.Net Memory-Mapped Files

(by John Roberts for Blayd Software)

## Contents



Note: this article relates to the .Net Framework version 4.0, some of the classes and class members referenced in this article were not available in earlier framework versions.

## Introduction

The memory-mapped file has been a feature of the Windows and other operating systems for a long time, although it tends to function under the surface as an enabling mechanism for other, more well-known technologies, rather than feature in its own right. A memory-mapped file contains the contents of a "file" in virtual memory. We have, deliberately, used quotes around the word *file* because the virtual memory may contain the contents of an actual file that persists on a disk or it may contain data that is not persisted to an actual disk file.

A persisted memory-mapped file is associated with an actual file on a disk. When the last process has finished with the file, the data is saved to the disk file. This type of memory-mapped file is suitable for working with extremely large source files and will often provide better I/O performance than more conventional file access techniques.

A non-persisted memory-mapped file is not associated with an actual file on a disk. When the last process has finished working with the file, the memory is reclaimed and the data is lost. This type of memory-mapped file is suitable for creating shared memory for inter-process communication (IPC).

A point worth noting from the previous two paragraphs is that a memory-mapped file can represent a resource that can be shared between two or more processes on the same machine. Many of the IPC technologies that developers have had and may still have a love/hate relationship with such as DDE, OLE, COM, etc. make use of memory-mapped files under the covers.

As the memory-mapped file has been a feature of Windows for so long the operating system does, as you have probably guessed, implement the APIs necessary to create and manipulate them. Therefore, as .Net Framework developers, we have had the ability to use memory-mapped files from our code since the first release of the framework through the Platform Invoke (P/Invoke) mechanism. The release of

version 4.0 of the .Net Framework has now brought the memory-mapped file into the managed code world with the inclusion of the System.IO.MemoryMappedFiles namespace and its contained types.

Typically, the, principal, types in the System.IO.MemoryMappedFiles namespace provide wrappers around underlying unmanaged API calls but this is nothing new in the .Net Framework as the unmanaged Windows API is used to provide the heavy lifting by a significant number of framework types. The important thing for .Net Framework developers is that we no longer need to make direct calls to unmanaged code in our applications, we can let the framework, transparently, handle it for us.

In this article we are going to concentrate on the use of memory-mapped files in Inter-Process Communication (IPC).

## Inter-Process Communication

In this article we are taking a simplistic approach to defining a process, we are assuming that a process is a .Net Framework assembly that has been compiled to an *.exe, typically known as an application, when used in this article, the terms *process* and *application* are synonymous. Each application has its own area of memory and that memory cannot, under normal circumstances, be accessed by another application.

An application can load other code or modules into its own area of memory by referencing and then calling other .Net Framework assemblies that have been compiled to *.dlls, our code does this all the time by making use of the framework base class libraries. Therefore sharing state and/or behaviour within a single process is very straightforward and is instinctive to most developers.

Sharing state and/or behaviour between two or more applications is less instinctive and typically is more complex than sharing state and/or behaviour within a single process. In the past we have had .Net Remoting but that is now considered to be a *legacy technology* and has been superseded by Windows Communication Foundation (WCF) which provides a service-oriented approach to developing distributed applications. Both .Net Remoting and WCF are primarily intended to facilitate communication, over a network, between two or more processes running on different machines, if all we want to do is to share some data between two or more processes running on the same machine we need a much more lightweight solution.

There are a number of techniques available to safely share data or memory between two or more processes running on the same machine and memory-mapped file I/O is one of the most versatile and popular. A memory-mapped file is a *system object* rather than a *process object*, therefore, two or more applications, on the same machine, can simultaneously map data into memory and access this memory.

When using a memory-mapped file for IPC there will, typically, be a single *source* application that creates the memory-mapped file and writes data to it and one or more *destination* applications that read the data written by the *source* application. A *destination* application, typically, does something with the data that it has read and may or may not write a response or a result to the memory-mapped file.

This is by no means the only scenario in which a memory-mapped file can be used for IPC but it does serve to illustrate another benefit of memory-mapped files, they make it possible to share larger, structured, data sets between processes. If we just need to pass a small amount of data to another process, we could, for example, use the command line and we could then get the result from the destination process exit code. However, we cannot, typically, use the command line to pass large

amounts of data nor can we easily pass structured data and if we need a complex response or result from the destination process, the exit code is not suitable.

## Sample Projects and Solution

To illustrate this article and to provide you with some concrete source code that you can see running and if required step through, we have created a Visual Studio 2010 solution named MmfSample that contains two projects named MmfSource and MmfDestination.

The MmfSource application has a UI that allows the user to create, edit and delete a set of sample tasks. When the *Run Process* button is clicked, the task information is written to a memory-mapped file and the MmfDestination application is launched to simulate the processing of a selected task. When the simulated task run has finished, the result of the run is read from the memory-mapped file and displayed in the UI.

The MmfDestination application reads the task information from the memory-mapped file and displays it in its UI to allow the user to select and configure a task to run. The user can then simulate running the selected task by clicking the *Run Task* button. The MmfDestination application then writes the details of the selected task plus the result of the task run to the memory-mapped file and exits.

All of the memory-mapped file functionality is contained in a class named MmFile and this class is used by both the MmfSource and MmfDestination projects. The MmFile class can be used to:

- create a unique name for a memory-mapped file
- create a new memory-mapped file
- open an existing memory-mapped file
- write string data to a memory-mapped file
- read string data from a memory-mapped file
- synchronize read and write access to a memory-mapped file

In the MmfSource application, the MmFile class is used to create a memory-mapped file, write the task information to the file and then to read the simulated task run result data from the file.

In the MmfDestination application, the MmFile class is used to open the memory-mapped file created by the MmfSource application, read the task information and then to write the simulated task run result to the file.

The sample code used in this article is CSharp as there is very little difference between CSharp and Visual Basic in most cases. However, the MmfSample solution is provided for download in either CSharp or Visual Basic.

You can download the sample source code used to illustrate this article using the relevant link in the Article Options section on the left-hand side of the page.

## Serializing and Deserializing Structured Data

In the source code that we are using to illustrate this article we are writing task information to a memory-mapped file which is then read by another process. The task information is contained in a class named TaskInfo which is used as a data structure i.e. it has no behaviour. For each task created by the user in the MmfSource application a new TaskInfo instance is created and this instance is added to an instance of a collection class named TaskInfoCollection.

The MmfSource application then needs to serialize the TaskInfoCollection to the memory-mapped file, therefore, we have a design decision to make, what is the best way to serialize and subsequently deserialize the TaskInfoCollection?

The easiest option is to decorate both the TaskInfo and TaskInfoCollection classes with the SerializableAttribute and then use a formatter, typically the BinaryFormatter, to serialize the collection to the memory-mapped file stream. However, there is a drawback to this approach when we are serializing in one application and deserializing in another, we will have to implement the TaskInfo and TaskInfoCollection classes in a separate (*.dll) assembly and each application will have to reference this assembly and will, therefore, have a dependency on this assembly.

Another option is to serialize the TaskInfoCollection in a way that does not impose any constraints or dependencies on the destination applications i.e. serialize using a technique that does not require the destination applications to use or even know about the TaskInfo and TaskInfoCollection classes. XML is a good solution in this type of situation as it is ideal for persisting structured data, however there are alternative solutions such as JSON for example. The main drawback to using XML is that both the source and destination applications will have to implement their own formatters to serialize and deserialize the task information data.

Which option is best? That is a bit like the proverbial question, how long is a piece of string? The answer usually boils down to expediency versus flexibility and maintainability, we would always recommend that you do not introduce dependencies if you do not have to. Therefore XML or something similar is the way to go, however, this will, almost certainly involve more work and therefore take longer to implement.

The MmfSource application, in the sample code, serializes the TaskInfoCollection using XML, however as this is just a sample we have shared the TaskInfo and TaskInfoCollection classes between the MmfSource and MmfDestination projects. We have also shared a formatter class named TaskInfoFormatter between both projects. We would not share the classes between projects in this way in a real system, honestly!

## Creating a Memory-Mapped File

To create and work with a memory-mapped file we will be using the MemoryMappedFile class which is located in the System.IO.MemoryMappedFiles namespace. The sample code has a class named MmFile that contains all of the code required to create, open, read and write memory-mapped files and the source code used in this article is taken from this class.

The source application, the MmfSource project in the sample code, creates a new memory-mapped file and serializes the task information to it. The memory-mapped file is created, in the MmFile class as follows:

```
    Create new memory-mapped file.

public void CreateNew(long fileSize)

{

   if (!_isOpen)

   {

      // Create file and mutex names...

      CreateNames();



      _mmf = MemoryMappedFile.CreateNew(_fileName, fileSize);

      _isOpen = true;

      if (_synchronized)

      {

         bool mutexCreated;

         _mutex = new Mutex(false, _mutexName, out mutexCreated);

      }

   }

}
```

The MmFile class has several fields that you will see in use in most of the sample code used to illustrate this article and most are self-explanatory. The points to note about the fields in the sample code above are that the _synchronized field is used to specify whether or not to synchronize access and the _mmf field is used to store the instance of the MemoryMappedFile class created in the CreateNew method.

The static MemoryMappedFile.CreateNew method has three overloads and the sample code above makes use of the simplest overload that takes two parameters, *mapName* (String) and *capacity* (Int64). The second overload adds a MemoryMappedFileAccess parameter and the third overload, further, adds MemoryMappedFileOptions, MemoryMappedFileSecurity and HandleInheritability parameters. Therefore, you can, if required, exercise complete control over access to the newly created memory-mapped file.

When we create a memory-mapped file using the CreateNew method we are not creating a file on the disk or loading a file from the disk, we are allocating an area of system memory which we can access using file I/O objects and techniques e.g. *stream*, *binary reader*, *binary writer*, etc.

The *mapName* parameter of the CreateNew method is the mechanism used to enable the sharing of the memory-mapped file with other processes. The source application specifies a name when creating a new memory-mapped file and any other process that wants to access the file must use the same name when opening the existing file. The MmFile class has a private helper method CreateNames that uses a Guid to create a unique file name and if required a mutex name. The MmfSource application passes the Guid, on the command line, to the MmfDestination application when it launches it.

Using a unique name for a memory-mapped file may seem unnecessary and certainly most of the examples and sample code that we have seen tend to use a hard-coded literal for the file name, however, we think that a unique name is safer. For example, using the same literal value in both the source process and the destination processes could cause problems if the user is running multiple instances of the source process. You can overcome this problem to certain extent by using the CreateOrOpen method, rather than CreateNew but that still leaves the source process prone to reading data that has not be written by one of its own, spawned, destination processes. The downside to using a unique name is that you need a way to pass the generated name to the destination processes to enable them to open the existing file. That is why the MmfSource application passes the file name Guid to the MmfDestination application(s) on the command line.

The *capacity* parameter of the CreateNew method specifies the size, in bytes, of the memory-mapped file. Getting this value right is critical and can be tricky if you have several processes writing an unknown amount of data to the memory-mapped file. Once you have created a memory-mapped file the only way to increase its capacity is to release the existing file and create a new larger one, possible but very tricky in some situations. Typically, in most real systems you will have, at least, a rough idea how much data is likely to be written to the file by other processes and should, therefore, be able to calculate the required capacity. As a rule of thumb, be generous but not wasteful when specifying the capacity.

We will be discussing synchronizing access to a memory-mapped file and therefore the use of the Mutex class in a later section so we will not go in to detail here.

A memory-mapped file uses system resources and these resources need to be released when the memory-mapped file is no longer required. When you release these resources, by disposing the MemoryMappedFile instance and what happens when you release them depends on whether you created the memory-mapped file or whether you are accessing an existing memory-mapped file.

The process that creates the memory-mapped file should maintain a reference to the file until all other processes have finished accessing it. Therefore, typically, you should not create the memory-mapped in a using statement but should explicitly call the Dispose method when your process and all other processes have finished with the file. When the creating process disposes a MemoryMappedFile instance its memory is released and therefore its data is lost.

A process that accesses an existing memory-mapped file can access the file from within a using statement and dispose of the file immediately it has finished its read or write operation. When a secondary or destination process disposes an existing memory-mapped file it just releases its connection, the memory is not released and the data is not lost. A secondary or destination process can open, read or write and then dispose an existing memory-mapped file as many times as it needs to, the memory can only be released by the creating or source process.

The MmFile class used in the sample code implements the IDisposable interface and can be used within a using statement by a destination processes or it can be disposed using either the Close method or the Dispose method by the creating or source process. Calling either Close or Dispose will dispose the MemoryMappedFile instance maintained in the _mmf field.

## Opening An Existing Memory-Mapped File

The destination application, the MmfDestination project in the sample code, opens the memory-mapped file created by the MmfSource application and deserializes the task information from it. The, existing, memory-mapped file is opened in the MmFile class as follows:

```
    Open an existing memory-mapped file.

public void OpenExisting(string fileId)

{

   if (!_isOpen)

   {

      if (string.IsNullOrEmpty(fileId))

      {

         throw new ArgumentException(RES_ERR_FILE_NAME_EMPTY, "fileId");

      }

      // Create file and mutex names from GUID...

      CreateNames(fileId);

      try

      {
```

```
            _mmf = MemoryMappedFile.OpenExisting(_fileName);


            if (_synchronized)
            {
                _mutex = Mutex.OpenExisting(_mutexName);
            }
        }
        catch (FileNotFoundException ex)
        {
            throw new InvalidOperationException(RES_ERR_NO_MMF, ex);
        }
        catch (WaitHandleCannotBeOpenedException)
        {
            // Mutex does NOT exist.
            // You may prefer to throw this exception, however,
            // this implementation continues without synchronization...
            _synchronized = false;
        }
        _isOpen = true;
    }
}
```

The MmFile class has several fields that you will see in use in most of the sample code used to illustrate this article and most are self-explanatory. The points to note about the fields in the sample code above are that the _synchronized field is used to specify whether or not to synchronize access and the _mmf field is used to store the instance of the MemoryMappedFile class created in the OpenExisting method.

The static MemoryMappedFile.OpenExisting method has three overloads and the sample code above makes use of the simplest overload that takes a single parameter, *mapName* (String). The second overloads adds a MemoryMappedFileRights parameter and the third overload, further, adds a HandleInheritability parameter.

To open a memory-mapped file that already exists i.e. has, typically, been created by another process, we need to specify the same file name that was specified when the memory-mapped file was created. The MmFile class has a private helper method CreateNames that uses a Guid to create a unique file name and if required a mutex name. The MmfSource application passes the Guid used when the memory-mapped file was created, on the command line, to the MmfDestination application when it launches it and this is the value that should be passed to the MmFile.OpenExisting method.

We will be discussing synchronizing access to a memory-mapped file and therefore the use of the Mutex class in a later section so we will not go in to detail here.

The OpenExisting method will throw an exception if a memory-mapped file with the specified name does not exist in system memory and the Mutex.OpenExisting method will similarly throw an exception if a mutex with the specified name cannot be found. Therefore, we need to catch these, potential, exceptions and have a reasonable exit strategy should they occur.

A process that accesses an existing memory-mapped file can access the file from within a using statement and dispose of the file immediately it has finished its read or write operation. When a secondary or destination process disposes an existing memory-mapped file it just releases its connection, the memory is not released and the data is not lost. A secondary or destination process can open, read or write and then dispose an existing memory-mapped file as many times as it needs to, the memory can only be released by the creating or source process.

The MmFile class used in the sample code implements the IDisposable interface and can be used within a using statement by a destination processes. Calling Dispose, either implicitly from a using statement or explicitly will dispose the MemoryMappedFile instance maintained in the _mmf field.

## Synchronizing Access To A Memory-Mapped File

When using a memory-mapped file for IPC we have two or more processes that need to access the file and each of these processes will have their own primary thread, therefore the potential for concurrent read/write access from multiple threads exists. If you have developed the code for both the source and destination processes yourself you may be in a position to guarantee that concurrent access will not occur but we would caution against taking such a view. We recommend that you always synchronize read/write access to a memory-mapped file that is being shared between two or more processes.

As stated earlier a memory-mapped file is a *system object* and can, therefore, be accessed by all processes running on the same machine. To synchronize access to a *system object* we need a synchronization object that is or can be a *system object* and for this we have opted for the Mutex class. The Mutex class, located in the System.Threading namespace, can be constructed with a *name* which will create a *system* mutex.

The use of the *name* parameter with the Mutex class works in the same way as a named memory-mapped file does i.e. the source process creates a new Mutex with a specified name and other processes call the static Mutex.OpenExisting method using the same name specified when creating the Mutex.

The source application, the MmfSource project in the sample code, creates a new memory-mapped file and by default will also create a new system mutex for synchronization. The MmFile class in the sample

code, by default, synchronizes read/write access to the memory-mapped file using the Mutex class, it can however be constructed not to synchronize access. When creating a new memory-mapped file, the MmFile.CreateNew method also, optionally, creates a new system mutex as follows:

```
    Create a new system mutex.

if (_synchronized)

{

   bool mutexCreated;

   _mutex = new Mutex(false, _mutexName, out mutexCreated);

}
```

The _synchronized field is initialized in the MmFile constructor and by default will be set to true. The _mutexName field is initialized in the private helper method CreateNames which uses a Guid to create a unique mutex name. The same Guid is used for both the memory-mapped file name and the mutex name, however, the file name and mutex name have different prefixes so that they can be easily identified.

The first, Boolean, parameter of the Mutex constructor is set to false to specify that the Mutex is not initially *owned* by the source process. This means that the WaitOne method must be called before a read/write operation to wait for the mutex to become signalled, giving the calling thread *ownership* of the mutex. The ReleaseMutex method must then be called when the read/write operation is complete.

You should note that the WaitOne method is a blocking call, if the mutex is *owned* by another thread when the call is made, the calling thread will block until the *owning* thread releases the mutex.

The MmfSource application passes the Guid, on the command line, to the MmfDestination application when it launches it.

The destination application, the MmfDestination project in the sample code, opens the memory-mapped file created by the MmfSource application and also opens the system mutex for synchronization. When opening an existing memory-mapped file, the MmFile.OpenExisting method also, optionally, opens an existing system mutex as follows:

```
    Open an existing system mutex.

if (_synchronized)

{

   _mutex = Mutex.OpenExisting(_mutexName);

}
```

The private helper method CreateNames uses the Guid passed to the MmFile.OpenExisting method to create the mutex name and initialize the _mutexName field. As with the CreateNew method the destination process does not initially own the mutex and must therefore call the WaitOne method before a read/write operation.

The following Writing To A Memory-Mapped File and Reading From A Memory-Mapped File sections show the mutex in use when writing to and reading from a memory-mapped file.

## Memory-Mapped File Reading and Writing

To write to or read from a memory-mapped file we need to create a view of either the entire file or a part of it. There are two, available, view types, stream access view represented by the MemoryMappedViewStream class an instance of which is created by the MemoryMappedFile.CreateViewStream method and random access view represented by the MemoryMappedViewAccessor class an instance of which is created by the MemoryMappedFile.CreateViewAccessor method.

Stream access view is recommended for the non-persisted file approach used in IPC and random access view is recommended for working with persisted files i.e. memory-mapped files created from disk files. As this article is concentrating on the use of memory-mapped files for IPC we will only be discussing the use of the stream access view.

The MemoryMappedViewStream class, as its name would suggest, derives, via the UnmanagedMemoryStream class, from the Stream class, located in the System.IO namespace. Therefore, once we have created an instance we can easily use that instance to construct a stream reader or stream writer and from then on we can access the memory-mapped file using familiar I/O techniques.

## Writing To A Memory-Mapped File

The MmFile class in the sample code encapsulates the functionality to write to a memory-mapped file and surfaces a method with a deceptively simple signature as follows:

```
void Write(int segment, string data)
```

The MmFile class treats each write to the memory-mapped file as a segment with the first write to the file being segment 0, the next segment 1, etc. Before writing the segment data to the file, the MmFile class writes the size (in bytes) of the data as an Int32 to the file, it then writes the data immediately after the segment size value.

The data to write to the file is passed to the Write method as a String, which keeps things nice and simple for callers but it is written to the file using an instance of the BinaryWriter class, located in the System.IO namespace and is therefore converted to a byte array, using *Unicode* encoding, before being written.

The *segment* parameter of the Write method specifies the segment number to write the data to, for example, to write to the file for the first time calling code should pass in 0 (zero) for the *segment* parameter value. The *data* parameter of the Write method specifies the string data to write to the specified segment. The data is always appended to the file or written sequentially, therefore callers

cannot write segment 1 if segment 0 has not already been written. However, segments can be written randomly if the previous segment(s) already exist in the file, for example, if the caller knows that the file already contains segment 0 and segment 1 the caller can specify 2 for the *segment* parameter in their first call to the Write method. The MmFile.Write method writes the data to the memory-mapped file as follows:

Write to a memory-mapped file.

```
public void Write(int segment, string data)
{
   // Object state and parameter validation elided for clarity.
   // ...
   if (_segments.Count == 0)
   {
      // Read any, existing, segment size data...
      InitializeSegments();
   }
   if (segment != _segments.Count)
   {
      // Can only write in segment order and cannot overwrite segment.
      throw new InvalidOperationException(RES_ERR_SEGMENT_WRITE_ORDER);
   }
   byte[] byteData = EncodeData(data);
   int dataSize = byteData.Length;
   int offset = GetCurrentOffset();
   if (_mutex != null)
   {
      _mutex.WaitOne();
   }
   try
```

```
    {
        using (MemoryMappedViewStream viewStream = _mmf.CreateViewStream(offset, 0))
        {
            using (BinaryWriter writer = new BinaryWriter(viewStream))
            {
                writer.Write(dataSize);
                writer.Write(byteData);
            }
        }
    }
    finally
    {
        if (_mutex != null)
        {
            _mutex.ReleaseMutex();
        }
    }
    _segments.Add(dataSize);
}
```

The _segments field is of type List<int> and stores the data size of each memory-mapped file segment. If there is no segment size data when the Write method is called, the helper method InitializeSegments is called to read through the memory-mapped file and load the data sizes of any existing segments. If there are no existing segments _segments.Count remains at 0 (zero).

Data is always appended to the memory-mapped file and existing segments are never overwritten, therefore the value of the *segment* parameter should always equal the number of segment data size values in the _segments field.

The string *data* is converted to a byte array, using *Unicode* encoding and the size of the data to be written is stored to a local variable. A helper method GetCurrentOffset is then called to calculate the offset within the memory-mapped file at which the data should be written. The calculation sums the segment data size values in the _segments field plus the size of an Int32 for each segment data size

written to the file. The first segment written to the memory-mapped file is always written at offset 0 (zero).

If required, the sample code then acquires *ownership* of the mutex by calling the WaitOne method. In a real application you may want to specify a timeout for the WaitOne call and take the appropriate action should the call fail. If we acquire the mutex we must release it when we have finished writing to the memory-mapped file, therefore the sample code that writes to the file is included within a try/finally block to ensure that the ReleaseMutex call is executed if required.

We then need to create a view and a writer and write the data to the memory-mapped file. Both of these operations are included within using statements to ensure that the objects get disposed when we have finished with them.

There are three overloads of the CreateViewStream method, the first overload has no parameters and creates a view of the whole memory-mapped file, the second overload, which is used in the sample code, takes two parameters, the *offset* at which the view starts and the *size* of the view. Specifying 0 (zero) for the *size* parameter creates a view that starts at *offset* and ends approximately at the end of the memory-mapped file. The third overload adds a MemoryMappedFileAccess parameter to the *offset* and *size* parameters.

We can use the CreateViewStream method to create a view of any area of the memory-mapped file and we can even create overlapping views. Once we have a view, the underlying view stream position starts at 0 (zero), therefore, in the sample code, the specified *offset* is translated in the underlying view stream to position 0 (zero).

Writing to the memory-mapped file now becomes a breeze, we use the MemoryMappedViewStream to construct an instance of the BinaryWriter class and write out the Int32 data size value and then the data byte array.

Finally the data size value is added to _segments field, so that we know where in the memory-mapped file to start writing the next data segment.

## Reading From A Memory-Mapped File

The MmFile class in the sample code encapsulates the functionality to read from a memory-mapped file and surfaces a method with a deceptively simple signature as follows:

```
string Read(int segment)
```

The MmFile class treats each read from the memory-mapped file as a segment with the first data segment in the file, segment 0, the next segment 1, etc. Immediately before each data segment in the file is an Int32 data segment size value.

The *segment* parameter of the Read method specifies the segment number to read the data from, for example, to read the first segment in the file calling code should pass in 0 (zero) for the *segment* parameter value.

Data segments should be read sequentially i.e. segment 0, segment 1, etc. however, if you know that the file contains a certain number of segments you can read any existing segment e.g. if you know that

the file contains segments 0 and 1, you can call the Read method with 1 to read the second segment first. The MmFile.Read method reads data from the memory-mapped file as follows:

```
    Read from a memory-mapped file.

public string Read(int segment)

{

   // Object state and parameter validation elided for clarity.

   // ...

   if (_segments.Count == 0)

   {

      // Read any, existing, segment size data...

      InitializeSegments();

   }

   if (segment >= _segments.Count)

   {

      throw new InvalidOperationException(RES_ERR_SEGMENT_READ_NOEXIST);

   }

   int offset = GetCurrentOffset(segment);

   int dataSize = 0;

   byte[] byteData = null;

   if (_mutex != null)

   {

      _mutex.WaitOne();

   }

   try

   {

      using (MemoryMappedViewStream viewStream = _mmf.CreateViewStream(offset, 0))

      {
```

```
        using (BinaryReader reader = new BinaryReader(viewStream))
        {
            try
            {
                dataSize = reader.ReadInt32();
                if (dataSize > 0)
                {
                    // Read the data...
                    byteData = reader.ReadBytes(dataSize);
                    if (byteData.Length < dataSize)
                    {
                        // EOS, so data is NOT as expected...
                        byteData = null;
                    }
                }
            }
            catch (EndOfStreamException)
            {
                // Data is NOT as expected...
                byteData = null;
            }
        }
    }
}
finally
{
    if (_mutex != null)
```

```
        {
            _mutex.ReleaseMutex();
        }
    }
    string data = string.Empty;
    if (byteData != null)
    {
        data = DecodeData(byteData);
    }
    return data;
}
```

The _segments field is of type List<int> and stores the data size of each memory-mapped file segment. If there is no segment size data when the Read method is called, the helper method InitializeSegments is called to read through the memory-mapped file and load the data sizes of any existing segments. If there are no existing segments _segments.Count remains at 0 (zero).

Data can only be read from the memory-mapped file if the specified *segment* exists, therefore the value of the *segment* parameter should never be greater than or equal to the value of _segments.Count.

A helper method GetCurrentOffset is called to calculate the offset within the memory-mapped file at which the data should be read. The calculation sums the segment data size values in the _segments field up to the specified *segment* plus the size of an Int32 for each segment data size value. The first segment read from the memory-mapped file is always read from offset 0 (zero).

If required, the sample code then acquires *ownership* of the mutex by calling the WaitOne method. In a real application you may want to specify a timeout for the WaitOne call and take the appropriate action should the call fail. If we acquire the mutex we must release it when we have finished reading from the memory-mapped file, therefore the sample code that reads from the file is included within a try/finally block to ensure that the ReleaseMutex call is executed if required.

We then need to create a view and a reader and read the data from the memory-mapped file. Both of these operations are included within using statements to ensure that the objects get disposed when we have finished with them.

There are three overloads of the CreateViewStream method, the first overload has no parameters and creates a view of the whole memory-mapped file, the second overload, which is used in the sample code, takes two parameters, the *offset* at which the view starts and the *size* of the view. Specifying 0 (zero) for the *size* parameter creates a view that starts at *offset* and ends approximately at the end of

the memory-mapped file. The third overload adds a MemoryMappedFileAccess parameter to the *offset* and *size* parameters.

We can use the CreateViewStream method to create a view of any area of the memory-mapped file and we can even create overlapping views. Once we have a view, the underlying view stream position starts at 0 (zero), therefore, in the sample code, the specified *offset* is translated in the underlying view stream to position 0 (zero).

In the sample code we would be able to use the data size value, in the _segments field, to calculate the size of the view, rather than specifying 0 for the *size* parameter and creating the view from *offset* to the end of the memory-mapped file. We could also specify a view *offset* that is beyond the data *size* value and save a read operation. If we were using larger data segments we would probably do this, however, in the sample the data segments are relatively small, as will often be the case in a real application and actually reading in the data size value makes sure that the format of the file is as expected.

Reading from the memory-mapped file now becomes a breeze, we use the MemoryMappedViewStream to construct an instance of the BinaryReader class and read in the Int32 data size value and then, using the data size value, read in the data byte array.

Finally the data byte array is decoded back into a *Unicode* string and returned to the caller.

## Conclusion

IPC is not a technique that you will need every day, although if you are involved in designing and/or developing multi-application systems rather than standalone applications you will almost certainly have to find a way for your applications to *talk* to each other. If your applications are running on the same machine and if your main requirement is to exchange data rather than invoke behaviour then memory-mapped files offer a, relatively, simple solution. The inclusion of memory-mapped file functionality within the .Net Framework, as from version 4.0, makes their use even easier for .Net developers, no more P/Invoke, working directly with unmanaged memory, pointers, etc. We strongly recommend that you take a look at memory-mapped files and familiarize yourself with the techniques required to get the best out of them.