncAction : IInspectable requires IAsyncInfo {
    [propput]
    HRESULT Completed([in] AsyncActionCompletedHandler* handler);
    [propget]
    HRESULT Completed([out, retval] AsyncActionCompletedHandler** handle\
r);
    HRESULT GetResults();
}
```

```
struct IAsyncAction : IInspectable {
  virtual HRESULT __stdcall
    put_Completed(IAsyncActionCompletedHandler* handler) = 0;
  virtual HRESULT __stdcall
    get_Completed(IAsyncActionCompletedHandler** handler) = 0;
  virtual HRESULT __stdcall GetResults(void) = 0;
};
```

The `requires IAsyncInfo` directive indicates that any object implementing `IAsyncAction` must also implement `IAsyncInfo`. Here is its IDL definition for completeness:

```
[
    object,
    uuid(00000036-0000-0000-C000-000000000046),
    pointer_default(unique)
]
interface IAsyncInfo : IInspectable {
    // Properties
    [propget] HRESULT Id([out, retval] unsigned __int32 *id);
    [propget] HRESULT Status([out, retval] AsyncStatus *status);
    [propget] HRESULT ErrorCode([out,retval] HRESULT *errorCode);

    // Methods
    HRESULT Cancel();
    HRESULT Close();
}
```

The `Completed` property allows the caller to specify a callback (called *delegate* in WinRT), to be invoked once the asynchronous operation completes. Although this might sound reasonable, it's not that easy in practice, because the function's state (e.g. local variables) may not be available when the callback is invoked. Here is a conceptual example:

```
void DoWork(int value) {
    // <local variables>
    auto action = DoSomethingAsync(value);
    action.Completed(MyCompletion);
    // do other work or return to the caller
}

void MyCompletion(IAsyncAction* action) {
    // get results...
    // local variables from the DoWork function are long gone
}
```

You may think this could be made better by using a C++ lambda function, but that isn't helping much. You could capture variables by value, but that would give you copies which may not be good enough as these might be shared with the function by the caller. Capturing by reference is generally unsafe, since the function is returning, destroying all local variables:

```
void DoWork(int value) {
    // <local variables>
    auto action = DoSomethingAsync(value);
    action.Completed([&](auto action) {
        // accessing the local variables is unsafe
        // and would likely cause a crash
    }]);
    // do other work or return to the caller
}
```

> There is another question that may come up: on which thread is the callback to be invoked? The answer to that depends on the asynchronous method implementation. If the call originates from a UI thread, then most asynchronous methods will invoke the callback on the UI thread. This can be achieved by leveraging the message queue that every UI thread has. In other cases, the callback is likely to be invoked on a thread from a thread pool.

The net result of the above discussion is that supporting asynchronous operation is not easy. In C++/CX, the *Concurrency Runtime* (described in chapter 10) was used with the `task<>` class and friends to make this a little easier, but it wasn't ideal. Fortunately, C++/WinRT provides a great solution by utilizing a C++ feature called *coroutines*.

Coroutines have been discussed in the C++ community for years, but the feature itself was standardized only with C++ 20. Still, Microsoft implemented those in its compiler a few years before C++ 20 was released, so that working with asynchronous operations would be relatively straighforward.

A full discussion of C++ coroutines is beyond the scope of this chapter, as it's a C++ feature and unrelated to System Programming. We'll learn just enough to use it.

The consumer using a coroutine uses the `co_await` C++ keyword to return to its caller while preserving state, and the next instruction to execute in its own function happens once the asynchronous operation completes.

Readers familiar with C# may recognize this pattern when working with the `await` keyword, introduced in C# 5. In essence, it's the same idea.

To demonstrate using coroutines, we'll extend the *NetworkInfo* sample to extract some information that uses asynchronous calls. Specifically, We'll call the `IConnectionProfile2::GetNetworkUsageAsync` method to return statistics about the network profile usage. This is an asynchronous method only, which is very typical of WinRT APIs.

The guideline used by the WinRT designers is that if a method *might* run for more than 50 milliseconds, it should be made asynchronous.

Here is what that method looks like in pure C++ (namespaces removed for clarity):

```
virtual HRESULT __stdcall GetNetworkUsageAsync(
    DateTime startTime, DateTime endTime,
    DataUsageGranularity granularity, NetworkUsageStates states,
    __FIAsyncOperation_1___FIVectorView_1_Windows__CNetworking__\
        CConnectivity__CNetworkUsage** value) = 0;
```

That last parameter monstrosity is something like `IAsyncOperation<IVectorView<NetworkUsage>>`, meaning it represents an asynchronous operation, whose result is `IVectorView<NetworkUsage>`. It looks scary because it had to be generated by the MIDL compiler, as there are no generics or templates in COM, so these are "simulated" by creating separate interfaces for each type used.

Fortunately, with C++/WinRT we don't have to work with the `IAsyncOperation<>` interface directly - no need to register a callback with the `Completed` property either.

Currently, the example from the *NetworkInfo* project looks like this:

```
int main() {
  init_apartment();

  for (auto profile : NetworkInformation::GetConnectionProfiles()) {
    printf("Name: %ws\n", profile.ProfileName().c_str());
  }

  return 0;
}
```

Let's now add to the loop a call to `GetNetworkUsageAsync` and use `co_await` to "strip" the `IAsyncOperation<T>`, leaving the result (T) in our hands. First, we'll add some `using` namespaces to simplify the code:

```
using namespace std::chrono;
using namespace std::literals::chrono_literals;
```

Before starting the loop, we'll initiate some variables to be used inside the loop:

```
auto now = clock::now();
NetworkUsageStates states{
    TriStates::DoNotCare, TriStates::DoNotCare
};
```

Our purpose would be to get usage statistics for the network profiles in the last 48 hours. For that, we need the current time, captured using the static `clock::now` method, returning a `time_point`. It uses the same types as defined by the standard C++ library. The second line initializes a `NetworkUsageStates` WinRT structure, which is not the same as a WinRT class. You can think of a structure like a C struct - it has data members only. Here is its simplified definition:

```
struct NetworkUsageStates {
    TriStates Roaming;
    TriStates Shared;
};

enum TriStates : int {
    TriStates_DoNotCare = 0,
    TriStates_No = 1,
    TriStates_Yes = 2,
};
```

It has two data members, `Roaming` and `Shared`, both of which are of type `TriStates` (an enum). The code initializes both members to `TriStates::DontCare`. Refer to the documentation for these types for more details on their meaning.

Now we can start the loop as before:

```
auto profiles = NetworkInformation::GetConnectionProfiles();
for (auto profile : profiles) {
    printf("Name: %ws\n", profile.ProfileName().c_str());
```

The new code is for obtaining the network usage for the current network profile:

```
auto usages = co_await profile.GetNetworkUsageAsync(now - 48h, now,
    DataUsageGranularity::PerDay, states);
```

The time is selected from 48 hours ago (`now - 48h`, which is intuitive to write thanks to the chrono literals support), until `now`, and the data we want is on a per day basis, with all the states (roaming and shared). The return value from the `GetNetworkUsageAsync` method is an `IAsyncOperation<IVectorView<NetworkUsage>>` - but instead of using the `Completed` property and such we simply `co_await` it. This cause the function to return to its caller, while somehow preserving the current context. When the operation completes, the following code executes:

```
for (auto usage : usages) {
    auto seconds = static_cast<unsigned>(
        duration_cast<std::chrono::seconds>(
            usage.ConnectionDuration()).count());
    if (seconds) {
        printf("Bytes Received = %llu\n", usage.BytesReceived());
        printf("Bytes Sent = %llu\n", usage.BytesSent());
        printf("Connection Duration = %u seconds\n\n", seconds);
    }
}
```

You can think of a `co_await` as "stripping away" the asynchronous interface, leaving only the result (`IVectorView<NetworkUsage>` in our case).

The code dissects the usages `IVectorView<>` and displays some information. Closing the outer loop and compiling will lead to a compiler error, saying something like *'main': cannot be a coroutine*. The problem is that `main` returns an `int`, but a function where `co_await` is used cannot return something like an `int`. Why?

Remember that calling `GetNetworkUsageAsync` returns an object that represents an ongoing operation and that `co_await` returns to its caller (really!). So how can it return an `int` when the operation in question has just started? In general, functions that have `co_await` can only return certain types that represent ongoing operations. This whole thing is comprised of some "magic" provided by the compiler, coupled with implementations of the async interfaces provided by WinRT.

The interested reader should search the web or modern C++ to understand exactly how coroutines work.

The net effect of all this is that the code above must move to its own function that must return one of the interfaces representing the operation. For out case, everything is done in that function, so it can return the basic interface `IAsyncAction` that represents an operation with no results:

```
IAsyncAction GetData() {
  auto now = clock::now();
  NetworkUsageStates states{
      TriStates::DoNotCare, TriStates::DoNotCare
  };

 for (auto profile : NetworkInformation::GetConnectionProfiles()) {
      printf("Name: %ws\n", profile.ProfileName().c_str());
      auto usages = co_await profile.GetNetworkUsageAsync(
```

```
            now - 48h, now, DataUsageGranularity::PerDay, states);
        for (auto usage : usages) {
            auto seconds = static_cast<unsigned>(
                duration_cast<std::chrono::seconds>(
                    usage.ConnectionDuration()).count());
            if (seconds) {
                printf("Bytes Received = %llu\n",
                    usage.BytesReceived());
                printf("Bytes Sent = %llu\n", usage.BytesSent());
                printf("Connection Duration = %u seconds\n\n", seconds);
            }
        }
    }
}
```

Notice in actuallity we don't return anything from the `GetData` function, but the compiler is still happy! That's because the compiler generates an object implementing `IAsyncAction` and returns that object upon the first `co_await` encountered in the function.

We can presumably invoke this function from `main` like so:

```
int main() {
    init_apartment();
    GetData();
    return 0;
}
```

Now the code compiles. If you try it, you'll find that the first network profile name is displayed, and that's it - the process exits. Can you explain why?

When the first `co_await` is encountered, the function returns to `main`, while the asynchronous operation (`GetNetworkUsageAsync`) is done asynchronously in some way (maybe with a separate thread). Regardless, `main` then simply returns, which causes the process to exit, because we know that the C runtime calls `ExitProcess` when the main thread returns. So we don't wait long enough for the results to come in.

In this simple console application, there is nothing for the app to do while waiting for the results, so for that we can call the `get` function on the `IAsyncAction` implemnentation to force the thread to wait for the operation to complete:

```
int main() {
    init_apartment();
    GetData().get();
    return 0;
}
```

Now it works as expected (you'll note that it takes time to get this stats back, hence the reason for the method `GetNetworkUsageAsync` to designed as asynchronous). Here is a sample partial output:

```
Name: Ethernet 2
Bytes Received = 1026003284
Bytes Sent = 1095498809
Connection Duration = 34293 seconds

Bytes Received = 17941089373
Bytes Sent = 2436644814
Connection Duration = 38960 seconds
```

What about the threads this code executes on? We can add calls to `GetCurrentThreadId` before and after the `co_await` like so (add an include for *Windows.h*):

```
printf("Before co_wait TID: %u\n", ::GetCurrentThreadId());
auto usages = co_await profile.GetNetworkUsageAsync(now - 48h, now,
    DataUsageGranularity::PerDay, states);
printf("After co_wait TID: %u\n", ::GetCurrentThreadId());
```

Here is an example output:

```
Name: Ethernet 2
Before co_wait TID: 39448
After co_wait TID: 48708
Bytes Received = 1026003284
Bytes Sent = 1095498809
Connection Duration = 34421 seconds

Bytes Received = 17941433021
Bytes Sent = 2439544702
Connection Duration = 38952 seconds

Name: Profile1
Before co_wait TID: 48708
After co_wait TID: 49156
Name: Profile2
```

```
Before co_wait TID: 49156
After co_wait TID: 48708
Name: Profile3
Before co_wait TID: 48708
After co_wait TID: 49156
Name: Profile4
Before co_wait TID: 49156
After co_wait TID: 48708
```

Notice the thread change after the `co_await`. The thread then continues to work normally until it encounters `co_await` again in the next loop iteration. After the `co_await`, the thread continuing on is different. You might get more insight if you debug this application by placing breakpoints before and after the `co_await`.

If you try the same thing with a UI application, you'll find that the code returns to the same thread.

Write a small dialog based WTL application that makes these calls and observe the threads and code behavior.

Other Projections

Microsoft provides other projections for WinRT:

- .NET languages (most commonly C#).
- JavaScript, to allow developers with a JS background to access WinRT.
- Rust, which is a recent addition to the languages that can consume WinRT in an idomatic way.

More language projections can be added by utilizing the full metadata provided by WinRT types (either from Microsoft or any third party WinRT types).

The most common projection used (at the time of this writing) is for C#. This is because C# is the most common language to write UWP applications with. With C#, working with WinRT types looks almost identical compared to working with .NET types, and in most cases it is identical. For example, C# can use the `await` keyword to await WinRT asynchronous operations without blocking just as it does for normal .NET asynchronous operations based on the .NET `Task` class. In fact, helpers exist to create a `Task` given an `IAsynOperation<>` or one of its variants.

Summary

The Windows Runtime is an unmanaged platform for consuming and developing components. Maybe someday it will completely replace the classic Windows API we've been using in all other chapters in the book. But we are not there yet, and maybe never will be.

Initially, WinRT was limited to be used in UWP applications, with only a handful of types supported for Desktop applications. Today, however, most of the WinRT APIs can be used in Desktop applications with no restrictions.

Chapter 23: Structured Exception Handling

Exceptions represent (relatively) unexpected conditions occurring at run-time. Without proper handling, an unhandled exception would cause the process to terminate. Windows provides a mechanism known as *Structured Exception Handling* (SEH), that provides a way to tackle these exceptions.
Before looking at exceptions, however, we'll take a quick look at termination handlers, that allow code to run no-matter-what to perform some sort of cleanup in the (possible) face of exceptions.

Both SEH and termination handlers are exposed through the compiler with the help of four new custom keywords, added by Microsoft to the C language, so that no assembly language would not be required.

In this chapter:

- **Termination Handlers**
- **Exception Handling**
- **Exception Information**
- **Unhandled Exceptions**
- **Vectored Exception Handling**
- **Software Exceptions**

Termination Handlers

Termination handlers deal with cases where some cleanup code must execute, regardless of how the code path exits the current scope/function. A common example is the case of acquiring a synchronization primitive, such as a critical section or a mutex. Here is a basic example:

```
CRITICAL_SECTION cs;

void DoWork() {
    // do some work

    ::EnterCriticalSection(&cs);
    // do work while holding the critical section

    // release the critical section
    ::LeaveCriticalSection(&cs);
}
```

This code example should be familiar, We've seen similar examples in chapters 7 and 8 (in part 1).

The above code is not fully safe, however. What happens if there is a `return` statement between the calls to `EnterCriticalSection` and `LeaveCriticalSection`? The critical section would remain acquired, causing a deadlock, as other threads in the process would wait indefinitely for it to be released. This may seem like a silly oversight on the part of the developer, but with non-trivial code paths (conditions, loops, etc.), it's an easy mistake to make. The code might get more complex to make sure that such mistakes don't happen.

The second issue is more problematic: what if an exception is raised in between the critical section calls? The system would search for a handler in the calling function (more on that in the next section), and even if found, the call to `LeaveCriticalSection` would not run, causing the same problem. This is unrelated to how the code is structured.

One solution provided by Windows is to use a termination handler, like so:

```
void DoWork() {
    // do some work

    ::EnterCriticalSection(&cs);
    __try {
        // do work while holding the critical section
        // no matter how the body is exited, the __finally block
        // is guranteed to be invoked
    }
    __finally {
        // release the critical section
        ::LeaveCriticalSection(&cs);
    }
}
```

`__try` and `__finally` are two of the keywords added to the Microsoft C/C++ compiler. Whatever happens within the `__try` block, the `__finally` block code is guaranteed to execute. Even `return` or `goto` statements cannot change that - this is the point of having a `__finally` block, after all.

Why not put the `EnterCriticalSection` call inside the `__try` block? If `EnterCriticalSection` fails for whatever reason, no call should be made to `LeaveCriticalSection`. Of course, this guideline applies to any pair of complementary functions.

One downside of `__try/__finally` (and `__try/__except` described in the next section), is that you cannot use stack-based objects that have destructors. If this is necessary, and the objects in question are outside the block, you can try to factor out the `__try/__finally` block to a separate function.

Termination handlers are important for code written in C, but with C++ there is a better way, which we have used many times in this book: RAII.

The `AbnormalTermination` function (actually, an intrinsic) can be called in the `__finally` block to test whether `__finally` was invoked normally, or because of an exception.

Replacing Termination Handlers with RAII

Termination handlers are not very convenient to use with C++ code. One reason has to do with the limitation of not being able to use stack-based objects that have destructors. The solution is the C++ idiom *Resource Acquisition Is Initialization* (RAII) used throughout these books. A wrapper class would be used to force releasing the critical section when the wrapper object goes out of scope:

```
struct CriticalSectionLocker {
    CriticalSectionLocker(CRITICAL_SECTION& cs) : _cs(cs) {
        ::EnterCriticalSection(&cs);
    }
    ~CriticalSectionLocker() {
        ::LeaveCriticalSection(&_cs);
    }
    CRITICAL_SECTION& _cs;
};

void DoWork() {
    // do some work

    {
        // artificial scope to limits the CS acquisition to a minimum
        CriticalSectionLocker locker(cs);
        // do work while holding the critical section

        // CS released here
    }
}
```

Even if an exception occurs while the critical section is held, the stack unwinding first causes the `CriticalSectionLocker` destructor to be called, releasing the critical section, before the exception handling mechanism traverses the call stack, looking for an exception handler.

Exception Handling

Most Windows API functions return some value indicating the success or failure of the call. It's up to the caller to check the return value and handle any error appropriately. The exception mechanism common in various languages, such as C++, C# and Java, makes handling errors somewhat easier, as the code can simply use a "happy flow" approach, where no error checks are performed, but if an error does occur, the function would "throw" (or "raise") an exception that can be caught and handled in a dedicated block of code. In this way, the code doing actual work is shorter and cleaner, as there are no checks for failures. Furthermore, the error handling code is bundled in its own block rather than interspersed with the rest of the code. Figure 23-1 shows this idea.

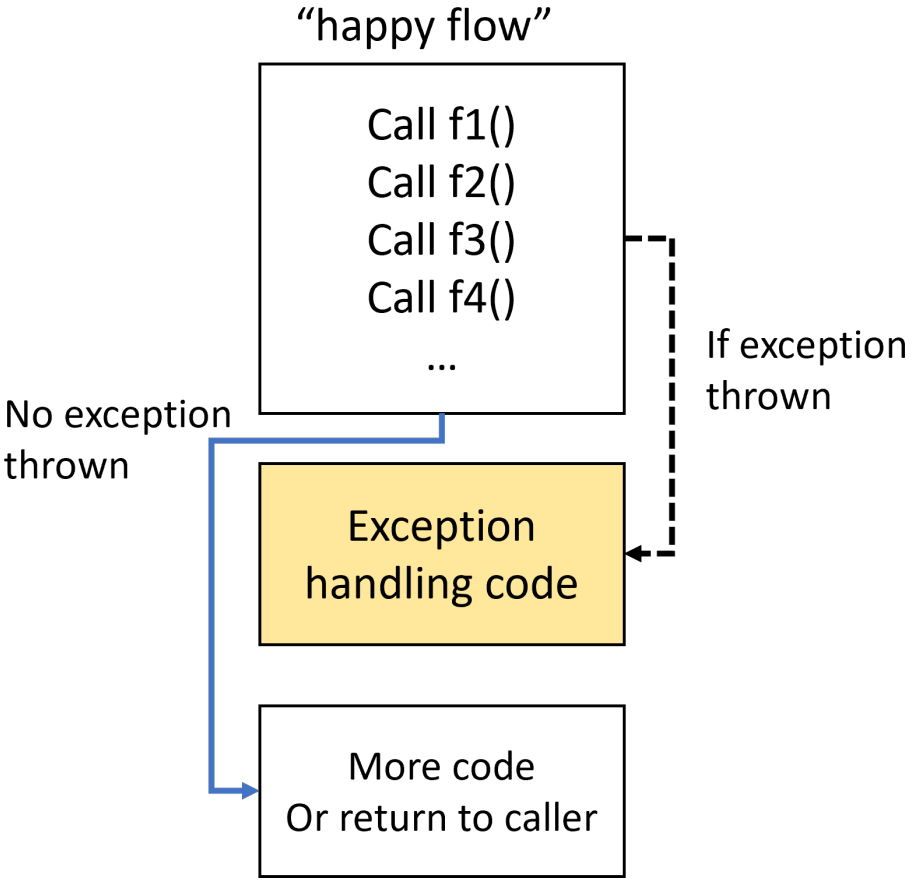


Figure 23-1: "Happy flow" and Exceptions

Windows provides a similar mechanism with the same idea in mind called *Structured Exception Handling* (SEH), which allows code to handle hardware or software exceptions. Hardware exceptions are generated by a CPU, when it encounters an instruction that cannot be executed, such as dividing by zero, accessing an unmapped memory address, or executing an invalid instruction. Software exceptions can be defined by developers, as we'll see in the section "Software Exceptions", later in this chapter.

Once a hardware or software exception occurs, the system tries to locate a handler for the exception. Here is the basic structure of a potential exception handling code:

```
__try {
    // code that may raise an exception
}
__except(filter) {
    // exception handler
}
```

A `__try` block must be followed by an `__except` or `__finally` block. Multiple `__except` blocks are illegal, just as is a `__try` / `__except` / `__finally` blocks trio. It is possible, however, to nest `__try` / `__except` and `__try` / `__finally` blocks to any level, so the above restrictions can be easily circumvented.

If an exception is raised within the `__try` block (sometimes called *guarded block*), the system checks the *filter* value to determine what should happen next (the code following the raised exception in the `__try` block to the end of the `__try` block is not executed). *filter* can be one of the following constants, or a function call returning one of these constants:

- `EXCEPTION_EXECUTE_HANDLER` (1) - execute the code in the `__except` block, and the exception is considered handled. Code resumes execution at the point after the `__except` block.
- `EXCEPTION_CONTINUE_SEARCH` (0) - don't run the `__except` block. Instead, look for an exception handler in an outer `__except` block (if any), or perform stack unwinding and look for a handler up the call stack.
- `EXCEPTION_CONTINUE_EXECUTION` (-1) - Run the instruction that caused the exception again. The assumption is that the filter function performed some operation to "fix" the condition that caused the exception in the first place, and the CPU should now attempt to execute the same instruction again.

Simple Exception Handling

The following example shows code that handles any and all exceptions that may be generated within the guarded block:

```
int Sample1(int x) {
    int y;
    __try {
        x++;
        y = 10 / x;
        printf("y = %d\n", y);
    }
    __except(EXCEPTION_EXECUTE_HANDLER) {
        printf("Handling exception 0x%X\n", GetExceptionCode());
    }
    return y;
}
```

Here is some simple code to test the `Sample1` function, followed by the resulting output:

```
printf("Result: %d\n", Sample1(1));
printf("Result: %d\n", Sample1(-6));
printf("Result: %d\n", Sample1(-1));
```

```
10 / 2 = 5
Result: 5
10 / -5 = -2
Result: -2
Handling exception 0xC0000094
Result: 0
```

With `x` passed as `-1`, the code raises a divide by zero exception. The exception code can be retrieved by calling `GetExceptionCode`, which is a *compiler intrinsic*, meaning it is translated to some machine instruction(s). This one in particular can only be invoked in the `__except` expression or its body. For integer divide by zero, the exception value is `0xc0000094` (`EXCEPTION_INT_DIVIDE_BY_ZERO` defined along with other exception codes in *winnt.h*).

The `__except` block in `Sample` is too generic - it handles any and all exceptions. Usually, the application might "excect" certain types of exceptions, but what if a different one is raised? Swallowing the exception may seem like a good idea because the process won't crash and continue running, however, could be worse. Think about it: something went wrong and the code marches on as if all is well. It's better to crash if there is no handler for the exception raised.

The `Sample2` function below is a better option, only handling a divide by zero. For all other exceptions it returns `EXCEPTION_CONTINUE_SEARCH` to continue up the call stack looking for a handler that is willing to take responsibilty for that exception (if any):

```
int Sample2(int x) {
    int y = 0;
    __try {
        x++;
        y = 10 / x;
        printf("10 / %d = %d\n", x, y);
    }
    __except (GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO
        ? EXCEPTION_EXECUTE_HANDLER
        : EXCEPTION_CONTINUE_SEARCH) {
        printf("Handling divide by zero exception only\n");
    }
    return y;
}
```

The `__except` block uses the ternary operator to return the correct value. If the decision whether to handle the exception or not is more complex, it's better to forward it to a filter function, as shown in the following example:

```
int CanHandle(DWORD code) {
    // more elaborate code...
    if (code == EXCEPTION_INT_DIVIDE_BY_ZERO
        || code == EXCEPTION_FLT_DIVIDE_BY_ZERO)
        return EXCEPTION_EXECUTE_HANDLER;

    return EXCEPTION_CONTINUE_SEARCH;
}

int Sample3(int x) {
    int y = 0;
    __try {
        x++;
        y = 10 / x;
        printf("10 / %d = %d\n", x, y);
    }
    __except (CanHandle(GetExceptionCode())) {
        printf("Handling divide by zero exception only\n");
    }
    return y;
}
```

Notice `CanHandle` is passed the exception code, rather than calling `GetExceptionCode` itself, as it's illegal to call this intrinsic in an arbitrary function.

Using `EXCEPTION_CONTINUE_EXECUTION`

The most common values returned from an `__except` expression are `EXCEPTION_EXECUTE_HANDLER` and `EXCEPTION_CONTINUE_SEARCH`. Returning `EXCEPTION_CONTINUE_EXECUTION` is uncommon, because it might not work the way you expect. Here is an example of a function that tries to handle a divide by zero exception by "fixing" it and retrying:

```
int x, y;

void DoWork() {
    __try {
        int z = x / y;
        printf("z = %d\n", z);
    }
    __except(GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO
        ? FixException() : EXCEPTION_CONTINUE_SEARCH) {}
}

int FixException() {
    y = 1;
    return EXCEPTION_CONTINUE_EXECUTION;
}
```

`DoWork` tries to divide `x` by `y`, and if `y` is zero, the division raises an exception causing `FixException` to execute and fix the issue by setting `y` to 1. This may seem reasonable, but it can cause an infinite loop. The instruction caused the esception is a machine (CPU) instruction rather than some C/C++ statement, which means the same instruction would run again.

Here is the code produced by the Visual C++ compiler in x64 Debug build for the expression `z = z / y`:

```
00007FF69C3B5A2C  mov         eax,dword ptr [x (07FF69C3BC190h)]
00007FF69C3B5A32  cdq
00007FF69C3B5A33  idiv        eax,dword ptr [y (07FF69C3BC194h)]  ; exce\
ption here
00007FF69C3B5A39  mov         dword ptr [rbp+4],eax
```

Since `y` and `x` are global variables, it's possible to divide directly by the memory address. This increases the likelihood of the "fix" to work, but doesn't guarantee it (internal CPU caching could spoil it). Even in this simple case, the code generated in x64 Release build is different:

```
00007FF6334510A4  mov         eax,8     ; called when x=8
00007FF6334510A9  cdq
00007FF6334510AA  xor         ecx,ecx
00007FF6334510AC  idiv        eax,ecx   ; exception here!
```

Now the fix is useless, because the `idiv` instruction works on `ecx` rather than the current value of `y`, causing the exception to be raised again (and again...).

You might be thinking we could fix that by declaring `y` as `volatile`. This doesn't help, because it just causes the CPU to read `y` from memory when it's needed (not relying on a value that might have been read a few instructions ago), but it doesn't prevent value from being cached in a CPU register.

In short, using `EXCEPTION_CONTINUE_EXECUTION` is discouraged, except for one scenario where it works as expected consistently. We used it in chapter 13 to commit a page in virtual memory from a reserved range when a page is accessed for the first time. Here is a similar example to the one shown in chapter 13:

```
long long cellx = 1024, celly = 1024;
int cellSize = 256;

int FixMemory(void* address) {
    printf("Committing memory at 0x%p\n", address);
    ::VirtualAlloc(address, cellSize, MEM_COMMIT, PAGE_READWRITE);
    return EXCEPTION_CONTINUE_EXECUTION;
}

void WriteString(void* buffer, int x, int y, const char* text) {
    char* p = nullptr;
    __try {
        p = (char*)buffer + (x + y * cellx) * cellSize;
        strcpy_s(p, strlen(text) + 1, text);
    }
    __except (GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION
        ? FixMemory(p)
        : EXCEPTION_CONTINUE_SEARCH) {
        // will never run
    }
}

void PrintString(void* buffer, int x, int y) {
    __try {
        printf("%s\n",
```

```
            (const char*)buffer + (x + y * cellx) * cellSize);
    }
    __except (GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION
        ? EXCEPTION_EXECUTE_HANDLER
        : EXCEPTION_CONTINUE_SEARCH) {
        printf("<memory cannot be accessed>\n");
    }
}

void MiniExcel() {
    BYTE* buffer = (BYTE*)::VirtualAlloc(nullptr,
        cellx * celly * cellSize, MEM_RESERVE, PAGE_READWRITE);

    WriteString(buffer, 10, 20, "Hello, world1");
    WriteString(buffer, 11, 20, "Hello, world2");
    WriteString(buffer, 223, 90, "Hello, world3");
    WriteString(buffer, 5, 788, "Hello, world4");

    PrintString(buffer, 223, 90);
    PrintString(buffer, 222, 90);
    PrintString(buffer, 11, 20);
    PrintString(buffer, 11, 21);
}
```

Running the `MiniExcel` function outputs the following:

```
Committing memory at 0x000001CA4B9C0A00
Committing memory at 0x000001CA4CB4DF00
Committing memory at 0x000001CA579C0500
Hello, world3

Hello, world2
<memory cannot be accessed>
```

`MiniExcel` first reserves a chunk of memory based on the number of rows, columns and cell size. Then `WriteString` is called to write some data to a cell. The data is written with `strcpy_s` directly to the address of the passed in cell. If the memory is not committed (which it never is on the first call at least), an `EXCEPTION_ACCESS_VIOLATION` exception is raised. `FixMemory` is called to fix it by committing the page where that cell happens to fall in. Finally, it returns `EXCEPTION_CONTINUE_EXECUTION` to instruct the CPU to try the failed write operation again.

`PrintString` is used to read data from a given cell. It has its own exception handler, but it does not try to commit memory if access fails. Instead, it just outputs the failed attempt. Note the empty line between "Hello World3" and "Hello world2" - this is a successful attempt at reading the memory from a cell that

was not written to, but was committed as part of an adjacent cell. "Printing" yields all zeros - an empty string.

All the examples shown are in the *SEHDemo* project, part of this chapter's samples.

Exception Information

When an exception is raised, the exception code is available via the `GetExceptionCode` intrinsic. The exception code might not be enough, however. More information on the exception (for some exceptions) is available by calling another intrinsic, `GetExceptionInformation`:

```
EXCEPTION_POINTERS* GetExceptionInformation();
```

`EXCEPTION_POINTERS` is defined like so:

```
#define EXCEPTION_MAXIMUM_PARAMETERS 15

typedef struct _EXCEPTION_RECORD {
    DWORD ExceptionCode;
    DWORD ExceptionFlags;
    struct _EXCEPTION_RECORD *ExceptionRecord;
    PVOID ExceptionAddress;
    DWORD NumberParameters;
    ULONG_PTR ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;
typedef EXCEPTION_RECORD *PEXCEPTION_RECORD;

typedef struct _EXCEPTION_POINTERS {
    PEXCEPTION_RECORD ExceptionRecord;
    PCONTEXT ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

Calling `GetExceptionInformation` is possible in a restricted context, just like `GetExceptionCode`. If you want a filter function that needs all the available information for an exception, you can create one that looks like this:

```
int MyFilter(DWORD code, PEXCEPTION_POINTERS ep) {
    //...
}
```

And call it from an `__except` block like so:

```
__except(MyFilter(GetExceptionCode(), GetExceptionInformation())) {
    //...
}
```

You can make it even simpler, passing just `GetExceptionInformation`, because the exception code is part of the first structure (`EXCEPTION_RECORD`).

`EXCEPTION_POINTERS` wraps two data structures. One is platform-independent (`ExceptionRecord`) and the other platform-dependent (`ContextRecord`). Let's look at an `EXCEPTION_RECORD` first.

`ExceptionCode` is the exception code, the same one provided with `GetExceptionCode`. `ExceptionFlags` is usually zero, but can also be `EXCEPTION_NONCONTINUABLE` (1), indicating an exception you cannot continue execution from. Any attempt to do so causes another exception to be raised, `EXCEPTION_NONCONTINUABLE_EXCEPTION`. `ExceptionRecord` may point to additional exception records, if the exception is a nested one. A nested exception may be raised if a filter function raised another exception. In most cases, the `ExceptionPointer` pointer is `NULL`.

`ExceptionAddress` is the address of the code that generated the exception. Finally, more details are possible with the `ExceptionInformation` array, where `NumberParameters` indicates the values entries in that array. For most standard exceptions, `NumberParameres` is zero, exception for the following:

- `EXCEPTION_ACCESS_VIOLATION` - the first element in the array indicates whether the access causing the exception was read (0) or write (1). If the value is 8, it means code execution was attempted in non-executable memory. The second element in the array indicates the address which failed access.
- `EXCEPTION_IN_PAGE_ERROR` - in addition to the values defined for `EXCEPTION_ACCESS_VIOLATION` above, a thirs array element indicates the underlying `NTSTATUS` code that resulted in the exception.

When defining custom exceptions, `ExceptionInformation` can be used to provide additional context (information) that may be useful to an exception handler.

The `CONTEXT` structure provided as the second member (`ContextRecord`) in `EXCEPTION_POINTERS` contains mostly CPU register data recorded at the time of the exception. Since this structure is CPU-dependent, handling its information must be done in a platform-specific manner.

> Before returning `EXCEPTION_CONTINUE_EXECUTION`, the filter function can use the `CONTEXT` structure to change the next instruction pointer (for example, the `RIP` register on x64), so that the thread can resume in a different location. Such a change is not easy to make correctly, but possible.

Here is a modified `FixMemory` function that uses some of the exception information:

```
int FixMemory2(void* address, PEXCEPTION_POINTERS ep) {
    printf("Exception accessing memory 0x%p. Operation: %s\n",
        (PVOID)ep->ExceptionRecord->ExceptionInformation[1],
        ep->ExceptionRecord->ExceptionInformation[0]
            ? "Write" : "Read");

    printf("Committing memory at 0x%p\n", address);
    ::VirtualAlloc(address, cellSize, MEM_COMMIT, PAGE_READWRITE);
    return EXCEPTION_CONTINUE_EXECUTION;
}
```

And a modified `WriteString` to invoke it:

```
void WriteString2(void* buffer, int x, int y, const char* text) {
    char* p = nullptr;
    __try {
        p = (char*)buffer + (x + y * cellx) * cellSize;
        strcpy_s(p, strlen(text) + 1, text);
    }
    __except (GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION
        ? FixMemory2(p, GetExceptionInformation())
        : EXCEPTION_CONTINUE_SEARCH) {
        // will never run
    }
}
```

Here is example output when calling `WriteString2` instead of `WriteString`:

```
Exception accessing memory 0x000001CAA96A0A00. Operation: Write
Committing memory at 0x000001CAA96A0A00
Exception accessing memory 0x000001CAAA82DF00. Operation: Write
Committing memory at 0x000001CAAA82DF00
Exception accessing memory 0x000001CAB56A0500. Operation: Write
Committing memory at 0x000001CAB56A0500
```

Passing the pointer that failed access to `FixMemory2` is not strictly necessary, as it's available in index 1 of the exception information array.

Unhandled Exceptions

What happens if an exception is not handled? The system searches up the call stack for a handler that is willing to take on the exception, but what if none is found?
This is an *unhandled exception*, that normally causes the process to crash. Before that happens, however, an optional callback is invoked that can be set up with *SetUnhandledExceptionFilter*:

```
typedef LONG (WINAPI *PTOP_LEVEL_EXCEPTION_FILTER)(
    _In_ struct _EXCEPTION_POINTERS *ExceptionInfo);

LPTOP_LEVEL_EXCEPTION_FILTER
SetUnhandledExceptionFilter(
    _In_opt_ LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter);
```

Calling `SetUnhandledExceptionFilter` replaces the current unhandled exception filter function with a new one. `SetUnhandledExceptionFilter` returns the previous handler, which could be `NULL`, meaning there was no previous handler.

Normally, the C/C++ runtime installs such an unhandled exception filter (`__scrt_unhandled_exception_filter` with the Visual Studio 2019 provided runtime), that checks if a C++ exception was thrown (more on that in the section "Software Exceptions", later in this chapter). If it's a C++ exception and it's' unhandled, it calls `abort`, which eventually calls the `UnhandledExceptionFilter` function, exported by *KernelBase.dll* (and *Kernel32.dll*).

Every thread created in a user mode process starts with some code in *NtDll* which. Here is a simplified version of that function:

```
void RtlUserThreadStart(PTHREAD_START_ROUTINE StartAddress, PVOID Argume\
nt) {
    __try {
        RtlExitUserThread(StartAddress(Argument));      // ExitThread
    }
    __except (UnhandledExceptionFilter(GetExceptionInformation ())) {
        NtTerminateProcess(NtCurrentProcess (),   // ExitProcess
            GetExceptionCode ());
    }
}
```

Every thread has a `__try`/`__except` block provided by `RtlUserThreadStart`, which means that if an exception goes unhandled, it bubbles up the call stack, eventually calling `UnhandledExceptionFilter` with the exception information. Its return value indicates how to proceed:

- `EXCEPTION_EXECUTE_HANDLER` - termination handlers are executed and the process terminates silently with the exception code as its exit code (as seem in the code above).
- `EXCEPTION_CONTINUE_SEARCH` - if a debugger is present, pass the exception to the debugger. Otherwise, continue with process termination based on configuration, as described in the next subsection "Just in Time Debugging".
- `EXCEPTION_CONTINUE_EXECUTION` - executes the instruction that caused the exception again.

This provided `UnhandledExceptionFilter` function does roughly the following:

- If the call is invoked recursively (another exception within an unhandled exception), the process terminates.
- If the process is being debugged, the debugger is invoked by returning `EXCEPTION_CONTINUE_SEARCH`.
- Otherwise, the registered unhandled exception filter is invoked, and its return value determines the final outcome.

Calling `SetUnhandledExceptionFilter` returns the previous handler, if any. In your handler, you might want to call the previous handler. Since you have no idea in the general case what this handler is doing, it might be too risky to invoke it. In case you think it's a good idea so that some "default" processing takes place, call `SetUnhandledExceptionFilter` with `NULL` before calling the original handler. This is in case the previous handler calls `UnhandledExceptionFilter` itself (which the CRT provided handler is doing) to prevent a second call to your handler. In any case, ensure you test your code thoroughly.

Just in Time Debugging

What happens when the exception is unhandled by returning `EXCEPTION_CONTINUE_SEARCH` as the final value (which is the default if no handler or filter is present)? If a *Just-in-time (JIT) debugger* is configured, it's now invoked. When working with Visual Studio, it's possible to configure it as the JIT

debugger by going to the *Tools/Options* menu and locating the *Debugging/Just-In-Time* node (figure 23-2).

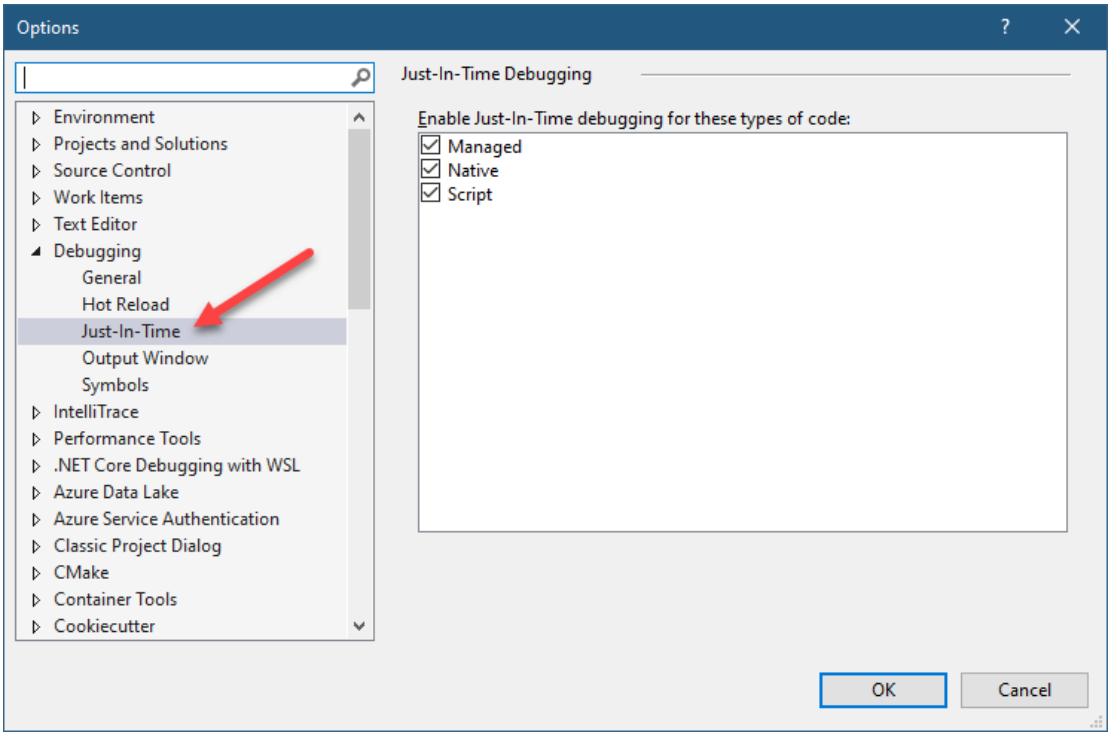


Figure 23-2: Configuring Visual Studio as the JIT Debugger

The configuration allows to select in which cases the JIT debugger should be invoked: Managed (.NET) code, native code (C/C++), or script (more options may be available depending on installed VS components and future updates).

This configuration makes changes to the Registry in the *HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug* key, as shown in a *RegEdit* window in figure 23-3:

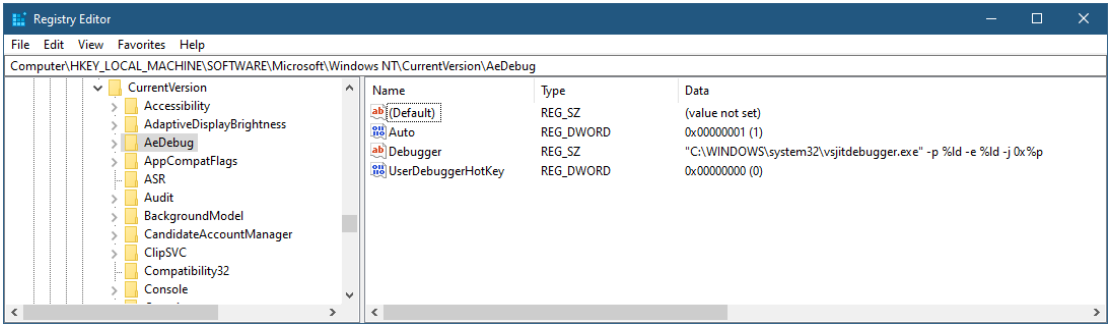


Figure 23-3: JIT Debugging configuration in the Registry

The *Debugger* value specifies the command line to execute. The Visual Studio-provided debugger is *vsjitdebugger.exe* located in the *System32* directory. The *Auto* value should be set to 1 (it's not present by default) to make sure the mechanism works, which is not done by the VS configurator. Now when a process crashes, you'll see a dialog like in figure 23-4.

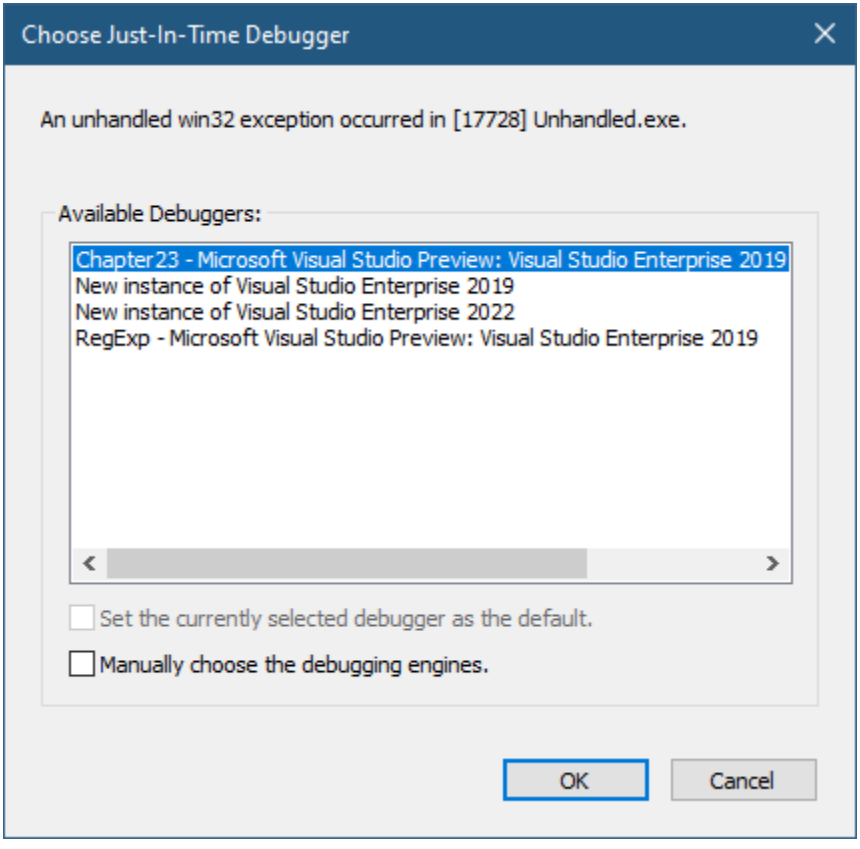


Figure 23-4: Visual Studio's JIT Debugger

You can select the debugger to use, click OK and then proceed to debug the unhandled exception.

For x86 application running on a 64-bit system, the 32-bit Registry key must be changed as well to get JIT Debugging to work: *HKLM\SOFTWARE\WOW6432Node\Microsoft\Windows NT\CurrentVersion\AeDebug*. Visual Studio configures the JIT debugger correctly, but you would need to add the *Auto* `DWORD` value and set it to 1.

Windows Error Reporting (WER)

Windows Error Reporting is a mechanism provided by Windows for collecting information when a process crashes, the kernel crashes ("Blue Screen of Death"), an application becomes unresponsive, and other issues, building a report, and optionally sending it to Microsoft or another server, for further analysis.

Full discussion of WER is outside the scope of this chapter. Here are the basics: when a process crashes (for example), the WER service is contacted by the kernel (*WerSVC*), which launches a process running *WerFault.exe* with the PID of the crashing process. *WerFault* collects the information based on configuration and optionally sends it on its way to Microsoft or another location. It's important to do all this processing in a separate process, outside of the possibly problematic state of the crashing process.

There are quite a few knobs that can be configured for WER. Two Registry keys are used to for this purpose. One is on a machine-wide basis (HKLM), and the other in the current user's hive (overriding settings from HKLM). The keys are:

- *HKLM\Software\Microsoft\Windows\Windows Error Reporting*
- *HKCU\Software\Microsoft\Windows\Windows Error Reporting*

WER does provide some programmatic control over its operation for the invoking process. More details about WER can be found in the official documentation at https://docs.microsoft.com/en-us/windows/win32/wer/windows-error-reporting.

Vectored Exception Handling

Normally, if an exception occurs, searching for handlers is performed up the call stack for the offending thread. *Vectored Exception Handling* (VEH) allows being notified of exceptions from any thread by registering a callback function.

Adding a VEH is accomplished with `AddVectoredExceptionHandler`:

```
typedef LONG (WINAPI *PVECTORED_EXCEPTION_HANDLER)(
    PEXCEPTION_POINTERS ep);

PVOID AddVectoredExceptionHandler(
    _In_ ULONG First,
    _In_ PVECTORED_EXCEPTION_HANDLER Handler);
```

`First` indicates if the handler should be added as the last in the handler list (0), or first (non zero). The handler itself must have the prototype described above by the `PVECTORED_EXCEPTION_HANDLER` type definition. The return value from `AddVectoredExceptionHandler` is an opaque value representing the registered handler (`NULL` is returned on failure). To unregister, call `RemoveVectoredExceptionHandler` with the opaque value:

```
ULONG RemoveVectoredExceptionHandler(_In_ PVOID Handle);
```

VEH handlers are invoked before normal SEH handlers. A VEH handler may be able to correct some state and allow the code to recover from the exception. For that, it should return `EXCEPTION_CONTINUE_EXECUTION`. In this case, SEH handlers are never invoked. The other option is to return `EXCEPTION_CONTINUE_SEARCH`, in which case the next VEH handler in the list is invoked (if any). If all VEH handlers returned `EXCEPTION_CONTINUE_SEARCH`, SEH handlers are now invoked as described in previous sections. Returning `EXCEPTION_EXECUTE_HANDLER` is not allowed by VEH handlers.

There is another capability to VEH - getting notified when an unhandled exception occurs, with *Vectored Continue Handlers*. Registration and unregistration is similar to the VEH type:

```
PVOID AddVectoredContinueHandler(
    _In_ ULONG First,
    _In_ PVECTORED_EXCEPTION_HANDLER Handler);
ULONG RemoveVectoredContinueHandler(_In_ PVOID Handle);
```

The continuation handler can return `EXCEPTION_CONTINUE_EXECUTION` if it can somehow correct the exception, in which case other continuations are not invoked. If it returns `EXCEPTION_CONTINUE_SEARCH`, the next continuation handler in the list is invoked.

My experience has been that continuation handlers are invoked only if a debugger is attached, after all VEHs executed and all returned `EXCEPTION_CONTINUE_SEARCH`.

Software Exceptions

The SEH mechanism can be used with custom exceptions defined by developers. One existing example is when using heaps. Normally, calling a heap allocation API such as `HeapAlloc` (as described in chapter 13) returns `NULL` on failure, forcing the caller to check that return value. Alternatively, it's possible to ask the heap to generate an exception instead.

A newly created heap (`HeapCreate`) can specify the `HEAP_GENERATE_EXCEPTIONS` (4) flag, so that allocation failures for that heap don't return `NULL`, but raises the `STATUS_NO_MEMORY` exception instead. This could simplify client code, eliminating the need to check for errors from allocations:

```
void DoWork() {
    HANDLE hHeap = ::HeapCreate(HEAP_GENERATE_EXCEPTIONS,
        64 << 10, 1 << 20);
    __try {
        void* p1 = ::HeapAlloc(hHeap, 0, 200);  // 200 bytes
        UseBlock(p1);   // use the block in some way
        void* p2 = ::HeapAlloc(hHeap, 0, 250);
        UseBlock(p2);
        //...
    }
```

```
    __except(GetExceptionCode() == STATUS_NO_MEMORY ?
        EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {
        // handle out of memory condition for the heap
    }
}
```

Regardless of how a heap was created, `HEAP_GENERATE_EXCEPTIONS` can be passed to the `HeapAlloc` API as part of the flags argument (second parameter). This means that the default process heap (available with `GetProcessHeap`) can also use the same scheme, although the default process heap is not created with this flag present.

The same idea can be used by developers. Instead of returning a value from functions indicating error or success, the function can raise an exception, optionally providing more context information related to the error. This is the purpose of the `RaiseException` API:

```
void RaiseException(
    _In_ DWORD dwExceptionCode,
    _In_ DWORD dwExceptionFlags,
    _In_ DWORD nNumberOfArguments,
    _In_reads_opt_(nNumberOfArguments) CONST ULONG_PTR* lpArguments);
```

`dwExceptionCode` is the exception code, which could be one of the system-defined exceptions, but could be application defined. To make sure the number does not clash with existing codes, bit 29 (sometimes referred to as the "customer" bit) should be set to 1. This ensures it will not collide with a Microsoft-defined exception code.

`dwExceptionFlags` is usually zero, but could be `EXCEPTION_NONCONTINUABLE` to indicate an exception from which continuing is not allowed. `nNumberOfArguments` and `lpArguments` allow the invoker to provide more data for the exception - with an array of values that is later available in the `EXCEPTION_RECORD` structure described earlier in this chapter.

High-Level Exceptions

Languages such as C++, .NET languages, Java, and many others provide their own notion of exceptions. In these higher-level languages, exceptions are *thrown* (rather than raised) using a dedicated keyword, most often named `throw`. The thrown exception is not merely a value with optional parameters, but an *object*, whatever that means for the language/platform in question.

On Windows, these platforms leverage the SEH mechanism by raising their own custom exception with `RaiseException`, adding any relevant data that would allow that runtime to interpret the "object" thrown by the platform's code. The platform has some infrastructure code that uses `__try/__except`, looking out for that particular value. Once found, the platform can interpret the extra information provided and take the correct action.

For Microsft C++ compiler, for example, the code selected is `0xE06D7363`. For .NET, the code selected is `0xE0434F4D`.

If you disregard the `E0` part of the codes (used to make sure it does not collide with other codes), then the values selected can be read as ASCII characters: `msc` (for C++) and `COM` (for .NET). The initial name for .NET was COM 3 or something similar. The name was obviously changed, but the exception code remained.

Visual Studio Exception Settings

Combining SEH and C++ is not fully supported. Specifically, SEH cannot be used in a function that has stack-based objects for which a destructor exists. The compiler will refuse to compile the code. For example, the following code does not compile:

```
struct SomeType {
    ~SomeType() {
        printf("SomeType::~SomeType called\n");
    }
};

void DoWork() {
    SomeType obj;   // requires destruction
    __try {
        //...
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        printf("Error 0x%X\n", GetExceptionCode());
    }
}
```

It may be possible to extract the C++ specific code that may require destructors to be invoked to another function to solve this problem. Also, SEH and C++ exception handling (`try/catch`) cannot be used in the same function.

The behavior of the Visual C++ compiler as far as exceptions are concerned can be configured using the *Enable C++ Exceptions* option, available in the *C/C++ / Code Generation* node in the projet's properties (figure 23-5).

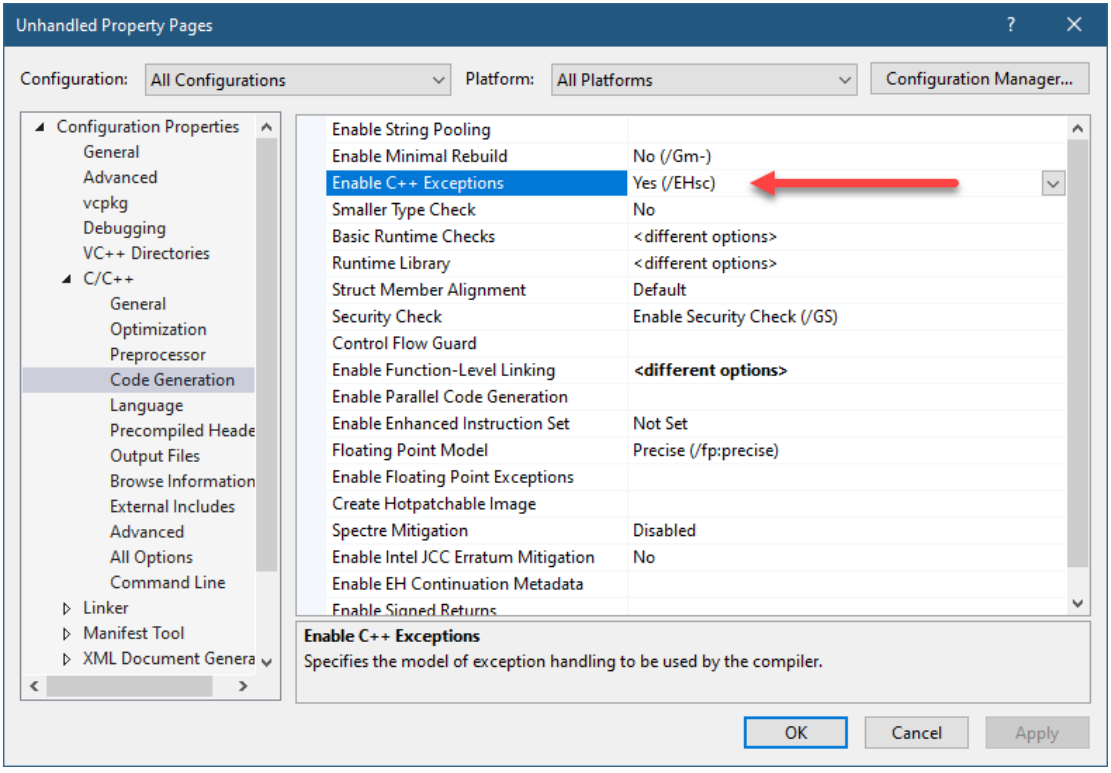


Figure 23-5: C++ exception settings

Here are the values and their meaning:

- Yes (/EHsc) (the default for new projects) - catches standard C++ exceptions. using `catch(...)` will not catch SEH exceptions. Assumes C functions marked with `extern "C"` never throw C++ exceptions.
- Yes with SEH (/EHa) - same as above, but `catch(...)` can catch SEH exceptions. Assumes `extern "C"` functions may throw C++ exceptions. I recommend using this option.
- Yes with Extern C functions (/EHs) - catches standard C++ exceptions and assumes `extern "C"` functions may throw C++ exceptions.
- No - No unwind semantics. Local objects destructors not called in functions where `throw` is used. Obviously, not a recommended option. Usually used if exception related options are specified using the *Command Line* node.

Summary

This chapter looked at the Windows-specific mechanism for handling and raising exceptions. Termination handlers can be used as well as a way of ensuring code gets executed in all conditions. Using SEH is necessary in cases where low-level exceptions need to be handled. In higher-level scenarios, C++ exceptions are recommended, as they can be naturally combined with any C++ code, and are standardized in C++, making writing cross-platform code easier, where needed.

Book Summary

This is the end of the *Windows 10 System Programming Part 2* book, concluding the two-part books on System Programming with C/C++. Windows 11 is coming very soon as I am writing these lines. A second edition might be written in the future, with Windows 11-specific features in mind. Still, all the material covered in these two books is applicable to Windows 11 just as it is for Windows 10. Windows 11 is mostly about new branding, some user-facing features, and useful applications out of the box - but for the System programmer, it's essentially identical to Windows 10.