Get Hex Dumps of Files in PowerShell

Use this script to work some magic

Bill Stewart | Mar 25, 2013

Downloads
143729.zip

A number of years ago, I had an old MS-DOS utility (the exact name of which I can no longer remember) that let me perform a hexadecimal dump of a file. That is, the utility output the hex value of each byte in the file and the ASCII characters for each byte (if printable). This kind of output can be helpful for examining the actual contents of a file, byte by byte. For example, you can see printable strings inside binary files, and you can examine the bytes of a text file.

Some programs can open a file in binary or hex mode. Figure 1 shows an example of a file opened in a text editor that offers hex mode. The gray area on the left is the offset within the file (in hex); the bytes in the file are shown as hex values, 16 bytes per row. The right-hand side of the figure displays the printable ASCII characters for each hex value (unprintable characters are displayed as dots).

Figure 1: Binary File Opened in Hex Mode

I wanted to have this kind of output available in Windows PowerShell. That way, I could quickly perform a hex dump of a file from the command line, without needing a separate program.

Using Get-Content to Dump a File

It's possible to dump a file, byte by byte, in PowerShell, using the Get-Content cmdlet's -Encoding Byte parameter. For example, the command

```
Get-Content C:\Windows\notepad.exe -Encoding Byte
```

outputs the Windows Notepad.exe program file as an array of bytes. This is a good start: We're getting the bytes from the file. But I want to convert this array of bytes into a hex dump view, similar to the one shown in Figure 1.

One of the first problems that I noticed was that this command can perform slowly, even on relatively small files. This is because Get-Content reads the entire file into memory before it outputs the file as an array.

For improved performance, the Get-Content cmdlet offers the -ReadCount parameter, which lets you specify a buffer size. For example:

```
-ReadCount 16
```

means that Get-Content will read the file 16 bytes at a time (i.e., as a series of 16-byte arrays). If the file's size isn't a multiple of 16 bytes, then the final array will contain fewer than 16 bytes. You access each array by using ForEach-Object and the $_ variable. For example, the PowerShell code in Listing 1 outputs Notepad.exe in hex format, with each byte represented as a pair of hex digits, 16 bytes per line.

Get-Content's -ReadCount parameter certainly improves performance, but I noticed in my testing that the repeated string concatenations (at Callout A in Listing 1) slowed the script as the size of the file increased. I wanted to find out whether I could improve performance by using a different method.

A Faster Method

To read the file, I decided to use the .NET Framework System.IO.File object's OpenRead method instead of Get-Content. The OpenRead method returns a read-only FileStream object. The FileStream object's Read method can read a file a certain number of bytes at a time, similar to the Get-Content cmdlet. But rather than limit the buffer size to 16 bytes, I decided to use a larger buffer and step through it 16 bytes at a time. By using this technique, I was able to avoid string concatenations, except for the last bytes of the file (if the file's size isn't a multiple of 16 bytes). This technique is shown in Listing 2.

The script in Listing 2 opens Notepad.exe as a read-only FileStream object. The script then creates a 64KB byte array, which acts as a buffer for the FileStream object's Read method, as shown at Callout A in Listing 2. The Read method returns the number of bytes that it retrieved from the file. Next, the code steps through the buffer in 16-byte increments. The code uses the

Math object's Floor method to find out how many 16-byte chunks are in the buffer (the final call to the Read method might return less than 64KB). By using the -f operator, the script in Listing 2 then takes a 16-byte slice of the buffer and outputs these 16 bytes as hex digits.

If the Read method retrieved a number of bytes from the buffer and that number isn't a multiple of 16, then the *if* expression in the first line at Callout B in Listing 2 returns a non-zero value. In this case, the code in Listing 2 uses string concatenation, similar to the code in Listing 1, to output the file's final bytes.

Both Listing 1 and Listing 2 produce identical output. However, Listing 2 provides a performance improvement over Listing 1 because the repeated string concatenation for the $output variable happens only once for the final bytes at the end of the file, and only if the file's size isn't a multiple of 16.

There are two things missing from these sample scripts: The file offset (the gray area in Figure 1) and the ASCII representation of each byte (the right-most 16 characters in Figure 1). All I needed to do was add these two enhancements, and my script was ready for prime time. The completed script Get-HexDump.ps1, which you can download, includes these features (and some other enhancements as well).

Using Get-HexDump.ps1

The script's command-line syntax is as follows:

```
Get-HexDump.ps1 [-Path]  [-UnprintableChar ] [-BufferSize ]
```

The -Path parameter specifies the name of a file. Because this parameter is

the script's first positional parameter, the -Path name itself is optional. Wildcards aren't permitted: The script can dump only one file at a time.

The -UnprintableChar parameter specifies the output character to use for characters that aren't in the standard ASCII printable range (characters 32 through 126). The default character is a dot (.). You can use this parameter to specify a different character. If you want to use a space, specify a single space character inside single (' ') or double (" ") quotation marks.

The -BufferSize parameter lets you specify the buffer size that the script uses for reading the contents of the file. The default buffer size is 65,536 bytes (64KB). The -BufferSize parameter's argument must be a multiple of 16.

For each 16 bytes of a file, Get-HexDump.ps1 outputs a string containing the following:

- the offset within the file, in hex
- the hex values of the 16 bytes
- the ASCII character representation for each printable byte
- a placeholder character (a dot or whichever character you specify for the -UnprintableChar parameter) for each unprintable byte

The script provides the following "sanity checks" before opening the file:

- makes sure that the file exists
- checks that the file is less than 4GB (the file offset in the output goes only to 0xFFFFFFFF)
- verifies that the requested buffer size is a multiple of 16

The script uses try/catch/finally blocks for error management. The main body of code is enclosed within the *try* block. If a terminating error occurs, the *catch* block outputs the error object. The *finally* block closes the file,

regardless of whether an error occurred. The script also uses the Write-Progress cmdlet to provide a visual indication of its progress. This is particularly helpful when redirecting the script's output to another command or file.

You can experiment with the -BufferSize parameter to assess the impact on the script's performance. (The Measure-Command cmdlet is helpful here.) Note that even with a much larger buffer, performance won't increase dramatically because PowerShell is still outputting a formatted string every 16 bytes. A small buffer (on my computer, about 6KB or smaller) will make the script run more slowly. The script updates its progress bar (using Write-Progress) after each buffer read, so the progress bar will move more slowly, with a greater percentage completed each time, as you increase the buffer size. After some experimentation, 64KB seemed like a reasonable default buffer size.

Figure 2 shows an example of the Get-HexDump.ps1 script's output of the first 720 bytes (i.e., 45 x 16) of C:\Windows\Notepad.exe on my Windows 7 x64 Service Pack 1 (SP1) system.

Figure 2: Get-HexDump.ps1 Output of First 720 Bytes of Notepad.exe

One Less Limitation

PowerShell doesn't have a native cmdlet suitable for viewing the content of binary files. With the Get-HexDump.ps1 script in your toolbox, this is a limitation you no longer need to live with.

Listing 1: GetHex1.ps1

```
Get-Content "C:\Windows\notepad.exe" -Encoding Byte `
 -ReadCount 16 | ForEach-Object {
 $output = ""
 foreach ( $byte in $_ ) {
#BEGIN CALLOUT A
   $output += "{0:X2} " -f $byte
#END CALLOUT A
 }
 $output
}
```

Listing 2: GetHex2.ps1

```
$bufferSize = 65536
$stream = [System.IO.File]::OpenRead(
 "C:\Windows\notepad.exe")
while ( $stream.Position -lt $stream.Length ) {
#BEGIN CALLOUT A
 $buffer = new-object Byte[] $bufferSize
 $bytesRead = $stream.Read($buffer, 0, $bufferSize)
#END CALLOUT A
 for ( $line = 0; $line -lt [Math]::Floor($bytesRead /
 16); $line++ ) {
   $slice = $buffer[($line * 16)..(($line * 16) + 15)]
   (("{0:X2} {1:X2} {2:X2} {3:X2} {4:X2} {5:X2} ") +
   ("{6:X2} {7:X2} {8:X2} {9:X2} {10:X2} {11:X2} ") +
   ("{12:X2} {13:X2} {14:X2} {15:X2} ")) -f $slice
 }
#BEGIN CALLOUT B
 if ( $bytesRead % 16 -ne 0 ) {
   $slice = $buffer[($line * 16)..($bytesRead - 1)]
```

```
    $output = ""
    foreach ( $byte in $slice ) {
      $output += "{0:X2} " -f $byte
    }
    $output
#END CALLOUT B
  }
}
$stream.Close()
```