

[Executive Bloggers](#)[Visual Studio](#)[Application Lifecycle Management](#)[Languages](#)[.NET Framework](#)[Platform Development](#)

# Base Class Library (BCL) Blog

Base types, Collections, Diagnostics, IO, RegEx...

## Memory Mapped File Quirks [Greg]

[BCL Team](#)

6 Jun 2011 1:56 PM

2

Memory mapped files are segments of virtual memory that are directly mapped to a physical file on disk, byte-by-byte. This technology has a number of benefits over traditional stream based I/O, such as performance during random access to large files, or the ability to share the mapped memory between different threads and processes. The .NET framework provides built-in support for memory mapped files starting from version 4. In most cases using memory mapped files in .NET is as easy as using traditional files streams: just map a file into the memory, open a seekable memory stream and use it as if it was a normal file while enjoying all the performance benefits. However, there are some subtle, but important differences. One of them is that memory mapped file views must always be aligned at the boundaries of the system's virtual memory pages. An important consequence is that:

Compared to the size of the underlying file on disk, the size of a memory mapped view stream is always rounded up towards a multiple of the system's virtual memory page size. You must take care to differentiate between useful data and junk bytes used to fill up the space.

### A little background

A memory mapped file is a continuous chunk of memory in a process address space that has a direct byte-by-byte mapping to an external resource. Typically, such resource is a data file on the physical hard drive. However, it is also possible to represent device memory or any other OS object that can be accessed via a file handle as a memory mapped file structure. In Windows, memory mapped files can also be backed by the system page file rather than by a separate file system object.

In Windows, [managing memory mapped files](#) has been supported for a while (see the [function reference](#)). The .NET framework [natively supports](#) this technology starting from version 4.0 through the types in the [System.IO.MemoryMappedFiles namespace](#).

In .NET you can create a memory mapping either for an existing file on the physical hard drive by specifying its path, or for an existing mapping by referring to the map name. Alternatively, you can create a new memory mapped file and have the system automatically manage a temporary file on disk to back it up. The temp file will be automatically deleted when you dispose of the memory mapped file; this is particularly useful when using memory mapped files for inter-process communication rather than for data persistence. In any case you need to use one of the static methods in the [MemoryMappedFile class](#) to create a memory map instance.

On the OS level, in order to actually read data from or write data to a [file mapping](#) you need to first [create](#) a view (projection) of that map into your process. Luckily, .NET handles this for you and all you need to do is to create either a [MemoryMappedViewAccessor](#) or a [MemoryMappedViewStream](#) using the [CreateViewAccessor](#) or [CreateViewStream](#) methods respectively. You can then use the accessor or the stream for I/O on the memory mapped file. Particularly when using the stream, the approach is not much different from using a traditional [FileStream](#).

### Subtle differences (A): The map is not quite the same as the source

Every technology must be fully understood in order to fully realise the improvements it brings over older technologies, and this is no different for memory mapped files. Although a memory map can be usually understood as a byte-by-byte mapping of the underlying resource (file) there are some subtle, but important cases where this does not apply. It is important to be aware of these cases when working with memory mapped files.

For instance, recall that NTFS optionally supports both, compression and encryption of files. Even if compression has not been activated, NTFS can also support sparse files. The idea behind sparse files is, in a nutshell, that if a file contains a very large number of consecutive zeros, the file system performs a very simple form of compression that avoids writing out all those zeros to the disk. Instead, just the number of the zeros is written, thus saving disk space without a measurable performance cost. When the zeros are overwritten with other data, the file system replaces those file chunks with normal data segments. All this happens transparently to the layers above the file system. So, when you create a memory mapping of an encrypted, compressed or sparse file, the memory map holds the respectively decrypted, decompressed or inflated content of the file. As a result, the memory page does not correspond to the physical disk file content on a byte-by-byte level. In most cases you do not need to worry about it. However, in some cases this difference may become important, for instance when you want to copy the map contents to a file manually rather than flushing the view, or when you want to use the memory map contents to reason about the size of the file on disk.

### Subtle differences (B): Where are all those zeros coming from?

Another important difference relates to the size or length of the file on disk vs. the size of the memory map. To emphasize

the problem, let us consider an example program that reads and displays the contents of a text file that has been mapped to memory:

```
using System;
using System.IO;
using System.IO.MemoryMappedFiles;
using System.Text;

namespace Microsoft.NetFx.BCL.MSDNBlog {
class MemoryMappedViewStreamLengthDemo {

    const String dataFilePath = @"data01.txt";

    public void Exec() {

        using (MemoryMappedFile mmFile = MemoryMappedFile.CreateFromFile(dataFilePath)) {

            Stream mmvStream = mmFile.CreateViewStream();
            Console.WriteLine("Stream length: {0}", mmvStream.Length);

            using (StreamReader sr = new StreamReader(mmvStream, ASCIIEncoding.ASCII)) {

                while (!sr.EndOfStream) {

                    String line = sr.ReadLine();
                    Console.WriteLine("{0:D4} \">{1}\"", line.Length, line);
                }
            }
        }

        public static void Main(String[] unusedArgs) {

            (new MemoryMappedViewStreamLengthDemo()).Exec();
        }
    }
}
```

The input file data01.txt looks like this:

```
111
222
333
```

This file is 13 bytes long: 9 bytes are taken up by the digit characters, and 4 more by the invisible carriage return (\r) and new line (\n) characters in lines 1 and 2 (line 3 does not have them because there is no new line 4). Here is a hex-binary view of the file:

```
31 31 31 0D 0A 32 32 32 0D 0A 33 33 33
1 1 1 \r \n 2 2 2 \r \n 3 3 3
```

So naturally, we expect the following output: (note that ReadLine() discards the trailing \r and \n characters)

```
Stream length: 13
[0003] "111"
[0003] "222"
[0003] "333"
```

(Note: the prefix "[0003]" describes the length of each line. It results from the "{0:D4}"-part of the WriteLine command in the inner loop of the above example program.)

However, this is not what we get. Instead, we get this (on my machine):

```
Stream length: 4096
[0003] "111"
[0003] "222"
[4086] "333"
```

"

If we step through our demo we can quickly discover that during the last iteration the string variable `line` consists of three '3'-characters followed by 4083 null characters:

```
line = "333\0\0\0\0\0 ... \0\0\0";
```

So what is the reason for this unexpected behaviour and how can we prevent it from happening

Recall that when you create a view of a memory mapped file, Windows [aligns](#) the view to virtual memory page boundaries. E.g., if your page size is, say, 4096 bytes, a memory-map view can only be 4096, 8192, 12288, ...,  $N \times 4096$  bytes large.

As our data file is smaller than a memory page, we get a view that is as large as a memory page (if the size of virtual memory pages on your machine is not 4096, then the output will also differ accordingly). The memory-map view contains the data supplemented with zeros. When we use a stream reader to read this data, we only reach the EOF mark when we get to the end of the entire view, not to the end of the original data.

One may wish that the .NET APIs were smart enough to notice the size of the underlying file on disk and to use this information to make sure that the stream reader hits an EOF at the end of the original data. However, this is not always possible, even in principle, for three types of reasons:

First, it is not always possible to even determine the path of the backing file on disk. The `MemoryMappedFile` may have been created not by specifying a physical file, but by [referring to a previously memory-mapped file \[OS ref\]](#). In such case, it is not possible to determine the backing file path. Recall that it is also [possible](#) to create a new `MemoryMappedFile` without specifying a backing file on disk at all (by either using the system page file of a temp file) [\[OS ref\]](#).

The second reason is that even when the path of the backing disk file is known, it is not in general possible to reliably determine the subset of the memory page that represents actual data versus the subset that is just filling zeros. The memory contents may not directly correspond to the file on disk due to any of the reasons pointed out in the previous section, or simply because the memory has been modified but is not yet flushed.

Finally, the framework APIs do not have the application-specific domain knowledge about whether consecutive zeros could at all be an intended content of the mapped file or must represent filler bytes to the end of the memory page.

## Solution: Know your data

On the other hand, application logic may very well be able to easily differentiate between useful and junk data.

For instance, in our particular case we know that the actual data does not contain any null characters, so we could peek ahead and stop reading when we see null characters coming up. Another approach is to define a constant such as

```
static readonly char[] FillerChars = new char[] { '\0' };
```

and then to trim the strings as we read them in:

```
String line = rdr.ReadLine().Trim(FillerChars);
```

## Summary

Memory mapped files provide many performance benefits and can be used for inter-process communication as well as for random file access. When using memory mapped files, developers need to be aware of some quirks. One of these quirks is that the size of a memory mapped file view is always blown up to the next virtual memory page boundary. When accessing data from such memory-map views, it is the responsibility of the application developer to use the domain knowledge in order to determine what memory contents represent original data, and what are junk bytes used to fill the space up to the next virtual memory page boundary.


Tweet

0

Like

0

Share

 Save this on Delicious

## Comments

---

14 Jun 2011 5:03 AM

**Iggi**

Hi Greg,

nice article very sophisticated but easy to understand. Thx.

On the second thought it seems the api is not complete.

Ig

15 Jun 2011 6:25 PM

**BCL Team**

Hello Iggi,

&gt;&gt; On the second thought it seems the api is not complete.

I am not certain what specifically you refer to. Could you please elaborate?

Greg