Antonio Tapper

Source files
Inter Process producer-Consumer Queue
MMChannel.cs
UTCutil.cs
Enums.cs

Inter Process producer-Consumer Queue Test Harness
ConsumerTestRunner.cs
ProducerTestRunner.cs
ShutdownTestRunner.cs

MMChannel.cs

```
/*****************************************************************************************************************************************************
 * The goal of the AlphaSystematics Project is create an open-source system for forward-testing systematic strategies with live market data and trade feeds.
 * It enables strategies developed in Excel to be connected to trading venues via industry standard FIX messaging.
 *
 * Copyright (C) 2009  Antonio Tapper. www.alphasystematics.org

 * This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

 * This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more details.

 * You should have received a copy of the GNU General Public License along with this program.  If not, see <http://www.gnu.org/licenses/>.
*****************************************************************************************************************************************************/

using System;
using System.Collections.Generic;
using System.Threading;
using System.Text;
using System.Diagnostics;
using System.IO.MemoryMappedFiles;

namespace com.alphaSystematics.concurrency
{
    public unsafe struct ControlData
    {
        public bool debug;
        public bool test;
    public int ds_type;
    public int queueAddPosition;
    public int queueTakePosition;
    public int initialCount;
    public int stackAddTakePosition;
    public long totalItemsEnqueued;
    public long totalItemsDequeued;
    public bool isInitialized;
    public bool areResultsLogged;
    public int startTimeLength;
    public fixed char startTime[30];
    public long startTimeTicks;
    public int endTimeLength;
    public fixed char endTime[30];
    public long endTimeTicks;
    public long ticksPerItem;
    public int throughput;
    public long microseconds;
    public long nanoseconds;
    public long testPutSum;
    public long testTakeSum;
    public bool shutdownFlag;
    public int reservations;
}


#region Class description

// Windows Data Alignment on IPF X86 and X64 by Kang Su Gatlin. referred to on page 492 of Concurrent Programming on Windows

// MemoryMappedQueue implements a fixed-length memory mapped file based queue with blocking put and take methods
// controlled by a pair of counting semaphores.
// The ConsumerSemaphore represents the number of items that can be removed from the queue and is
// initially set to zero.
// ProducerSemaphore represents the number of items that can be inserted into the queue and is
// initially set to the capacity of the queue.

// A 'take' operation first requires that a permit be obtained from ConsumerSemaphore.
// This succeeds immediately if the queue is non-empty or blocks until the queue becomes non-empty
// Once the permit is obtained then the data structure is locked with a mutex then an
// element is removed from the head of the list and  a permit is released to the Producersemaphore.

// The 'put' operation works conversely
// On exit from either 'put' or 'take' the sum of the counts of both semaphores always equals the capacity of the buffer
// The semaphores do not have 'thread affinity' (or coherence) so the Producer Semaphore can be released by the Consumer
// thread and vice-versa but the mutex does have and so must be released by the theead that acquires it.

// The element could be guaranteed to be an object
//  We might declare the data structure to accept only objects and box any primitive if, for example, we wanted to be able
// to call Dispose on them. We would then have to box any primitive values used as elements
// See Joe Duffy's Concurrent Programming in Windows, chapter 10, Memory Models and Lock Freedom page 527
// public class MMQueueDEV<E> where E : class | public class MMQueueDEV<E> where E : struct
#endregion Class description


public class MMChannel : IDisposable
{
    // See Concurrent Programming on Windows, J.Duffy, Chap 5 Windows Kernel Synchronization, pg 225 for Queue algorithm
    #region constructor

    private static MMChannel channel;
    private readonly static object lockConstructor = new object();

    public static MMChannel GetInstance(string ipcName, int fileSize, int viewSize, int capacity,
        bool debug = false, bool test = false, DataStructureType dsType = DataStructureType.Queue)
    {
        // The mutex is used to ensure atomic
        // creation and initialization of the IPC artefacts. If another process has already acquired the mutex then the method
        // will return the artefacts created by the first and only process to create them

        // Ensure that even if more than one thread in the same process attempts to create a channel - referencing the system-wide IPC artefacts
        // only one instance will be created in a process. Not really necessary as we have a system-wide mutex and semaphores so it doesn't
        // matter if this class is instantiated more than once but seems cleaner as we also don't need more than one instance

        // Why I didn't use static lazy initialization. See Note 1. Another possible initialization method
        lock (lockConstructor)
        {
            if (channel == null) channel = new MMChannel(ipcName, fileSize, viewSize, capacity, debug, test, dsType);
        }

        return channel;
    }

    private MMChannel(string ipcName, int fileSize, int viewSize, int capacity,
        bool debug = false, bool test = false, DataStructureType dsType = DataStructureType.Queue)
    {
        // We received _ipcFileName, fileSize, viewSize, collection type, timeout and capacity in the constructor
        _fileSize = fileSize; _viewSize = viewSize; _capacity = capacity; _ipcName = ipcName;
        _dsType = dsType; _debug = debug; _test = test;
        if (ipcName.Length == 0 || _fileSize <= 0 || _viewSize <= 0 || _capacity <= 0)
        {
            string msg = string.Format("Invalid arguments (ipcName {0}, FileSize {1}, ViewSize {2}, Capacity {3}",
                ipcName, _fileSize, viewSize, _capacity);
            throw new Exception(msg);
        }
        // The capacity is the number of views, effectively elements in a queue, to be created in the file
        // so the number of elements times the size of each element must not be greater than the size of the file
        int cap_times_view = _capacity * _viewSize;
        if (_capacity * _viewSize > _fileSize)
        {
            string msg = "Invalid arguments (Capacity * ViewSize " + cap_times_view +
                            " > FileSize " + _fileSize + ") passed to Memory Mapped File constructor";
            throw new Exception(msg);
        }

        // Create the IPC artefact names by adding the pre-defined names to the user requested queue name
        _consumerSemaphoreName = IPCName + _consumerSemaphoreNameAppend;
        _producerSemaphoreName = IPCName + _producerSemaphoreNameAppend;
        _mutexLockChannelName = IPCName + _mutexLockChannelNameAppend;
        _memoryMappedDataFileName = IPCName + _memoryMappeDataFileNameAppend;
        _memoryMappedControlFileName = IPCName + _memoryMappedControlFileNameAppend;

        Start();
    }
    #endregion constructor

    #region variable declarations

    // space in a memory mapped file for the control variables
    const int CONTROL_DATA_FILE_SIZE = 1000; const int ZERO = 0;
    const int DEFAULT_TIMEOUT = System.Threading.Timeout.Infinite;

    private int _fileSize; private int _viewSize; private int _capacity; private bool _debug; private bool _test;

    // All the Inter Process artefacts need names so they can be looked up
    // Inter-process throttle on the number of items that can be enqueued.
    protected Semaphore _consumerSemaphore; private String _consumerSemaphoreName;
    protected Semaphore _producerSemaphore; private String _producerSemaphoreName;

    // guarded by the mutex _mutexLockChannel
    protected string _ipcName = "";

    // The name of the queue is passed into the constructor and either creates a new one or looks up an exisiting one
    // It is pre-pended to the semaphore and mutex names to create names that can be looked up inter process
    // These names are hidden from client programs to try to avoid accidental (or malicious) name collisions
    // Inter-process lock to guard the mutable shared state - the queue or stack
    private Mutex _mutexLockChannel; private String _mutexLockChannelName;

    // Currently the data structure can be instantiated as a queue or a stack
    protected DataStructureType _dsType;

    // Append to IPCName to form names for semaphores and mutexes
    private static string strGUID = "_{5C00361E-3C88-48A7-BB0A-F6ADF376C5A1}"; // Guid.NewGuid().ToString("N");
    private string _consumerSemaphoreNameAppend = "_consumer_" + strGUID;
    private string _producerSemaphoreNameAppend = "_producer_" + strGUID;
    private string _mutexLockChannelNameAppend = "_channel_mutex_" +strGUID;

    // Memory mapped file for IPC
    private MemoryMappedFile _memoryMappedDataFile; private String _memoryMappedDataFileName;
    private MemoryMappedFile _memoryMappedControlFile; private String _memoryMappedControlFileName;

    // Random Access views of the memory mapped data file.
    private MemoryMappedViewAccessor[] _viewAccessor;

    // Random Access views of the memory mapped control file - channel control section
    private MemoryMappedViewAccessor _controlDataAccessor;

    // Append to _ipcName to form name
    private static string _memoryMappeDataFileNameAppend = "_memoryMappedDataFileNameAppend_" + strGUID;
    private static string _memoryMappedControlFileNameAppend = "_memoryMappedControlFileNameAppend_" + strGUID;

    private bool _didThisThreadCreateTheMutex;

    #endregion variable declarations

    private void Start()
    {
        // Return values from creating the IPC artefacts
        bool IsChannelMutexOwned = false; bool IsProducerSemaphoreNew = false; bool IsConsumerSemaphoreNew = false;

        // Add the event handler for handling UI thread exceptions to the event.
        // Application.ThreadException += new
        //    ThreadExceptionEventHandler(ErrorHandlerForm.Form1_UIThreadException);
        // Set the unhandled exception mode to force all Windows Forms
        // errors to go through our handler.
        // Application.SetUnhandledExceptionMode(UnhandledExceptionMode.CatchException);

        // Add the event handler for handling non-UI thread exceptions to the event.
```

```
AppDomain.CurrentDomain.UnhandledException +=
    new UnhandledExceptionEventHandler(CurrentDomain_UnhandledException);

#region Atomic IPC artefact creation
// Trying to make this atomic so we just create the inter-process artefacts of the Channel once only in one thread
// Plan is get the mutex first then use it to lock other threads and processes out of the critical section that creates the
// semaphores, memory mapped files and anything else that may be added in the future
// If the mutex creation returns a value of "Existing" then we skip the critical section as the semaphores etc
// should already have been created by the first thread that created the mutex as new

// Don't initially acquire the mutex unless you're sure that you can lock out other threads
// Cos if you do it increments the acquition count and subsequently you can acquire it and the release in a
// finally block all day but it'll never get reset to zero so only the main thread can ever own it
// This is why the tests that directly executed Put and Take methods from the main thread worked fine
// but any test executed in a background thread blocked indefinately

// Bigger problem is even when I did initially acquire the mutex using the safe idiom in
// Concurrent Programming on Windows, Joe Duffy, Chap 5 Windows Kernel Synchronization page 214
// as follows:
//
//      bool IsMutexOwned;
//      _mutexLockQueue = new Mutex(true, _mutexLockQueueName, out IsMutexOwned);
//      if (!IsMutexOwned) _mutexLockQueue.WaitOne();
//      ... critical region, release etc....
//
// This doesn't work. I followed it with a block of code to create new semaphores, memory mapped files etc
// and another 'else' block to open existing ones. Every run produced errors either trying to create files
// that already exist or trying to open files that don't
// It works as I've done it now where I don't acquire the mutex on creation, wait for it and then create new
// semaphores which returns a handle to an existing one if there is one. I also use the CreateOrOpen method to
// open or get a handle to the memory mapped files
//
// Could understand this not working if this MMChannel were shared between threads because the IsMutexOwned field is
// not guarded and so could be modified by another thread before the if statement is executed but this class is
// supposed to be effectively Thread Local and I checked to see that one object is instantiated for each thread
//
// Needs more investigation!!!
#endregion Atomic IPC artefact creation

_mutexLockChannel = new Mutex(false, _mutexLockChannelName, out IsChannelMutexOwned);
// This critical section should be used to create ALL inter-process artefacts used in the system
_mutexLockChannel.WaitOne();
try
{
    // Save the state of whether this thread originally owned the mutex or not for use in the Close() method
```

```
_didThisThreadCreateTheMutex = IsChannelMutexOwned;

_consumerSemaphore = new Semaphore(0, _capacity, _consumerSemaphoreName, out IsConsumerSemaphoreNew);
_producerSemaphore = new Semaphore(_capacity, _capacity, _producerSemaphoreName, out IsProducerSemaphoreNew);
_memoryMappedDataFile = MemoryMappedFile.CreateOrOpen(_memoryMappedDataFileName, _fileSize);
_memoryMappedControlFile = MemoryMappedFile.CreateOrOpen(_memoryMappedControlFileName, CONTROL_DATA_FILE_SIZE);

if (_didThisThreadCreateTheMutex)
{
    string msg = string.Format("Start {0} Name = {1}, \n IsConsumerSemaphoreNew = {2}, IsProducerSemaphoreNew = {3}, IsMutexNew = {4}, Channel Type = {5}",
        DateTime.Now, _memoryMappedDataFileName, IsConsumerSemaphoreNew, IsProducerSemaphoreNew, IsChannelMutexOwned, _dsType);
    Console.WriteLine(msg);
}

// Create an array of views to access the data file
_viewAccessor = (MemoryMappedViewAccessor[])new MemoryMappedViewAccessor[_capacity];

// Populate the array of views from the memory mapped file.
// Each view starts at an offset calculated as the index times the size of the view and the size is specified as _viewSize
for (int i = 0; i < _capacity; i++)
{
    _viewAccessor[i] = _memoryMappedDataFile.CreateViewAccessor(i * _viewSize, _viewSize);
}

// Create a view to access the control file - queue control section
_controlDataAccessor = _memoryMappedControlFile.CreateViewAccessor(0, CONTROL_DATA_FILE_SIZE);

ControlData data = default(ControlData);

// Read the control data from the file. If this thread is the first to try to create the file then it will
// not have been initialized
_controlDataAccessor.Read(ZERO, out data);

// Just need one thread to log results for the lifetime of the channel.
if (!data.isInitialized)
{
    data.queueAddPosition = 0;
    data.queueTakePosition = 0;
    data.ds_type = (int) _dsType;
    data.initialCount = 0;
    data.stackAddTakePosition = 0;
    data.isInitialized = false;
    data.areResultsLogged = false;
    data.totalItemsDequeued = 0;
    data.totalItemsEnqueued = 0;
```

```
            data.startTimeLength = 0;
            data.startTimeTicks = 0;
            data.endTimeLength = 0;
            data.endTimeTicks = 0;
            data.ticksPerItem = 0;
            data.throughput = 0;
            data.microseconds = 0;
            data.nanoseconds = 0;
            data.testPutSum = 0;
            data.testTakeSum = 0;
            // Save the constructor parameters
            data.debug = _debug;
            data.test = _test;

            DateTime dtNow = DateTime.Now;
            if (!data.debug) { data.startTimeTicks = dtNow.Ticks; }

            string sTime = Convert.ToString(dtNow);
            char[] cStart = sTime.ToCharArray();

            data.startTimeLength = cStart.Length;
            for (int k = 0; k < data.startTimeLength; k++) { unsafe { data.startTime[k] = cStart[k]; } }

            data.startTimeTicks = DateTime.Now.Ticks;

            data.shutdownFlag = false;
            data.reservations = 0;

            data.isInitialized = true;

            // Save the isInitialized = true flag to the memory mapped file so we don't execute this code again
            _controlDataAccessor.Write(ZERO, ref data);
        }
    }
    catch (Exception e) { Console.WriteLine(e); throw; }
    finally { MLockChannel.ReleaseMutex(); }
}

#region Properties (getter/setter methods)
private Mutex MLockChannel { get { return _mutexLockChannel; } set { _mutexLockChannel = value; } }
protected DataStructureType DSType { get { return _dsType; } set { _dsType = value; } }

public string IPCName { get { return _ipcName; } }
// public int FileSize { get { return _fileSize; } }
// public int ViewSize { get { return _viewSize; } }
// public int Capacity { get { return _capacity; } }

#region Control Data properties

public unsafe ControlData MMFControlData
{
    // External - must be guarded by the mutex
    get
    {
        ControlData data = default(ControlData);
        MLockChannel.WaitOne();
        try
        {
            _controlDataAccessor.Read(ZERO, out data);
            return data;
        }
        catch (Exception e) { Console.WriteLine(e); throw; }
        finally { MLockChannel.ReleaseMutex(); }
    }
    set
    {
        ControlData data = default(ControlData);

        MLockChannel.WaitOne();
        try
        {
            data = value; _controlDataAccessor.Write(ZERO, ref data);
        }
        catch (Exception e) { Console.WriteLine(e); throw; }
        finally { MLockChannel.ReleaseMutex(); }
    }
}

public unsafe void ControlDataToString(ControlData data, String label = "")
{
    MLockChannel.WaitOne();
    try
    {
        if (label != null && label.Trim().Length > 0) { Console.WriteLine(label); }
        Console.WriteLine("Debug = {0}", data.debug);
        Console.WriteLine("Test = {0}", data.test);
        Console.WriteLine("Data structure type = {0}", (DataStructureType)_dsType);
        Console.WriteLine("QueueAddPosition = {0}", data.queueAddPosition);
        Console.WriteLine("QueueTakePosition = {0}", data.queueTakePosition);
        Console.WriteLine("InitialCount = {0}", data.initialCount);
```

```
                Console.WriteLine("StackAddTakePosition = {0}", data.stackAddTakePosition);
                Console.WriteLine("TotalItemsEnqueued = {0}", data.totalItemsEnqueued);
                Console.WriteLine("TotalItemsDequeued = {0}", data.totalItemsDequeued);
                Console.WriteLine("IsInitialized = {0}", data.isInitialized);
                Console.WriteLine("AreResultsLogged = {0}", data.areResultsLogged);

                // Copy the fixed byte array to an object byte array then convert the object byte array to a string
                char[] bStart = new char[data.startTimeLength];
                for (int i = 0; i < data.startTimeLength; i++) { bStart[i] = data.startTime[i]; }
                Console.WriteLine("StartTime = {0}", new string(bStart));
                //Console.WriteLine("StartTimeTicks = {0}", data.startTimeTicks);
                char[] bEnd = new char[data.endTimeLength];
                for (int k = 0; k < data.endTimeLength; k++) { bEnd[k] = data.endTime[k]; }
                Console.WriteLine("EndTime = {0}", new string (bEnd));
                Console.WriteLine("EndTimeTicks = {0}", data.endTimeTicks);
                Console.WriteLine("TicksPerItem = {0}", data.ticksPerItem);
                Console.WriteLine("Throughput = {0} items/second", data.throughput);
                Console.WriteLine("Microseconds = {0} per item", data.microseconds);
                Console.WriteLine("Nanoseconds = {0} per item", data.nanoseconds);
                Console.WriteLine("Test Put Sum = {0} per item", data.testPutSum);
                Console.WriteLine("Test Take Sum = {0} per item", data.testTakeSum);

                Console.WriteLine("\n");

                Console.WriteLine("Channel Shutdown = {0}", data.shutdownFlag);
                Console.WriteLine("Count of reservations = {0}", data.reservations);

                Console.WriteLine("\n");

            }
            catch (Exception e) { Console.WriteLine(e); throw; }
            finally { MLockChannel.ReleaseMutex(); }
        }

        //public unsafe void ShutdownDataToString(ShutdownData data, String label = "")
        //{
        //    MLockShutdown.WaitOne();
        //    try
        //    {
        //        if (label != null && label.Trim().Length > 0) { Console.WriteLine(label); }
        //        Console.WriteLine("ShutdownFlag = {0}", data.shutdownFlag);
        //        Console.WriteLine("Reservations = {0}", data.reservations);

        //        Console.WriteLine("\n");
        //    }
```

```
//    catch (Exception e) { Console.WriteLine(e); throw; }
//    finally { MLockShutdown.ReleaseMutex(); }
//}


#endregion Control Data properties

// Decodes byte array to unicode string.
public static string ByteArrayToString(byte[] data)
{
    Encoding utf16 = Encoding.Unicode;
    return utf16.GetString(data);
}

// Encodes, unicode, string to byte array.
public static byte[] StringToByteArray(string data)
{
    Encoding utf16 = Encoding.Unicode;
    return utf16.GetBytes(data);
}

#endregion Properties (getter/setter methods)

#region Add/take elements

#region Put a Scalar

public void Put<T>(T data, int timeoutMillis = DEFAULT_TIMEOUT ) where T : struct
{
    ControlData controlData = default(ControlData);

    _producerSemaphore.WaitOne(timeoutMillis);

    try
    {
        MLockChannel.WaitOne(timeoutMillis);

        _controlDataAccessor.Read(ZERO, out controlData);

        if (controlData.shutdownFlag) { throw new Exception("Channel is shutdown - cannot enqueue any more items"); }

        // Increment the number of items in the queue waiting to be dequeued
        controlData.reservations++;

        int addPosition = controlData.queueAddPosition;
        int originalAddPosition = addPosition;

        _viewAccessor[addPosition].Write(0, ref data);

        if (controlData.ds_type == (int)DataStructureType.Queue)
        {
            controlData.queueAddPosition = (++addPosition == _capacity) ? 0 : addPosition;
        }
        else
        {
            // Assuming the type defaults to Queue and the only alternative is a Stack
            controlData.queueTakePosition = addPosition;
            controlData.queueAddPosition = ++addPosition;
        }

        controlData.totalItemsEnqueued++;

        #region DEBUG
        // Attempt to catch ArrayIndexOutOfBoundsExceptions or data corruption due to cursors getting out of wack
        if (controlData.debug)
        {
            int diff = Math.Abs(controlData.queueAddPosition - originalAddPosition);
            if (!(diff == 1 || controlData.queueAddPosition == 0)) throw new Exception(string.Format
                ("New Add Position = {0} Originally = {1}", controlData.queueAddPosition, originalAddPosition));
        }
        #endregion DEBUG

        // Currently in test mode ONLY integers can be processed
        if (controlData.test) { controlData.testPutSum += Convert.ToInt64(data); }
    }
    catch (Exception e) { Console.WriteLine(e); throw; }
    finally { _controlDataAccessor.Write(ZERO, ref controlData); MLockChannel.ReleaseMutex(); _consumerSemaphore.Release(); }

}
#endregion Put a Scalar

#region Put an Array

public void Put<T>(T[] data, int timeoutMillis = DEFAULT_TIMEOUT) where T : struct
{
    // TODO Timeout not yet implemented as I haven't figured out what to do in that case - block forever
    ControlData controlData = default(ControlData);

    _producerSemaphore.WaitOne(timeoutMillis);
    try
    {
        MLockChannel.WaitOne(timeoutMillis);

        _controlDataAccessor.Read(ZERO, out controlData);

        if (controlData.shutdownFlag) { throw new Exception("Channel is shutdown - cannot enqueue any more items"); }

        // Increment the number of items in the queue waiting to be dequeued
        controlData.reservations++;

        int addPosition = controlData.queueAddPosition;
        int originalAddPosition = addPosition;

        _viewAccessor[addPosition].Write(0, data.Length);
        _viewAccessor[addPosition].WriteArray(4, data, 0, data.Length);

        if (controlData.ds_type == (int)DataStructureType.Queue)
        {
            controlData.queueAddPosition = (++addPosition == _capacity) ? 0 : addPosition;
        }
        else
        {
            // Assuming the type defaults to Queue and the only alternative is a Stack
            controlData.queueTakePosition = addPosition;
            controlData.queueAddPosition = ++addPosition;
        }
        controlData.totalItemsEnqueued++;

        #region DEBUG
        // Attempt to catch ArrayIndexOutOfBoundsExceptions or data corruption due to cursors getting out of wack
        if (controlData.debug)
        {
            int diff = Math.Abs(controlData.queueAddPosition - originalAddPosition);
            if (!(diff == 1 || controlData.queueAddPosition == 0)) throw new Exception(string.Format
                ("New Add Position = {0} Originally = {1}", controlData.queueAddPosition, originalAddPosition));
        }
        #endregion DEBUG
    }
    catch (Exception e) { Console.WriteLine(e); throw; }
    finally { _controlDataAccessor.Write(ZERO, ref controlData); MLockChannel.ReleaseMutex(); _consumerSemaphore.Release(); }
}

#endregion Put an Array
```

```
#region Take a Scalar

public T Take<T>(int timeoutMillis = DEFAULT_TIMEOUT) where T : struct
{
    // TODO Timeout not yet implemented as I haven't figured out what to do in that case - block forever

    T data = default(T);
    ControlData controlData = default(ControlData);

    _consumerSemaphore.WaitOne(timeoutMillis);

    try
    {
        MLockChannel.WaitOne(timeoutMillis);

        _controlDataAccessor.Read(ZERO, out controlData);

        if (controlData.shutdownFlag && controlData.reservations == 0)
            { throw new Exception("Channel is shutdown and empty - now disposing all resources"); }

        int takePosition = controlData.queueTakePosition;
        int originalTakePosition = takePosition;

        _viewAccessor[takePosition].Read<T>(0, out data);

        if (controlData.ds_type == (int)DataStructureType.Queue)
        {
            controlData.queueTakePosition = (++takePosition == _capacity) ? 0 : takePosition;
        }
        else
        {
            // Assuming the type defaults to Queue and the only alternative is a Stack
            controlData.queueAddPosition = takePosition;
            controlData.queueTakePosition = --takePosition;
        }

        controlData.totalItemsDequeued++;
        controlData.reservations--;

        #region DEBUG
        // Attempt to catch ArrayIndexOutOfBoundsExceptions or data corruption due to cursors getting out of wack
        if (controlData.debug)
        {
            int diff = Math.Abs(controlData.queueTakePosition - originalTakePosition);
            if (!(diff == 1 || controlData.queueTakePosition == 0)) throw new Exception(string.Format
                ("New Take Position = {0} Originally = {1}", controlData.queueTakePosition, originalTakePosition));
        }
        #endregion DEBUG

        // Currently in test mode ONLY integers can be processed
        if (controlData.test) { controlData.testTakeSum += Convert.ToInt64(data); }
    }
    catch (Exception e) { Console.WriteLine(e); throw; }
    finally { _controlDataAccessor.Write(ZERO, ref controlData); MLockChannel.ReleaseMutex(); _producerSemaphore.Release(); }

    return data;

    #region Type Parameters and Conversions
    // The most common scenario is when you want to perform a reference conversion:
    // StringBuilder Foo<T> (T arg)
    // {
    //     if (arg is StringBuilder)
    //     return (StringBuilder) arg; // Will not compile
    // }
    // Without knowledge of T’s actual type, the compiler is concerned that you might
    // have intended this to be a custom conversion. The simplest solution is to instead use
    // the as operator, which is unambiguous because it cannot perform custom conversions:
    // StringBuilder Foo<T> (T arg)
    // {
    //     StringBuilder sb = arg as StringBuilder;
    //     if (sb != null) return sb;
    // }
    // A more general solution is to first cast to object.
    // This works because conversions to/from object are assumed not to be custom conversions, but reference or boxing/
    // unboxing conversions. In this case, StringBuilder is a reference type, so it has to be a reference conversion:
    // return (StringBuilder) (object) arg;
    #endregion Type Parameters and Conversions
    #region default generic types and explicitly nulling a reference
    // INFO This is one of the few cases where explicitly setting to null is necessary because the element wouldn't otherwise go out of scope
    // TODO when replaced by memory mapped file the 'view' will just be overwritten
    // NOTE: This does not release the memory!! Garbage collection is still necessary. All it does is let the GC know that
    // the object is dead when the object is checked during a collection
    // itegms[i] = default(E); // null;

    // E enull = default(E);
    // this needs to write the default value to the viewSize buffer
    // _accessor[i].Write(0, ref enull);

    // Could use items[i] = null if the element was guaranteed to be an object (see class declaration above)
    // We might declare the data structure to accept only objects and box any primitive if, for example, we wanted to be able
```

```
    // to call Dispose on them.
    // See Joe Duffy's Concurrent Programming in Windows, chapter 10, Memory Models and Lock Freedom page 527
    #endregion default generic types and explicitly nulling a reference
}

#endregion Take a Scalar

#region Take an Array

public int Take<T>(out T[] data, int timeoutMillis = DEFAULT_TIMEOUT) where T : struct
{
    // TODO Timeout not yet implemented as I haven't figured out what to do in that case - block forever

    int numItems = 0;
    data = default(T[]);
    ControlData controlData = default(ControlData);

    _consumerSemaphore.WaitOne(timeoutMillis);

    try
    {
        MLockChannel.WaitOne(timeoutMillis);

        _controlDataAccessor.Read(ZERO, out controlData);

        if (controlData.shutdownFlag && controlData.reservations == 0)
            { throw new Exception("Channel is shutdown and empty - now disposing all resources"); }

        int takePosition = controlData.queueTakePosition;
        int originalTakePosition = takePosition;

        #region Array size issue
        // Read an array of data items from the view and assign it to the output parameter - type T[]
        // The length of the array was written to the view by the Put method as an Int in 4 bytes starting at position 0
        // Seems unlikely that we would have an array of data bigger than 2 billion - odd but be careful if you change the
        // array size to a long and the ReadInt32 to ReadInt64. Did that accidently without changing the return value of this
        // method. NUnit reported an arithmetic overflow exception but on the line "int numItems = 0";
        // Took a long time to find the real cause of the problem i.e. changing to ReadInt64
        #endregion Array size issue
        data = new T[_viewAccessor[takePosition].ReadInt32(0)];
        numItems = _viewAccessor[takePosition].ReadArray(4, data, 0, data.Length);

        if (controlData.ds_type == (int)DataStructureType.Queue)
        {
            controlData.queueTakePosition = (++takePosition == _capacity) ? 0 : takePosition;
    }
    else
    {
        // Assuming the type defaults to Queue and the only alternative is a Stack
        controlData.queueAddPosition = takePosition;
        controlData.queueTakePosition = --takePosition;
    }

    controlData.totalItemsDequeued++;
    controlData.reservations--;

    #region DEBUG
    // Attempt to catch ArrayIndexOutOfBoundsExceptions or data corruption due to cursors getting out of wack
    if (controlData.debug)
    {
        int diff = Math.Abs(controlData.queueTakePosition - originalTakePosition);
        if (!(diff == 1 || controlData.queueTakePosition == 0)) throw new Exception(string.Format
            ("New Take Position = {0} Originally = {1}", controlData.queueTakePosition, originalTakePosition));
    }
    #endregion DEBUG
}
catch (Exception e) { Console.WriteLine(e); throw; }
finally { _controlDataAccessor.Write(ZERO, ref controlData); MLockChannel.ReleaseMutex(); _producerSemaphore.Release(); }

return numItems;

#region Type Parameters and Conversions
// The most common scenario is when you want to perform a reference conversion:
// StringBuilder Foo<T> (T arg)
// {
//     if (arg is StringBuilder)
//     return (StringBuilder) arg; // Will not compile
// }
// Without knowledge of T’s actual type, the compiler is concerned that you might
// have intended this to be a custom conversion. The simplest solution is to instead use
// the as operator, which is unambiguous because it cannot perform custom conversions:
// StringBuilder Foo<T> (T arg)
// {
//     StringBuilder sb = arg as StringBuilder;
//     if (sb != null) return sb;
// }
// A more general solution is to first cast to object.
// This works because conversions to/from object are assumed not to be custom conversions, but reference or boxing/
// unboxing conversions. In this case, StringBuilder is a reference type, so it has to be a reference conversion:
// return (StringBuilder) (object) arg;
```

```
    #endregion Type Parameters and Conversions
    #region default generic types and explicitly nulling a reference
    // See Joe Duffy's Concurrent Programming in Windows, chapter 10, Memory Models and Lock Freedom page 527
    #endregion default generic types and explicitly nulling a reference
}

#endregion Take an Array

#endregion Add/take elements


public bool Debug
{
    // External - must be guarded by the mutex
    get
    {
        ControlData data = default(ControlData);
        MLockChannel.WaitOne();
        try
        {
            _controlDataAccessor.Read(ZERO, out data);
            return data.debug;
        }
        catch (Exception e) { Console.WriteLine(e); throw; }
        finally { MLockChannel.ReleaseMutex(); }
    }
    set
    {
        ControlData data = default(ControlData);

        MLockChannel.WaitOne();
        try
        {
            data.debug = value; _controlDataAccessor.Write(ZERO, ref data);
        }
        catch (Exception e) { Console.WriteLine(e); throw; }
        finally { MLockChannel.ReleaseMutex(); }
    }
}

public bool Test
{
    // External - must be guarded by the mutex
    get
    {
        ControlData data = default(ControlData);
        MLockChannel.WaitOne();
        try
        {
            _controlDataAccessor.Read(ZERO, out data);
            return data.test;
        }
        catch (Exception e) { Console.WriteLine(e); throw; }
        finally { MLockChannel.ReleaseMutex(); }
    }
    set
    {
        ControlData data = default(ControlData);

        MLockChannel.WaitOne();
        try
        {
            data.test = value; _controlDataAccessor.Write(ZERO, ref data);
        }
        catch (Exception e) { Console.WriteLine(e); throw; }
        finally { MLockChannel.ReleaseMutex(); }
    }
}


public void clearTestData()
{
    // External - must be guarded by the mutex

        ControlData data = default(ControlData);
        MLockChannel.WaitOne();
        try
        {
            _controlDataAccessor.Read(ZERO, out data);

            data.totalItemsEnqueued = 0;
            data.ticksPerItem = 0;
            data.nanoseconds = 0;
            data.microseconds = 0;
            data.throughput = 0;
        }
        catch (Exception e) { Console.WriteLine(e); throw; }
        finally { _controlDataAccessor.Write(ZERO, ref data); MLockChannel.ReleaseMutex(); }
 }
```

```
private static void CurrentDomain_UnhandledException(object sender, UnhandledExceptionEventArgs e)
{
    // Set up uncaught exception handler in case some dodgy code throws a RunTimeException
    // This won't work if the exception is passed to some even more dodgy 3rd psrty code that swallows
    // the exception. Does work in the case of dodgy 3rd party rogue code like ActiveMQ which kindly throws
    // some kind of runtime exception if you don't have a 'geronimo' jar in your classpath when you try to
    // instantiate a connectionFactory or ActiveMQConnectionFactory
    // Java version looks like this - ASExceptionHandler UEH = new ASExceptionHandler();
    //                                Thread.setDefaultUncaughtExceptionHandler(UEH);
    // Java also has per-thread scheduler handlers set up using the same class
    Console.Write(e.ExceptionObject.ToString());

}

#region Dispose of IPC artefacts

// TODO Implement IDisposable
public void Report()
{
    ControlData data = default(ControlData);

    MLockChannel.WaitOne();
    try
    {
        _controlDataAccessor.Read(ZERO, out data);

        // Just need one thread to log results from the lifetime of the channel.
        if (!data.areResultsLogged)
        {
            data.areResultsLogged = true;

            DateTime dtNow = DateTime.Now;

            string eTime = Convert.ToString(dtNow);
            char[] cEnd = eTime.ToCharArray();

            data.endTimeLength = cEnd.Length;
            for (int k = 0; k < data.endTimeLength; k++) { unsafe { data.endTime[k] = cEnd[k]; } }
            data.endTimeTicks = DateTime.Now.Ticks;

            long elapsedTime = data.endTimeTicks - data.startTimeTicks;

            // calculate throughput if any data was actually processed
            if (data.totalItemsEnqueued > 0)
            {
                data.ticksPerItem = (int)(elapsedTime / data.totalItemsEnqueued);
                TimeSpan elapsedSpan = new TimeSpan(data.ticksPerItem);
                double milliSeconds = elapsedSpan.TotalMilliseconds;
                data.nanoseconds = data.ticksPerItem * 100;
                data.microseconds = data.nanoseconds / 1000;
                data.throughput = (int)(1000000000 / data.nanoseconds);
            }

            // Print out the results
            ControlDataToString(data, "MMChannel");

            // Save the areResultsLogged = true flag to the memory mapped file so we don't execute this code again
            _controlDataAccessor.Write(ZERO, ref data);
        }
    }
    catch (Exception e) { Console.WriteLine(e); throw; }
    finally
    {
        MLockChannel.ReleaseMutex();
    }
}

public void Dispose() // NOT virtual
{
    Dispose(true);
    GC.SuppressFinalize(this); // Prevent finalizer from running.
}

public void shutdown()
{
    // Goal is to shutdown gracefully so if there are still items in the queue then allow the consumer(s) to drain them
    // Once we have the mutex then the producers cannot enqueue any more items and once we release the mutex in here
    // the checks in the 'Put()' methods will prevent them doing so in the future
    // The consumers(s) will continue to drain the queue until it is empty
    ControlData data = default(ControlData);

    MLockChannel.WaitOne();
    try
    {
        _controlDataAccessor.Read(ZERO, out data);

        data.shutdownFlag = true;
        // Dispose();
    }
    catch (Exception e) { Console.WriteLine(e); throw; }
    finally { _controlDataAccessor.Write(ZERO, ref data); MLockChannel.ReleaseMutex(); }
}

protected virtual void Dispose(bool disposing)
{
    if (disposing)
    {
        // Call Dispose() on other objects owned by this instance.
        // You can reference other finalizable objects here.
        Report();
    }

    // Release unmanaged resources owned by (just) this object.
    _consumerSemaphore.Dispose();
    _producerSemaphore.Dispose();

    for (int i = 0; i < _capacity; i++) { _viewAccessor[i].Dispose(); }

    _controlDataAccessor.Dispose();

    #region Garbage Collection and Finalizers
    // I forgot to dispose of the memory mapped files - oops!
    // This bug survived literally hundreds of tests runs because I was running them in groups of three, each
    // creating a memory mapped file with a different name.
    // It wasn't until I tried running the same test re[eatedly that it failed - throwing an Exception that
    // the file already exists (I was using the CreateNew() method to create them)
    // The mm file has a built-in finalizer which gets rid of it when the GC collector runs so I gues that by the time
    // you've recycled round to the first test the GC has disposed of the file it created in its previous incarnation
    // Once I started repeating the same test it only took two or three goes before the test tried to create a file
    // with the same name as one left over from the previous run
    // Moral of the story is you can't depend on finalizers being run in a timely fashion or in fact ever which
    // any fule know of course but the first time I've seen it in action
    #endregion Garbage Collection and Finalizers

    _memoryMappedDataFile.Dispose();
    _memoryMappedControlFile.Dispose();

    _mutexLockChannel.Dispose();
}
~MMChannel()
{
    Dispose(false);
}
```

```
        // Dispose is overloaded to accept a bool disposing flag. The parameterless version is not declared as virtual
        // and simply calls the enhanced version with true.
        // The enhanced version contains the actual disposal logic and is protected and virtual; this provides a safe
        // point for subclasses to add their own disposal logic.
        // The disposing flag means it’s being called “properly” from the Dispose method rather than in “last-resort mode”
        // from the finalizer. The idea is that when called with disposing set to false, this method should not, in general,
        // reference other objects with finalizers (because such objects may themselves have been finalized and
        // so be in an unpredictable state). This rules out quite a lot! Here are a couple of tasks it can still perform in
        // last-resort mode, when disposing is false:
        // • Releasing any direct references to operating system resources (obtained, perhaps, via a P/Invoke call to the Win32 API)
        // • Deleting a temporary file created on construction
        // To make this robust, any code capable of throwing an exception should be wrapped n a try/catch block, and the exception,
        // ideally, logged. Any logging should be as simple and robust as possible.
        // Notice that we call GC.SuppressFinalize in the parameterless Dispose method—this prevents the finalizer from running when
        // the GC later catches up with it. Technically, this is unnecessary, as Dispose methods must tolerate repeated calls. However,
        // doing so improves performance because it allows the object (and its referenced objects) to be garbage-collected in a single cycle.


        #endregion Dispose of IPC artefacts
    }



}


#region Note 1. Another possible initialization method
// Doesn't seem feasible though because either the variables should be readonly or guarder with a lock
// Not possible to set the values of readonly variables except in a static constructor or variable initializer and if we need to
// guard with a lock then no point trying to use lazy static initialization

// private static readonly string ipcName;
// private static readonly int fileSize;
// private static readonly int viewSize;
// private static readonly int capacity;
// private static readonly bool debug;
// private static readonly bool test;
// private static readonly DataStructureType dsType;

// public static void init(string aIpcName, int aFileSize, int aViewSize, int aCapacity,
// bool aDebug = false, bool aTest = false, DataStructureType aDsType = DataStructureType.Queue)
// {
//     ipcName = aIpcName;
//     fileSize = aFileSize;
//     viewSize = aViewSize;
//     capacity = aCapacity;
//     debug = aDebug;
//     test = aTest;
//     dsType = aDsType;
// }

//     private class LazyResourceHolder {

// Problem. How do we get the parameters to pass to the static initializer?
// Store them somewhere externally before calling the getResource() method?
//         private static MMChannel channel; // new MMChannel(ipcName, fileSize, viewSize, capacity, debug, test, dsType);

//         public static MMChannel getResource(string ipcName, int fileSize, int viewSize, int capacity,
//                                         bool debug = false, bool test = false, DataStructureType dsType = DataStructureType.Queue) {

//         return LazyResourceHolder.channel;
//         }
//     }

// Using static lazy initialization. The static LazyResourceHolder inner class only exists to create the resource the first time it
// is referenced by calling getResource()
// return LazyResourceHolder.getResource(ipcName, fileSize, viewSize, capacity, debug, test, dsType);

#endregion another possible initialization method
```

UTCutil.cs

```
/*****************************************************************************************************************************************************
 * The goal of the AlphaSystematics Project is create an open-source system for forward-testing systematic strategies with live market data and trade feeds.
 * It enables strategies developed in Excel to be connected to trading venues via industry standard FIX messaging.
 *
 * Copyright (C) 2009  Antonio Tapper. www.alphasystematics.org

 * This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

 * This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more details.

 * You should have received a copy of the GNU General Public License along with this program.  If not, see <http://www.gnu.org/licenses/>.
*****************************************************************************************************************************************************/
```

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Globalization;
using System.Collections.Concurrent;
using System.Collections;
using System.Collections.Specialized;
using System.Diagnostics;
using System.Reflection;
using System.Runtime.InteropServices;


namespace com.alphaSystematics.concurrency
{
    public static class UTCutil
    {

        public const String performanceCounter_bytes_in_all_heaps = "bytes_in_all_heaps";
        public const String performanceCounter_gc_handles = "gc_handles";
        public const String performanceCounter_gen_0_heap_size = "gen_0_heap_size";
        public const String performanceCounter_gen_1_heap_size = "gen_1_heap_size";
        public const String performanceCounter_gen_2_heap_size = "gen_2_heap_size";
        public const String performanceCounter_large_object_heap_size = "large_object_heap_size";

        // T.Tapper MOD 0002.2 Change dates from local to UTC
        static public Nullable<DateTime> convertLocalToUTC(String sDate)
        {
            DateTime utcTime;

            try
            {
                DateTime dtEntered = DateTime.Parse(sDate, CultureInfo.InvariantCulture);
                dtEntered = dtEntered.Date;
                //this sets the time to 00:00:00 if one was passed in in the string
                // if not then concatenate " 00:00:00" to the date part of sDate string - sDate += " 00:00:00";
                utcTime = dtEntered.ToUniversalTime();
                return utcTime;
            }
            catch (System.ArgumentException)
            {
                // log invalid date string passed in;
            }
            catch (System.FormatException)
            {
                // log invalid date string passed in;
            }
            // test return value
            //if (!value.HasValue)
            return null;

        }

        static public Nullable<DateTime> convertLocalToUTC(DateTime dtDate)
        {
            DateTime utcTime;

            try
            {
                utcTime = dtDate.ToUniversalTime();
                return utcTime;
            }
            catch (System.ArgumentException)
            {
                // log invalid date string passed in;
            }
            catch (System.FormatException)
            {
                // log invalid date string passed in;
            }
            //if (!utcTime.value.HasValue)
            return null;
            // test return value if (!value.HasValue)
        }

        static public Nullable<DateTime> convertLocalToUTCFromSOD(DateTime dtDate)
        {
            DateTime utcTime;

            try
            {
                // Remove the time component to determine "00:00:00" on the requested day i.e S.O.D
                utcTime = dtDate.Date.ToUniversalTime();
                // Return the UTC that is equivalent to "00:00:00" on the requested day
                return utcTime;
            }
            catch (System.ArgumentException)
            {
                // log invalid date string passed in;
            }
            catch (System.FormatException)
            {
                // log invalid date string passed in;
            }
            return null;
            // test return value if (!value.HasValue)
        }

        static public Nullable<DateTime> convertLocalToUTCToEOD(DateTime dtDate)
        {
            DateTime utcTime;

            try
            {
                dtDate = dtDate.AddDays(1);
                // Remove the time component, to determine "00:00:00" on the day after the requested day i.e S.O.D of the following day
                utcTime = dtDate.Date.ToUniversalTime();
                // To filter up to EOD of the requested day you must specify "<" the returned value
                // e.g "toDate < (dateTime) UTCutil.convertLocalToUTCToEOD( dtRequired )"
                // If you say "toDate < (dateTime) UTCutil.convertLocalToUTCToEOD( dtRequired )" you will also get anything
                // with a timestamp of "00:00:00" on the day following the one you requested.
                // Return the UTC that is equivalent to "00:00:00" on the day following the requested day
                // Ex. Requested date to = 1 December EDT (Aus). We want everything up to midnight on 1 December
                // or before "00:00:00 2 December" EDT, in other words
                // UTC equivalent = 13:00 1 December. You must specify " todate < 13:00 1 December"
                // " todate <= 13:00 1 December " would also select anything that was actually equivalent to  "00:00:00 2 December" EDT
                return utcTime;
            }
            catch (System.ArgumentException)
            {
                // log invalid date string passed in;
            }
            catch (System.FormatException)
            {
                // log invalid date string passed in;
            }
            return null;
            // test return value if (!value.HasValue)
        }

        // INT
        public static int toInt(object i)
        {
            bool wasNull = false;
            return toInt(i, out wasNull);
        }
```

```
public static int toInt(object i, out bool wasNull)
{
    // have to initialize output parameters
    // this is just in case you're interested in knowing whether the value you got back was originally null
    wasNull = false;

    if (i == System.DBNull.Value)
    {
        wasNull = true;
        return 0;
    }
    else
    {
        try
        {
            return Convert.ToInt32(i);
        }
        catch (System.FormatException e)
        {
            return 0;
        }
    }
}

#region Lazy static singleton initialization as per Java Concurrency in Practice Chap 16 Memory Model
public static class ExecutingAssembly
{
    private static class InitExecutingAssembly
    {
        // Instantiate a target object and set the Type instance to the target class type
        // Instantiate an Assembly class to the assembly housing the Integer type.
        public static Assembly assembly = Assembly.GetAssembly(new Int32().GetType());
    }
    public static Assembly GetExecutingAssembly { get { return InitExecutingAssembly.assembly; } }
}
#endregion Lazy static singleton initialization as per Java Concurrency in Practice Chap 16 Memory Model


public static string GetInstanceNameForProcessId(int pid)
{
    // The CLR counters are per instance counters, thus you need to specify the instance name for the process you wish
    // to query the counters for.
    // Should also use the constructor overload that allows you to specify that you wish to access the instance in
    // "read-only" mode:
    // new PerformanceCounter(".NET CLR Memory", "# bytes in all heaps", Process.GetCurrentProcess().ProcessName, true);
    // The instance name is not necessarily the same as Process.ProcessName (or Process.GetCurrentProcess().ProcessName
    // for that matter). If there are multiple instances of a process, i.e. executable, the process name is created by
    // appending a #<number>. To figure out the actual instance name of a process you should query the
    // .NET CLR Memory\Process ID counter.

    var cat = new PerformanceCounterCategory(".NET CLR Memory");
    foreach (var instanceName in cat.GetInstanceNames())
    {
        using (var pcPid = new PerformanceCounter(cat.CategoryName, "Process ID", instanceName))
        {
            if ((int)pcPid.NextValue() == pid)
            {
                return instanceName;
            }
        }
    }

    throw new ArgumentException(
        string.Format("No performance counter instance found for process id '{0}'", pid),
        "pid");
}


public static Dictionary<String, PerformanceCounter> ReadKeyMemoryAndHandlePerformanceCounters(String applicationInstance)
{
    // Declare a variable of type String named applicationInstance.
    // String applicationInstance = GetInstanceNameForProcessId(Process.GetCurrentProcess().Id);

    // Declare a variable of type ArrayList named performanceCounters.
    // ArrayList performanceCounters = new ArrayList();
    Dictionary<String, PerformanceCounter> performanceCounters = new Dictionary<String, PerformanceCounter>();

    // Instantiate the PeformanceCounters that can indicate memory and handle performance issues.
    // Add each PerformanceCounter to the performanceCounters ArrayList as it is instantiated.

    // No. of bytes in all heaps
    performanceCounters.Add(performanceCounter_bytes_in_all_heaps,
        new PerformanceCounter(".NET CLR Memory", "# bytes in all heaps", applicationInstance, true));
    // No. of GC Handles
    performanceCounters.Add(performanceCounter_gc_handles, new PerformanceCounter(".NET CLR Memory", "# GC Handles", applicationInstance, true));
    // Gen 0 heap Size
    performanceCounters.Add(performanceCounter_gen_0_heap_size,
        new PerformanceCounter(".NET CLR Memory", "Gen 0 Heap Size", applicationInstance, true));
    // Gen 1 heap Size
    performanceCounters.Add(performanceCounter_gen_1_heap_size,
        new PerformanceCounter(".NET CLR Memory", "Gen 1 heap Size", applicationInstance, true));
    // Gen 2 heap Size
    performanceCounters.Add(performanceCounter_gen_2_heap_size,
        new PerformanceCounter(".NET CLR Memory", "Gen 2 heap Size", applicationInstance, true));
    // Large Object heap size
    performanceCounters.Add(performanceCounter_large_object_heap_size,
        new PerformanceCounter(".NET CLR Memory", "Large Object Heap size", applicationInstance, true));

    //StringBuilder counterSnapshot = new StringBuilder();

    //// Loop through the PerformanceCounters in performanceCounters ArrayList.
    //Dictionary<String, PerformanceCounter>.ValueCollection counters = performanceCounters.Values;

    //foreach (PerformanceCounter typePerformanceCounter in counters)
    //{
    //    // Append the PerformanceCounter's name and its Value to the counterSnapshot.
    //    counterSnapshot.Append(
    //        typePerformanceCounter.CounterName.ToString() + " " + GetCounterValue(typePerformanceCounter).ToString() + "\n");
    //}
    //// Console.WriteLine(counterSnapshot.ToString());

    return performanceCounters;
}

public static String GetCounterValue (PerformanceCounter pPerformanceCounter) {

    String retval = "";

    // Retrieve PerformanceCounter result based on its CounterType.
    switch (pPerformanceCounter.CounterType)
    {
        case PerformanceCounterType.NumberOfItems32:
            retval = pPerformanceCounter.RawValue.ToString();
            break;

        case PerformanceCounterType.NumberOfItems64:
            retval = pPerformanceCounter.RawValue.ToString();
            break;

        case PerformanceCounterType.RateOfCountsPerSecond32:
            retval = pPerformanceCounter.NextValue().ToString();
            break;

        case PerformanceCounterType.RateOfCountsPerSecond64:
            retval = pPerformanceCounter.NextValue().ToString();
            break;

        case PerformanceCounterType.AverageTimer32:
            retval = pPerformanceCounter.NextValue().ToString();
            break;

        default:
            retval = null;
            break;
    }

    return retval;
}




// DOUBLE
public static double toDouble(object d)
{
    bool wasNull = false;
    return toDouble(d, out wasNull);
}
public static double toDouble(object d, out bool wasNull)
{
    // have to initialize output parameters
    // this is just in case you're interested in knowing whether the value you got back was originally null
    wasNull = false;

    if (d == System.DBNull.Value)
    {
        wasNull = true;
        return 0;
    }
    else
    {
        try
        {
            return Convert.ToDouble(d);
        }
        catch (System.FormatException e)
        {
            return 0;
        }
    }
```

```
}



// DECIMAL
public static decimal toDecimal(object d)
{
    bool wasNull = false;
    return toDecimal(d, out wasNull);
}
public static decimal toDecimal(object d, out bool wasNull)
{
    // have to initialize output parameters
    // this is just in case you're interested in knowing whether the value you got back was originally null
    wasNull = false;

    if (d == System.DBNull.Value)
    {
        wasNull = true;
        return 0;
    }
    else
    {
        try
        {
            return Convert.ToDecimal(d);
        }
        catch (System.FormatException e)
        {
            return 0;
        }
    }
}

// DATETIME
public static Nullable<DateTime> toDateTime(object d)
{
    bool wasNull = false;
    return toDateTime(d, out wasNull);
}
public static Nullable<DateTime> toDateTime(object d, out bool wasNull)
{
    // have to initialize output parameters
    // this is just in case you're interested in knowing whether the value you got back was originally null
    wasNull = false;
    DateTime dt = new DateTime();
    if (d == System.DBNull.Value)
    {
        wasNull = true;
        return dt;
    }
    else
    {
        try
        {
            return Convert.ToDateTime(d);
        }
        catch (System.FormatException e)
        {
            return null;
        }
    }
}

// STRING
public static string toString(object s)
{
    bool wasNull = false;
    return toString(s, out wasNull);
}
public static string toString(object s, out bool wasNull)
{
    // have to initialize output parameters
    // this is just in case you're interested in knowing whether the value you got back was originally null
    wasNull = false;

    if (s == System.DBNull.Value)
    {
        wasNull = true;
        return string.Empty;
    }
    else
    {
        try
        {
            return Convert.ToString(s);
        }
        catch (System.FormatException e)
        {
```

```
                    return String.Empty;
                }
            }
        }


    }
}
```

Enums.cs

```
/********************************************************************************************************************************************
 * The goal of the AlphaSystematics Project is create an open-source system for forward-testing systematic strategies with live market data and trade feeds.
 * It enables strategies developed in Excel to be connected to trading venues via industry standard FIX messaging.
 *
 * Copyright (C) 2009  Antonio Tapper. www.alphasystematics.org

 * This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

 * This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more details.

 * You should have received a copy of the GNU General Public License along with this program.  If not, see <http://www.gnu.org/licenses/>.
********************************************************************************************************************************************/

using System;
using System.Collections;
using System.Collections.Generic;
using System.Data;

namespace com.alphaSystematics.concurrency
{
    public enum DataStructureType
    {
        Default,
        Queue,
        Stack
    }

}
```

Inter Process producer-Consumer Queue Test Harness

ConsumerTestRunner.cs

```
/*********************************************************************************************************************************************************
 * The goal of the AlphaSystematics Project is create an open-source system for forward-testing systematic strategies with live market data and trade feeds.
 * It enables strategies developed in Excel to be connected to trading venues via industry standard FIX messaging.
 *
 * Copyright (C) 2009  Antonio Tapper. www.alphasystematics.org

 * This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

 * This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more details.

 * You should have received a copy of the GNU General Public License along with this program.  If not, see <http://www.gnu.org/licenses/>.
*********************************************************************************************************************************************************/

using System;
using System.Collections.Generic;
using System.IO;
using System.IO.MemoryMappedFiles;
using System.Text;
using System.Threading;
using com.alphaSystematics.concurrency;
using System.Linq;
using System.Threading.Tasks;
using System.Collections.Concurrent;
using System.Diagnostics;
// using QuickFix;
using System.Runtime.InteropServices;

namespace TestMMFile_Destination
{
    public class ConsumerTestRunner
    {

        // The hardest part of writing tests is that when they fail you don't know if it is the test or the application
        // thats broken unless you have confidence that the tests themselves have been tested thoroughly
        // in this case we are lucky in that we are trying to mimic the functionality of an existing library class but extend
        // it to use inter process.
        // We can drop in the library class here in order to test the test because we have confidence that the library class
        // works so if the tests fail when using library class then the tests are broken - For NUnit tests in a single process

        static DataStructureType initTestDataStructureType = default(DataStructureType);
        const long AVERAGE_THROUGHPUT_THRESHOLD_TICKS = 1000;
        int initNoOfTrials = 0; int initTestRunNumber = 0; int initTestSuiteNumber = 0;
        const int defaultNoOfTrials = 1000000;
        const bool DEBUG = true; static bool TEST = false;

        static int Menu()
        {
            string result = ""; int choice = 0; bool valid = false;

            while (!valid)
            {
                Console.Clear();
                Console.WriteLine("Memory Mapped Message Channel test suite (Consumers). Please choose from the following options:\n");

                Console.WriteLine("1: Test menu for the Consumers\n");
                // Console.WriteLine("1: Test that the queue is empty when constructed\n");
                // Console.WriteLine("2: Test that the queue is full after Puts and empty after Takes\n");
                Console.WriteLine("3: Test that the Take method blocks when the queue is empty\n");
                Console.WriteLine("4: Test that the Take method is unblocked when an item is added\n");
                Console.WriteLine("5: Test Put and Take methods with Int data and equal numbers of producers and consumers\n");
                Console.WriteLine("6: Test Put and Take methods with Long data and equal numbers of producers and consumers\n");
                Console.WriteLine("7: Test Put and Take methods with array data (chars) and equal numbers of producers and consumers\n");
                Console.WriteLine("8: Test Put and Take methods with struct data and equal numbers of producers and consumers\n");
                Console.WriteLine("10: Execute Test Group No. 00 - 1 Billion Integers\n");
                Console.WriteLine("11: Execute Test Group No. 01 - 1 Billion Longs\n");
                Console.WriteLine("12: Execute Test Group No. 02 - 1 Billion Strings\n");
                Console.WriteLine("13: Execute Test Group No. 03 - 1 Billion Structs\n");
                Console.WriteLine("14: Execute Test Groups 00, 01, 02 and 03\n");


                Console.WriteLine("Q: Quit\n");

                // get the 1st character of input and quit if it is "Q"
                result = Console.ReadLine();
                if (result.ToUpper().Equals("Q")) { result = "99"; }

                try
                {
                    choice = int.Parse(result);
                }
                catch (ArgumentException) { }
                catch (FormatException) { }

                switch (choice)
                {
                    case 0:
                        Console.WriteLine("Quitting test harness {0} please wait...", result);
                        valid = true;
                        break;
                    case 1:
                    // case 2:
                    case 3:
                    case 4:
                    case 5:
                    case 6:
                    case 7:
                    case 8:
                    case 10:
                    case 11:
                    case 12:
                    case 13:
                    case 14:
                        Console.WriteLine("Executing test {0} please wait...", result);
                        valid = true;
                        break;

                    default:
                        Console.WriteLine("Invalid selection {0}. Please select 1, 4, 5, 6, 7, 8, 9, 10, 11. 12, 13, 14  or Quit.\n\n\n\n\n", result);
                        break;
                }
            }
            return choice;
        }

        static string queueOrStack()
        {
            Console.WriteLine("Please choose to test a Queue or a Stack (Default = Queue)");
            string result = Console.ReadLine();
            if (result.ToUpper().Equals("S")) { result = "S"; } else { result = "Q"; }

            return result;
        }
```

```
static int numberOfTrials()
{
    Console.WriteLine("Please a number of trials, between 1 and 1,000,000, to test (Default = 1,000,000)");
    string result = Console.ReadLine();
    int choice = 0;

    try
    {
        choice = int.Parse(result);
    }
    catch (ArgumentException) { }
    catch (FormatException) { }

    if (!(choice > 0 && choice < 1000000))
    {
        Console.WriteLine(choice + " is invalid. Defaulting to 1,000,000)");
        choice = defaultNoOfTrials;
    }
    return choice;
}

static void Main(String[] args)
{
    try
    {
        // Add the event handler for handling UI thread exceptions to the event.
        // Application.ThreadException += new
        //    ThreadExceptionEventHandler(ErrorHandlerForm.Form1_UIThreadException);
        // Set the unhandled exception mode to force all Windows Forms
        // errors to go through our handler.
        // Application.SetUnhandledExceptionMode(UnhandledExceptionMode.CatchException);

        // Add the event handler for handling non-UI thread exceptions to the event.
        AppDomain.CurrentDomain.UnhandledException +=
            new UnhandledExceptionEventHandler(CurrentDomain_UnhandledException);

        // ConsumerTestRunner consumer = new ConsumerTestRunner();
        int choice = 0;

        do {
            ConsumerTestRunner consumer = null;
            consumer = new ConsumerTestRunner();

            choice = Menu();
```

```
                if (choice > 0) {
                    String channelType = queueOrStack();
                    int numberOftrials = numberOfTrials();
                    consumer.Init(channelType, numberOftrials);
                }

                switch (choice)
                {
                    case 1:
                    Console.WriteLine("Press ENTER to complete the Menu test for the Consumers");
                    Console.ReadLine();
                    break;

                    // case 2:
                    //case 3:
                    //Console.WriteLine("Press ENTER to execute Test _03_testTakeBlocksWhenEmpty();");
                    //Console.ReadLine();
                    //consumer._03_testTakeBlocksWhenEmpty();
                    //Console.WriteLine("Press ENTER to EXIT Test _03_testTakeBlocksWhenEmpty();");
                    //Console.ReadLine();
                    //break;

                    case 4:
                    TEST = false;
                    Console.WriteLine("Run the Producer FIRST then press ENTER to execute the consumer component of Test
_04_testTakeIsUnblockedWhenElementAdded();");
                    Console.ReadLine();
                    consumer._04_testTakeIsUnblockedWhenElementAdded();
                    Console.WriteLine("Press ENTER to EXIT the consumer component of Test _04_testTakeIsUnblockedWhenElementAdded();");
                    Console.ReadLine();
                    break;

                    case 5:
                    TEST = true;
                    Console.WriteLine("Press ENTER to execute the consumer component of Test _05_testPutTakeInt();");
                    Console.ReadLine();
                    consumer._05_testPutTakeInt();
                    Console.WriteLine("Press ENTER to EXIT the consumer component of Test _05_testPutTakInt();");
                    Console.ReadLine();
                    break;

                    case 6:
                    TEST = true;
                    Console.WriteLine("Press ENTER to execute the consumer component of Test _05_testPutTakeLong();");
                    Console.ReadLine();
```

```
consumer._05_testPutTakeLong();
Console.WriteLine("Press ENTER to EXIT the consumer component of Test _05_testPutTakeLong();");
Console.ReadLine();
break;

case 7:
TEST = false;
Console.WriteLine("Press ENTER to execute the consumer component of Test_07_testPutTakeString");
Console.ReadLine();
consumer._07_testPutTakeString();
Console.WriteLine("Press ENTER to EXIT the consumer component of Test _07_testPutTakeString");
Console.ReadLine();
break;

case 8:
TEST = false;
Console.WriteLine("Press ENTER to execute the consumer component of Test _08_testPutTake_fixed");
Console.ReadLine();
consumer._08_testPutTake_fixed();
Console.WriteLine("Press ENTER to EXIT the consumer component of Test _08_testPutTake_fixed");
Console.ReadLine();
break;

case 10:
TEST = true;
Console.WriteLine("Press ENTER to execute the consumer component of Test Group 00 - Integers");
Console.ReadLine();
consumer.test_group_00();
Console.WriteLine("Press ENTER to EXIT the consumer component of Test Group 00 - Integers");
Console.ReadLine();
break;

case 11:
TEST = true;
Console.WriteLine("Press ENTER to execute the consumer component of Test Group 01 - Longs");
Console.ReadLine();
consumer.test_group_01();
Console.WriteLine("Press ENTER to EXIT the consumer component of Test Group 01 - Longs");
Console.ReadLine();
break;

case 12:
TEST = true;
Console.WriteLine("Press ENTER to execute the consumer component of Test Group 02 - Strings");
Console.ReadLine();
```

```
                    consumer.test_group_02();
                    Console.WriteLine("Press ENTER to EXIT the consumer component of Test Group 02 - Strings");
                    Console.ReadLine();
                    break;

                    case 13:
                    TEST = true;
                    Console.WriteLine("Press ENTER to execute the consumer component of Test Group 03 - Structs");
                    Console.ReadLine();
                    consumer.test_group_03();
                    Console.WriteLine("Press ENTER to EXIT the consumer component of Test Group 03 - Structs");
                    Console.ReadLine();
                    break;

                    case 14:
                    TEST = true;
                    Console.WriteLine("Press ENTER to execute the consumer component of Test Groups 00, 01, 02 and 03");
                    Console.ReadLine();
                    consumer.test_group_00();
                    consumer.test_group_01();
                    consumer.test_group_02();
                    consumer.test_group_03();
                    Console.WriteLine("Press ENTER to EXIT the consumer component of Test Groups 00, 01, 02 and 03");
                    Console.ReadLine();
                    break;

                    default:
                    Console.WriteLine("No valid test selection was made. Shutting down...");
                    break;
                }
            }
            while (choice > 0);
        }
        catch (Exception ex)
        {
            // Ignore ex - We should have displayed it in the individual TEST that failed
            Console.Write(ex.Message);
        }
    }

    private static void CurrentDomain_UnhandledException(object sender, UnhandledExceptionEventArgs e)
    {
        // Set up uncaught exception handler in case some dodgy code throws a RunTimeException
        // This won't work if the exception is passed to some even more dodgy 3rd psrty code that swallows
        // the exception. Does work in the case of dodgy 3rd party rogue code like ActiveMQ which kindly throws
    // some kind of runtime exception if you don't have a 'geronimo' jar in your classpath when you try to
    // instantiate a connectionFactory or ActiveMQConnectionFactory
    // Java version looks like this - ASExceptionHandler UEH = new ASExceptionHandler();
    //                              Thread.setDefaultUncaughtExceptionHandler(UEH);
    // Java also has per-thread scheduler handlers set up using the same class
    Console.Write(e.ExceptionObject.ToString());
}

    public void Init(string channelType, int numberOfTrials)
    {
        // Configure all tests to be run on a queue or a stack type channel

        if (channelType.ToUpper() == "S")
        {
            initTestDataStructureType = DataStructureType.Stack;
        } else {
            initTestDataStructureType = DataStructureType.Queue;
        }

        initNoOfTrials = numberOfTrials;
        // These values are not used in the Consumers
        // maxIntRandomSeed = 1000;
        // maxLongRandomSeed = 1000000;
    }


    public void _03_testTakeBlocksWhenEmpty()
    {
        int LOCKUP_DETECT_TIMEOUT_MILLIS = 1000;
        int viewSize = 1000;
        int fileSize = 1000000;
        int capacity = 500;

        TEST = false;

        string QueueName = "_03_testTakeBlocksWhenEmpty";
        // BlockingCollection<int> mmq = new BlockingCollection<int>(new ConcurrentQueue<int>(), maxCount);
        MMChannel mmMain = null;

        // Create the Consumer thread with anonymous lambda expression
        Thread Consumer =
            new Thread(
                new ThreadStart(
                // Old way - replace lamda expression '() =>' with 'delegate'
                () =>
            {
                try
                {
                    char unused = mmMain.Take<char>();
                    Console.WriteLine("_03_testTakeBlocksWhenEmpty() = Fail - the test thread was not blocked in 'Take()'");
                }
                catch (ThreadInterruptedException success)
                {
                    Console.WriteLine("_03_testTakeBlocksWhenEmpty() = Pass - ThreadInterruptedException was thrown");
                    Console.WriteLine(success);
                    // DO NOT rethrow. Thes test was a success if Interrupted Exception was thrown
                }
                // Any other Exceptions we will not handle. Let them bubble up to the Main() method
            }
            )
        );

    // perform the test from the main thread
    try
    {
        mmMain = MMChannel.GetInstance(QueueName, fileSize, viewSize, capacity, DEBUG, TEST, initTestDataStructureType);

        Consumer.Start();
        Thread.Sleep(LOCKUP_DETECT_TIMEOUT_MILLIS);

        ControlData controlData = mmMain.MMFControlData;

        Consumer.Interrupt();
        Consumer.Join(LOCKUP_DETECT_TIMEOUT_MILLIS);
        Console.WriteLine("_03_testTakeBlocksWhenEmpty() = Join the main thread to the Consumer thread returned after {0} ms. Consumer thread alive = {1}",
            LOCKUP_DETECT_TIMEOUT_MILLIS, Consumer.IsAlive);
    }
    catch (Exception unexpected)
    {
        Console.Write(unexpected);
        throw;
    }
    finally
    {
        Thread.Sleep(1000);
        mmMain.Report();
        mmMain.Dispose();
    }
}
```

```
// This is the default layout that the compiler would use anyway
[StructLayout(LayoutKind.Sequential, Pack = 1, CharSet = CharSet.Unicode)]
unsafe struct _04_MMData
{
    public int Value;
    public char Letter;
    public int NumbersLength;
    public fixed float Numbers[10];
    public int TextLength;
    public fixed char Text[100];
}
unsafe struct _04_args
{
    public MMChannel mQueue;
    public _04_MMData dData;
}
private void _04_DequeueData(_04_args arg)
{
    try
    {
        _04_MMData unused = arg.mQueue.Take<_04_MMData>();
        StringBuilder numbers = new StringBuilder();
        String text;
        for (int i = 0; i < unused.NumbersLength; i++)
        {
            unsafe
            {
                numbers.Append(unused.Numbers[i] + ", ");
            }
        }

        char[] txt = new char[unused.TextLength];
        for (int i = 0; i < unused.TextLength; i++)
        {
            unsafe { txt[i] = unused.Text[i]; }
        }
        text = new String(txt);

        Console.WriteLine("_04_testTakeIsUnblockedWhenElementAdded() = data item dequeued = \n'{0}', \n'{1}', \n'{2}', \n'{3}'",
            unused.Value, unused.Letter, numbers, text);
    }
    catch (Exception unexpected)
    {
        Console.Write(unexpected);
        throw;
    }
}

public void _04_testTakeIsUnblockedWhenElementAdded()
{
    int LOCKUP_DETECT_TIMEOUT_MILLIS = 1000;
    int initialCount = 0;
    int viewSize = 1000;
    int fileSize = 1000000;
    int capacity = 500;

    TEST = false;

    MMChannel mmMain = null;

    _04_MMData data = default(_04_MMData); // A struct containing data to be enqueued and dequeued
    _04_args arg;    // A struct containing the data struct and the Memory Mapped File View Accessor to be passed as a parameter
    // to a parameterized threadstart

    string QueueName = "_04_testTakeIsUnblockedWhenElementAdded";
    mmMain = MMChannel.GetInstance(QueueName, fileSize, viewSize, capacity, DEBUG, TEST, initTestDataStructureType);

    // Assign the data and the view accessor to the struct that we will use for the parameterized threadstart
    arg.mQueue = mmMain;
    arg.dData = data;

    // Create the Consumer threads with anonymous lamda expression
    Thread Consumer_1 = new Thread(() => _04_DequeueData(arg));
    Thread Consumer_2 = new Thread(() => _04_DequeueData(arg));

    // perform the test from the main thread
    try
    {
        ControlData controlData = mmMain.MMFControlData;

        // verify that the queue is empty
        Console.WriteLine("_04_testTakeIsUnblockedWhenElementAdded() Queue is empty? = (Count {1} == initialCount {2}) = {0}",
             controlData.totalItemsEnqueued - controlData.totalItemsDequeued == initialCount,
             controlData.totalItemsEnqueued - controlData.totalItemsDequeued, initialCount);

        // Start a thread to dequeue an element
        Consumer_1.Start();

        // Wait for a period for the thread to die
```

```
                Thread.Sleep(LOCKUP_DETECT_TIMEOUT_MILLIS);
                Consumer_1.Join(LOCKUP_DETECT_TIMEOUT_MILLIS);
                // Verify the thread has died
                Console.WriteLine("_04_testTakeIsUnblockedWhenElementAdded() = Join main thread to Consumer thread returned before {0} ms and Isalive = {1}",
                    LOCKUP_DETECT_TIMEOUT_MILLIS, Consumer_1.IsAlive);

                Console.WriteLine("verify that the queue is empty");
                controlData = mmMain.MMFControlData;

                Console.WriteLine("_04_testTakeIsUnblockedWhenElementAdded() = (Count {1} == initialCount {2}) = {0} after dequeueing {3} items\n",
                    controlData.totalItemsEnqueued - controlData.totalItemsDequeued == initialCount,
                    controlData.totalItemsEnqueued - controlData.totalItemsDequeued, initialCount, controlData.totalItemsDequeued);

                Console.WriteLine("Press ENTER to run another consumer to block on the empty queue");
                Console.ReadLine();

                Consumer_2.Start();
                // Wait for a period for the thread to die
                Thread.Sleep(LOCKUP_DETECT_TIMEOUT_MILLIS);
                Consumer_2.Join(LOCKUP_DETECT_TIMEOUT_MILLIS);
                Console.WriteLine("Verify the Consumer thread has NOT died");
                Console.WriteLine("_04_testTakeIsUnblockedWhenElementAdded() = Joining the main thread to the Consumer thread returned before {0} ms and Isalive =
{1}\n",
                    LOCKUP_DETECT_TIMEOUT_MILLIS, Consumer_2.IsAlive);

                Console.WriteLine("Go to the producer and Press ENTER to unblock the consumer waiting on the empty queue");
                Console.ReadLine();

                Console.WriteLine("Press ENTER to FINISH");
                Console.ReadLine();

            }
            catch (Exception unexpected)
            {
                Console.Write(unexpected);
                throw;
            }
            finally
            {
                Thread.Sleep(1000);
                mmMain.Report();
                mmMain.Dispose();
            }
        }
```

```
public void _05_testPutTakeInt()
{
    // Test that the queue performs correctly under unpredictable concurrent access by using multiple threads to
    // to to perform Put and Take operations over a period of time and that nothing wnet wrong

    TEST = true;

    Console.WriteLine("\nStart of Test Run No. {0} in Test Suite No. {1}\n", ++initTestRunNumber, initTestSuiteNumber);

    MMChannel mmMain = null;

    try
    {
        int capacity = 10, fileSize = 1000000, viewSize = 1000;
        string QueueName = "_05_testPutTakeInt";

        // If only performing a small number of trials then GC could impact the timing tests so try and request it beforehand
        // In the case of a small number of trials, hopefully GC won't be required again before the end of the test
        System.GC.Collect();

        // INFO Cannot use the Property (get/set) with an Interlocked -
        // Store the value of the computed checksums here using Interlocked to ensure atomicty
        long takeSum = 0;
        // Start and end times of the test run
        long timerStartTime = 0, timerEndTime = 0;

        // test parameters
        int nPairs = 10, nTrials = initNoOfTrials;

        // Create the MMChannel which will instantiate the memory mapped files, mutexes, semaphores etc ...
        mmMain = MMChannel.GetInstance(QueueName, fileSize, viewSize, capacity, DEBUG, TEST, initTestDataStructureType);


        #region Barrier and Barrier Action declaration
        // The barrier will wait for the test runner thread plus a producer and consumer each for the number of pairs
        // Waits for them all to be ready at the start line and again at the finish
        Barrier _barrier = new Barrier(nPairs  + 1,
            actionDelegate =>
            {
                // Check to see if the start time variable has been assigned or still = zero
                // If false then this is the first execution of the barrier action (at the start). Otherwise it is the
                // second execution 9at the finish)
                const long zeroFalse_1 = 0; // Not passed by ref so no need to be assignable
        bool started = Interlocked.Equals(timerStartTime, zeroFalse_1);
        started = !started;

        // Store the start time or the end time depending on which execution this is
        long t = DateTime.Now.Ticks;
        if (!started)
        {
            Interlocked.Exchange(ref timerStartTime, t);
        }
        else
        {
            Interlocked.Exchange(ref timerEndTime, t);
        }
    }
);
#endregion Barrier and Barrier Action declaration

// create pairs of threads to put and take items to/from the queue
// Including the test runner thread the barriers will wait for nPairs * 2 + 1 ther
for (int i = 0; i < nPairs; i++)
{
    #region Consumer Lamda declaration

    new Thread(
        new ThreadStart(
        // Old way - replace lamda expression '() =>' with 'delegate'
        () =>
        {
            try
            {
                long result = 0;

                // Wait at the barrier (start line) until all test threads have been created and are ready to go
                _barrier.SignalAndWait();

                // The Producer's sum should equal the Consumer's sum at the end of the test
                for (int k = nTrials; k > 0; --k)
                {
                    result += Convert.ToInt64(mmMain.Take<long>());
                }

                // Atomically store the computed checksum
                Interlocked.Add(ref takeSum, result);

                // Wait at the barrier (finish line) until all test threads have been finished
                _barrier.SignalAndWait();

                //=================================================================================
                // throw new Exception("Test Exception handling!!");
                //=================================================================================
            }
            catch (Exception unexpected)
            {
                Console.WriteLine("_05_testPutTakeInt() Consumers = An unexpected Exception was thrown");
                Console.WriteLine(unexpected);
                throw;
            }
        }
    })).Start();
    #endregion Consumer Lamda declaration
}

_barrier.SignalAndWait();   // Wait for all the threads to be ready
_barrier.SignalAndWait();   // Wait for all the threads to finish

// calculate the number of ticks elapsed during the test run
long elapsedTime = Interlocked.Read(ref timerEndTime) - Interlocked.Read(ref timerStartTime);
Console.WriteLine("Intermediate Result of _05_testPutTakeInt() - elapsed time = {0} timer ticks for {1} producer/consumer pairs and {2} Messages",
    elapsedTime, nPairs, nTrials);

// Calculate the number of ticks per item enqueued and dequeued - the throughput of the queue
// A single tick represents one hundred nanoseconds or one ten-millionth of a second.
// There are 10,000 ticks in a millisecond.
long ticksPerItem = elapsedTime / (nPairs * (long)nTrials);
TimeSpan elapsedSpan = new TimeSpan(ticksPerItem);
double milliSeconds = elapsedSpan.TotalMilliseconds;
long nanoSeconds = ticksPerItem * 100;
long throughput = 1000000000 / nanoSeconds;

// Compares the checksum values computed to determine if the data enqueued was exactly the data dequeued
Console.WriteLine("_05_testPutTakeInt() = (data enqueued = {0} after {1} trials each by {2} pairs of producers/consumers",
    Interlocked.Read(ref takeSum), nTrials, nPairs);

Console.WriteLine("_05_testPutTakeInt() = (Average latency = {0} timer ticks <= Threshold value {1}) = {2}",
    Interlocked.Read(ref ticksPerItem),
    AVERAGE_THROUGHPUT_THRESHOLD_TICKS,
    Interlocked.Read(ref ticksPerItem) <= AVERAGE_THROUGHPUT_THRESHOLD_TICKS);

Console.WriteLine("_05_testPutTake Throughput = {0} messages per second ", throughput);
```

```
        Console.WriteLine("_05_testPutTake {0} timer ticks = {1} nanoseconds or {2} milliseconds",
            ticksPerItem, nanoSeconds, milliSeconds);

        Console.WriteLine("\nEnd of Test Run No. {0} in Test Suite No. {1}\n", initTestRunNumber, initTestSuiteNumber);
    }
    catch (Exception unexpected)
    {
        Console.Write(unexpected);
        throw;
    }
    finally
    {
        // Temporarily delay disposing the queue and its IPC artefacts to allow the consumers to finish draining the queue
        // This will be fixed by waiting in an interrupible loop for the mutex inside the queue and checking if shutdown
        Thread.Sleep(1000);
        mmMain.Report();
        mmMain.Dispose();
    }
}


public void _05_testPutTakeLong()
{
    // Test that the queue performs correctly under unpredictable concurrent access by using multiple threads to
    // to to perform Put and Take operations over a period of time and that nothing wnet wrong

    TEST = true;

    Console.WriteLine("\nStart of Test Run No. {0} in Test Suite No. {1}\n", ++initTestRunNumber, initTestSuiteNumber);

    MMChannel mmMain = null;

    try
    {
        int capacity = 10, fileSize = 1000000, viewSize = 1000;
        string QueueName = "_05_testPutTakeLong";

        // If only performing a small number of trials then GC could impact the timing tests so try and request it beforehand
        // In the case of a small number of trials, hopefully GC won't be required again before the end of the test
        System.GC.Collect();

        // INFO Cannot use the Property (get/set) with an Interlocked -
        // Store the value of the computed checksums here using Interlocked to ensure atomicty
        long takeSum = 0;
        // Start and end times of the test run
```

```
long timerStartTime = 0, timerEndTime = 0;

// test parameters
int nPairs = 10, nTrials = initNoOfTrials;

// Create the MMChannel which will instantiate the memory mapped files, mutexes, semaphores etc ...
mmMain = MMChannel.GetInstance(QueueName, fileSize, viewSize, capacity, DEBUG, TEST, initTestDataStructureType);


#region Barrier and Barrier Action declaration
// The barrier will wait for the test runner thread plus a producer and consumer each for the number of pairs
// Waits for them all to be ready at the start line and again at the finish
Barrier _barrier = new Barrier(nPairs + 1,
    actionDelegate =>
    {
        // Check to see if the start time variable has been assigned or still = zero
        // If false then this is the first execution of the barrier action (at the start). Otherwise it is the
        // second execution 9at the finish)
        const long zeroFalse_1 = 0; // Not passed by ref so no need to be assignable
        bool started = Interlocked.Equals(timerStartTime, zeroFalse_1);
        started = !started;

        // Store the start time or the end time depending on which execution this is
        long t = DateTime.Now.Ticks;
        if (!started)
        {
            Interlocked.Exchange(ref timerStartTime, t);
        }
        else
        {
            Interlocked.Exchange(ref timerEndTime, t);
        }
    }
);
#endregion Barrier and Barrier Action declaration

// create pairs of threads to put and take items to/from the queue
// Including the test runner thread the barriers will wait for nPairs * 2 + 1 ther
for (int i = 0; i < nPairs; i++)
{
    #region Consumer Lamda declaration

    new Thread(
        new ThreadStart(
        // Old way - replace lamda expression '() =>' with 'delegate'
```

```
                () =>
                {
                    try
                    {
                        long result = 0;

                        // Wait at the barrier (start line) until all test threads have been created and are ready to go
                        _barrier.SignalAndWait();

                        // The Producer's sum should equal the Consumer's sum at the end of the test
                        for (int k = nTrials; k > 0; --k)
                        {
                            result += mmMain.Take<long>();
                            // result += Convert.ToInt64(mmMain.Take<long>());
                        }

                        // Atomically store the computed checksum
                        Interlocked.Add(ref takeSum, result);

                        // Wait at the barrier (finish line) until all test threads have been finished
                        _barrier.SignalAndWait();

                    }
                    catch (Exception unexpected)
                    {
                        Console.Write(unexpected);
                        throw;
                    }
                }
            )).Start();
            #endregion Consumer Lamda declaration
        }

        _barrier.SignalAndWait();   // Wait for all the threads to be ready
        _barrier.SignalAndWait();   // Wait for all the threads to finish

        // calculate the number of ticks elapsed during the test run
        long elapsedTime = Interlocked.Read(ref timerEndTime) - Interlocked.Read(ref timerStartTime);
        Console.WriteLine("Intermediate Result of _05_testPutTakeLong() - elapsed time = {0} timer ticks for {1} producer/consumer pairs and {2} Messages",
            elapsedTime, nPairs, nTrials);

        // Calculate the number of ticks per item enqueued and dequeued - the throughput of the queue
        // A single tick represents one hundred nanoseconds or one ten-millionth of a second.
        // There are 10,000 ticks in a millisecond.
        long ticksPerItem = elapsedTime / (nPairs * (long)nTrials);
        TimeSpan elapsedSpan = new TimeSpan(ticksPerItem);
        double milliSeconds = elapsedSpan.TotalMilliseconds;
        long nanoSeconds = ticksPerItem * 100;
        long throughput = 1000000000 / nanoSeconds;

        // Compares the checksum values computed to determine if the data enqueued was exactly the data dequeued
        Console.WriteLine("_05_testPutTakeLong() = (data enqueued = {0} after {1} trials each by {2} pairs of producers/consumers",
            Interlocked.Read(ref takeSum), nTrials, nPairs);

        Console.WriteLine("_05_testPutTakeLong() = (Average latency = {0} timer ticks <= Threshold value {1}) = {2}",
            Interlocked.Read(ref ticksPerItem),
            AVERAGE_THROUGHPUT_THRESHOLD_TICKS,
            Interlocked.Read(ref ticksPerItem) <= AVERAGE_THROUGHPUT_THRESHOLD_TICKS);

        Console.WriteLine("_05_testPutTakeLong Throughput = {0} messages per second ", throughput);

        Console.WriteLine("_05_testPutTakeLong {0} timer ticks = {1} nanoseconds or {2} milliseconds",
            ticksPerItem, nanoSeconds, milliSeconds);

        Console.WriteLine("\nEnd of Test Run No. {0} in Test Suite No. {1}\n", initTestRunNumber, initTestSuiteNumber);
    }
    catch (Exception unexpected)
    {
        Console.Write(unexpected);
        throw;
    }
    finally
    {
        // Temporarily delay disposing the queue and its IPC artefacts to allow the consumers to finish draining the queue
        Thread.Sleep(1000);
        mmMain.Report();
        mmMain.Dispose();
    }
}

public void _07_testPutTakeString()
{
    // Test that the queue performs correctly under unpredictable concurrent access by using multiple threads to
    // to to perform Put and Take operations over a period of time and that nothing went wrong

    TEST = false;
    MMChannel mmMain = null;

    Console.WriteLine("\nStart of Test Run No. {0} in Test Suite No. {1}\n", ++initTestRunNumber, initTestSuiteNumber);
```

```
String applicationInstance = UTCutil.GetInstanceNameForProcessId(Process.GetCurrentProcess().Id);
Dictionary<String, PerformanceCounter> map = UTCutil.ReadKeyMemoryAndHandlePerformanceCounters(applicationInstance);

PerformanceCounter all_heaps_counter;
map.TryGetValue(UTCutil.performanceCounter_bytes_in_all_heaps, out all_heaps_counter);
String name = all_heaps_counter.CounterName.ToString();

try
{
    int capacity = 500, fileSize = 1000000, viewSize = 1000;
    string QueueName = "_07_testPutTakeString";

    // If only performing a small number of trials then GC could impact the timing tests so try and request it beforehand
    // In the case of a small number of trials, hopefully GC won't be required again before the end of the test
    System.GC.Collect();

    // INFO Cannot use the Property (get/set) with an Interlocked -
    // Store the value of the computed checksums here using Interlocked to ensure atomicty
    long takeSum = 0;
    // Start and end times of the test run
    long timerStartTime = 0, timerEndTime = 0;

    // test parameters
    int nPairs = 10, nTrials = initNoOfTrials;

    Random rand = new Random();

    // Create the MMChannel which will instantiate the memory mapped files, mutexes, semaphores etc ...
    mmMain = MMChannel.GetInstance(QueueName, fileSize, viewSize, capacity, DEBUG, TEST, initTestDataStructureType);

    #region Barrier and Barrier Action declaration
    // The barrier will wait for the test runner thread plus a producer and consumer each for the number of pairs
    // Waits for them all to be ready at the start line and again at the finish
    Barrier _barrier = new Barrier(nPairs + 1,
        actionDelegate =>
        {
            // Check to see if the start time variable has been assigned or still = zero
            // If false then this is the first execution of the barrier action (at the start). Otherwise it is the
            // second execution at the finish)
            const long zeroFalse_1 = 0; // Not passed by ref so no need to be assignable
            bool started = Interlocked.Equals(timerStartTime, zeroFalse_1);
            started = !started;

            // Store the start time or the end time depending on which execution this is
            long t = DateTime.Now.Ticks;
        if (!started)
        {
            Interlocked.Exchange(ref timerStartTime, t);
        }
        else
        {
            Interlocked.Exchange(ref timerEndTime, t);
        }
    }
);
#endregion Barrier and Barrier Action declaration

// create pairs of threads to put and take items to/from the queue
// Including the test runner thread the barriers will wait for nPairs * 2 + 1 there
for (int i = 0; i < nPairs; i++)
{
    #region Consumer Lamda declaration

    new Thread(
        new ThreadStart(
        // Old way - replace lamda expression '() =>' with 'delegate'
        () =>
        {
            try
            {
                long result = 0;

                // Wait at the barrier (start line) until all test threads have been created and are ready to go
                _barrier.SignalAndWait();

                // take the data from the queue as Strings, convert back to ints and sum them
                // The Producer's sum should equal the Consumer's sum at the end of the test
                for (int k = nTrials; k > 0; --k)
                {
                    char[] data;
                    int numItems = mmMain.Take<char>(out data);
                    string retval = new string(data);
                    result += Convert.ToInt64(retval);
                }

                // Atomically store the computed checksum
                // Comment out for Test 01 as we have already incremented it
                Interlocked.Add(ref takeSum, result);

                // Wait at the barrier (finish line) until all test threads have been finished
                _barrier.SignalAndWait();

            }
            catch (Exception unexpected)
            {
                Console.Write(unexpected);
                throw;
            }
        }
    )).Start();
    #endregion Consumer Lamda declaration

}

int THRESHOLD = 1000; long diff;

#region heap profiling testing notes
// I got this off Java Concurrency in Practice, chap 12 Testing Concurrent Programs Page 258
// It doesn't work as written though!
// Generally, the heap size after testing was fraction of the size before the test
// Obviously, processing these huge messages has triggered a GC during the test
// Even then you would expect this to result in a false positive where the two snapshot were similar even if
// your code was leaking memory so the most likely explanation seems to be that NUnit itself is creating objects
// which have not yet been reclaimed before the test starts
// Requesting a GC before the initial snapshot solves the problem but you have to accept that you cannot
// completely control managed memory allocation and at some point the GC will probably ignore your request
// and the test will fail
#endregion heap profiling testing notes

System.GC.Collect();
long heapSizeBeforeTest = Convert.ToInt64(UTCutil.GetCounterValue(all_heaps_counter));

_barrier.SignalAndWait();   // Wait for all the threads to be ready
_barrier.SignalAndWait();   // Wait for all the threads to finish

System.GC.Collect();
long heapSizeAfterTest = Convert.ToInt64(UTCutil.GetCounterValue(all_heaps_counter));
diff = Math.Abs(heapSizeBeforeTest - heapSizeAfterTest);

Console.WriteLine("Result of TestLeak() Heap size at end of run = {0}, Heap size at start of run = {1} Difference = {2}, Passed = {3}",
    heapSizeAfterTest, heapSizeBeforeTest, diff, diff <= THRESHOLD);

// calculate the number of ticks elapsed during the test run
long elapsedTime = Interlocked.Read(ref timerEndTime) - Interlocked.Read(ref timerStartTime);
Console.WriteLine("Intermediate Result of _07_testPutTakeString() - elapsed time = {0} timer ticks for {1} producer/consumer pairs and {2} Messages",
            elapsedTime, nPairs, nTrials);

        // Calculate the number of ticks per item enqueued and dequeued - the throughput of the queue
        // A single tick represents one hundred nanoseconds or one ten-millionth of a second.
        // There are 10,000 ticks in a millisecond.
        long ticksPerItem = elapsedTime / (nPairs * (long)nTrials);
        TimeSpan elapsedSpan = new TimeSpan(ticksPerItem);
        double milliSeconds = elapsedSpan.TotalMilliseconds;
        long nanoSeconds = ticksPerItem * 100;
        long throughput = 1000000000 / nanoSeconds;

        // Compares the checksum values computed to determine if the data enqueued was exactly the data dequeued
        Console.WriteLine("1st Result of _07_testPutTakeString() = (data dequeued = {0} after {1} trials each by {2} pairs of producers/consumers",
            Interlocked.Read(ref takeSum), nTrials, nPairs);

        Console.WriteLine("2nd Result of _07_testPutTakeString() = (Average latency = {0} timer ticks <= Threshold value {1}) = {2}",
            Interlocked.Read(ref ticksPerItem),
            AVERAGE_THROUGHPUT_THRESHOLD_TICKS,
            Interlocked.Read(ref ticksPerItem) <= AVERAGE_THROUGHPUT_THRESHOLD_TICKS);

        Console.WriteLine("_07_testPutTakeString Throughput = {0} messages per second ", throughput);

        Console.WriteLine("_07_testPutTakeString n.b. {0} timer ticks = {1} nanoseconds or {2} milliseconds",
            ticksPerItem, nanoSeconds, milliSeconds);

        Console.WriteLine("\nEnd of Test Run No. {0} in Test Suite No. {1}\n", initTestRunNumber, initTestSuiteNumber);
    }
    catch (Exception unexpected)
    {
        Console.Write(unexpected);
        throw;
    }
    finally
    {
        // Temporarily delay disposing the queue and its IPC artefacts to allow the consumers to finish draining the queue
        // This will be fixed by waiting in an interrupible loop for the mutex inside the queue and checking if shutdown
        Thread.Sleep(1000);
        mmMain.Report();
        mmMain.Dispose();
    }
}



// This is the default layout that the compiler would use anyway
```

```
[StructLayout(LayoutKind.Sequential, Pack = 1, CharSet = CharSet.Unicode)]
unsafe struct _08_MMData
{
    public int TextLength;
    public fixed char Text[100];
}

public void _08_testPutTake_fixed()
{
    TEST = false;

    // Test that the queue performs correctly under unpredictable concurrent access by using multiple threads to
    // to to perform Put and Take operations over a period of time and that nothing wnet wrong

    MMChannel mmMain = null;

    Console.WriteLine("\nStart of Test Run No. {0} in Test Suite No. {1}\n", ++initTestRunNumber, initTestSuiteNumber);

    try
    {
        int capacity = 500, fileSize = 1000000, viewSize = 1000;
        string QueueName = "_08_testPutTake_fixed";

        // If only performing a small number of trials then GC could impact the timing tests so try and request it beforehand
        // In the case of a small number of trials, hopefully GC won't be required again before the end of the test
        System.GC.Collect();

        // INFO Cannot use the Property (get/set) with an Interlocked -
        // Store the value of the computed checksums here using Interlocked to ensure atomicty
        long takeSum = 0;
        // Start and end times of the test run
        long timerStartTime = 0, timerEndTime = 0;

        // test parameters
        // Performance nPairs = 10, capacity = 10, nTrials = 1,000,000 = BlockingCollection = 60s, MMQueue = 254s
        int nPairs = 10, nTrials = initNoOfTrials;

        Random rand = new Random();

        // Create the MMChannel which will instantiate the memory mapped files, mutexes, semaphores etc ...
        mmMain = MMChannel.GetInstance(QueueName, fileSize, viewSize, capacity, DEBUG, TEST, initTestDataStructureType);

        #region Barrier and Barrier Action declaration
        // The barrier will wait for the test runner thread plus a producer and consumer each for the number of pairs
        // Waits for them all to be ready at the start line and again at the finish
```

```
Barrier _barrier = new Barrier(nPairs + 1,
    actionDelegate =>
    {
        // Check to see if the start time variable has been assigned or still = zero
        // If false then this is the first execution of the barrier action (at the start). Otherwise it is the
        // second execution 9at the finish)
        const long zeroFalse_1 = 0; // Not passed by ref so no need to be assignable
        bool started = Interlocked.Equals(timerStartTime, zeroFalse_1);
        started = !started;

        // Store the start time or the end time depending on which execution this is
        long t = DateTime.Now.Ticks;
        if (!started)
        {
            Interlocked.Exchange(ref timerStartTime, t);
        }
        else
        {
            Interlocked.Exchange(ref timerEndTime, t);
        }
    }
);
#endregion Barrier and Barrier Action declaration

// create pairs of threads to put and take items to/from the queue
// Including the test runner thread the barriers will wait for nPairs * 2 + 1 ther
for (int i = 0; i < nPairs; i++)
{
    #region Consumer Lamda declaration

    new Thread(
        new ThreadStart(
        // Old way - replace lamda expression '() =>' with 'delegate'
        () =>
        {
            try
            {
                long result = 0;

                // Wait at the barrier (start line) until all test threads have been created and are ready to go
                _barrier.SignalAndWait();

                // take the data from the queue as Strings, convert back to ints and sum them
                // The Producer's sum should equal the Consumer's sum at the end of the test
                for (int k = nTrials; k > 0; --k)
                {
                    _08_MMData data = mmMain.Take<_08_MMData>();

                    String decodedData;

                    char[] txt = new char[data.TextLength];
                    for (int m = 0; m < data.TextLength; m++)
                    {
                        unsafe { txt[m] = data.Text[m]; }
                    }
                    decodedData = new string(txt);

                    result += Convert.ToInt64(decodedData);
                }

                // Atomically store the computed checksum
                // Comment out for Test 01 as we have already incremented it
                Interlocked.Add(ref takeSum, result);

                // Wait at the barrier (finish line) until all test threads have been finished
                _barrier.SignalAndWait();

            }
            catch (Exception unexpected)
            {
                Console.Write(unexpected);
                throw;
            }
        }
    )).Start();
    #endregion Consumer Lamda declaration
}

_barrier.SignalAndWait();   // Wait for all the threads to be ready
_barrier.SignalAndWait();   // Wait for all the threads to finish

// calculate the number of ticks elapsed during the test run
long elapsedTime = Interlocked.Read(ref timerEndTime) - Interlocked.Read(ref timerStartTime);
Console.WriteLine("Intermediate Result of _08_testPutTake_fixed() - elapsed time = {0} timer ticks for {1} producer/consumer pairs and {2} Messages",
    elapsedTime, nPairs, nTrials);

// Calculate the number of ticks per item enqueued and dequeued - the throughput of the queue
// A single tick represents one hundred nanoseconds or one ten-millionth of a second.
// There are 10,000 ticks in a millisecond.
long ticksPerItem = elapsedTime / (nPairs * (long)nTrials);
```

```
        TimeSpan elapsedSpan = new TimeSpan(ticksPerItem);
        double milliSeconds = elapsedSpan.TotalMilliseconds;
        long nanoSeconds = ticksPerItem * 100;
        long throughput = 1000000000 / nanoSeconds;

        // Compares the checksum values computed to determine if the data enqueued was exactly the data dequeued
        Console.WriteLine("1st Result of _08_testPutTake_fixed() = (data dequeued = {0} after {1} trials each by {2} pairs of producers/consumers",
            Interlocked.Read(ref takeSum), nTrials, nPairs);

        Console.WriteLine("2nd Result of _08_testPutTake_fixed() = (Average latency = {0} timer ticks <= Threshold value {1}) = {2}",
            Interlocked.Read(ref ticksPerItem),
            AVERAGE_THROUGHPUT_THRESHOLD_TICKS,
            Interlocked.Read(ref ticksPerItem) <= AVERAGE_THROUGHPUT_THRESHOLD_TICKS);

        Console.WriteLine("_08_testPutTake_fixed Throughput = {0} messages per second ", throughput);

        Console.WriteLine("_08_testPutTake_fixed {0} timer ticks = {1} nanoseconds or {2} milliseconds",
            ticksPerItem, nanoSeconds, milliSeconds);

        Console.WriteLine("\nEnd of Test Run No. {0} in Test Suite No. {1}\n", initTestRunNumber, initTestSuiteNumber);

    }
    catch (Exception unexpected)
    {
        Console.Write(unexpected);
        throw;
    }
    finally
    {
        // Temporarily delay disposing the queue and its IPC artefacts to allow the consumers to finish draining the queue
        // This will be fixed by waiting in an interrupible loop for the mutex inside the queue and checking if shutdown
        Thread.Sleep(1000);
        mmMain.Report();
        mmMain.Dispose();
    }
}

#region TEST Groups

public void test_group_00()
{
    // No catch block as we've already displayed any exceptions and re-thrown in the tests themselves
    // Allow to bubble up to Main() where they will be caught and the program will exit
    int numTestRuns = 100;
```

```
            Console.WriteLine("Start Test Suite No. {0} using Integers - {1} test runs\n", ++initTestSuiteNumber, numTestRuns);

            for (int i = 0; i < numTestRuns; i++)
            {
                _05_testPutTakeInt();
            }

            Console.WriteLine("End Test Suite No. {0}\n", initTestSuiteNumber);
        }

        public void test_group_01()
        {
            // No catch block as we've already displayed any exceptions and re-thrown in the tests themselves
            // Allow to bubble up to Main() where they will be caught and the program will exit

            int numTestRuns = 100;

            Console.WriteLine("Start Test Suite No. {0} using Long Integers - {1} test runs\n", ++initTestSuiteNumber, numTestRuns);

            for (int i = 0; i < numTestRuns; i++)
            {
                _05_testPutTakeLong();
            }

            Console.WriteLine("End Test Suite No. {0}\n", initTestSuiteNumber);
        }

        public void test_group_02()
        {
            // No catch block as we've already displayed any exceptions and re-thrown in the tests themselves
            // Allow to bubble up to Main() where they will be caught and the program will exit

            int numTestRuns = 100;

            Console.WriteLine("Start Test Suite No. {0} using Strings - {1} test runs\n", ++initTestSuiteNumber, numTestRuns);

            for (int i = 0; i < numTestRuns; i++)
            {
                _07_testPutTakeString();
            }

            Console.WriteLine("End Test Suite No. {0}\n", initTestSuiteNumber);
        }

        public void test_group_03()
```

```
            {
                // No catch block as we've already displayed any exceptions and re-thrown in the tests themselves
                // Allow to bubble up to Main() where they will be caught and the program will exit

                int numTestRuns = 100;

                Console.WriteLine("Start Test Suite No. {0} using Structs inc. Fixed Arrays - {1} test runs\n", ++initTestSuiteNumber, numTestRuns);

                for (int i = 0; i < numTestRuns; i++)
                {
                    _08_testPutTake_fixed();
                }

                Console.WriteLine("End Test Suite No. {0}\n", initTestSuiteNumber);
            }

            #endregion TEST groups

    }
}
```

Inter Process producer-Consumer Queue Test Harness

ProducerTestRunner.cs

```
/*********************************************************************************************************************************************************
 * The goal of the AlphaSystematics Project is create an open-source system for forward-testing systematic strategies with live market data and trade feeds.
 * It enables strategies developed in Excel to be connected to trading venues via industry standard FIX messaging.
 *
 * Copyright (C) 2009  Antonio Tapper. www.alphasystematics.org

 * This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

 * This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more details.
 * You should have received a copy of the GNU General Public License along with this program.  If not, see <http://www.gnu.org/licenses/>.
****************************************************************************************************************************************************************/

using System;
using System.Collections.Generic;
using System.IO;
using System.IO.MemoryMappedFiles;
using System.Text;
using System.Threading;
using com.alphaSystematics.concurrency;
using System.Linq;
using System.Threading.Tasks;
using System.Collections.Concurrent;
using System.Diagnostics;
using System.Runtime.InteropServices;

namespace TestMMFile_Source
{
    class ProducerTestRunner
    {
        // The hardest part of writing tests is that when they fail you don't know if it is the test or the application
        // thats broken unless you have confidence that the tests themselves have been tested thoroughly
        // in this case we are lucky in that we are trying to mimic the functionality of an existing library class but extend
        // it to use inter process.
        // We can drop in the library class here in order to test the test because we have confidence that the library class
        // works so if the tests fail when using library class then the tests are broken - For NUnit tests in a single process

        static DataStructureType initTestDataStructureType = default(DataStructureType);
        const long AVERAGE_THROUGHPUT_THRESHOLD_TICKS = 1000;
        int initNoOfTrials = 0; int initTestRunNumber = 0; int initTestSuiteNumber = 0; int maxLongRandomSeed = 0; int maxIntRandomSeed = 0;
        const int defaultNoOfTrials = 1000000;
        const bool DEBUG = true; static bool TEST = false;

        static int Menu()
        {
            string result = ""; int choice = 0; bool valid = false;

            while (!valid)
            {
                Console.Clear();
                Console.WriteLine("Memory Mapped Message Channel test suite (Producers). Please choose from the following options:\n");

                Console.WriteLine("1: Test menu for the Producers\n");
                // Console.WriteLine("1: Test that the queue is empty when constructed\n");
                // Console.WriteLine("2: Test that the queue is full after Puts and empty after Takes\n");
                // sole.WriteLine("3: Test that the Take method blocks when the queue is empty\n");
                Console.WriteLine("4: Test that the Take method is unblocked when an item is added\n");
                Console.WriteLine("5: Test Put and Take methods with Int data and equal numbers of producers and consumers\n");
                Console.WriteLine("6: Test Put and Take methods with Long data and equal numbers of producers and consumers\n");
                Console.WriteLine("7: Test Put and Take methods with array data (chars) and equal numbers of producers and consumers\n");
                Console.WriteLine("8: Test Put and Take methods with struct data and equal numbers of producers and consumers\n");
                Console.WriteLine("10: Execute Test Group No. 00 - 1 Billion Integers\n");
                Console.WriteLine("11: Execute Test Group No. 01 - 1 Billion Longs\n");
                Console.WriteLine("12: Execute Test Group No. 02 - 1 Billion Strings\n");
                Console.WriteLine("13: Execute Test Group No. 03 - 1 Billion Structs\n");
                Console.WriteLine("14: Execute Test Groups 00, 01, 02 and 03\n");

                Console.WriteLine("Q: Quit\n");

                // get the 1st character of input and quit if it is "Q"
                result = Console.ReadLine();
                if (result.ToUpper() == "Q") { result = "99"; }

                try
                {
                    choice = int.Parse(result);
                }
                catch (ArgumentException) { }
                catch (FormatException) { }

                switch (choice)
                {
                    case 0:
                        Console.WriteLine("Quitting test harness {0} please wait...", result);
                        valid = true;
                        break;
                    case 1:
                    // case 2:
                    // case 3:
                    case 4:
                    case 5:
                    case 6:
                    case 7:
                    case 8:
                    case 10:
                    case 11:
                    case 12:
                    case 13:
                    case 14:
                        Console.WriteLine("Executing test {0} please wait...", result);
```

```
                valid = true;
                break;

            default:
                Console.WriteLine("Invalid selection {0}. Please select 1, 4, 5, 6, 7, 8, 9, 10, 11. 12, 13, 14  or Quit.\n\n\n\n\n", result);
                break;
        }
    }
    return choice;
}

static string queueOrStack()
{
    Console.WriteLine("Please choose to test a Queue or a Stack (Default = Queue)");
    string result = Console.ReadLine();
    if (result.ToUpper().Equals("S")) { result = "S"; } else { result = "Q"; }

    return result;
}

static int numberOfTrials()
{
    Console.WriteLine("Please a number of trials, between 1 and 1,000,000, to test (Default = 1,000,000)");
    string result = Console.ReadLine();
    int choice = 0;

    try
    {
        choice = int.Parse(result);
    }
    catch (ArgumentException) { }
    catch (FormatException) { }

    if (!(choice > 0 && choice <= 1000000))
    {
        Console.WriteLine(choice + " is invalid. Defaulting to 1,000,000)");
        choice = defaultNoOfTrials;
    }
    return choice;
}
static void Main(String[] args)
{
    try
    {
        // Add the event handler for handling UI thread exceptions to the event.
// Application.ThreadException += new
//    ThreadExceptionEventHandler(ErrorHandlerForm.Form1_UIThreadException);
// Set the unhandled exception mode to force all Windows Forms
// errors to go through our handler.
// Application.SetUnhandledExceptionMode(UnhandledExceptionMode.CatchException);

// Add the event handler for handling non-UI thread exceptions to the event.
AppDomain.CurrentDomain.UnhandledException +=
    new UnhandledExceptionEventHandler(CurrentDomain_UnhandledException);

int choice = 0;

do
{
    ProducerTestRunner producer = null;
    producer = new ProducerTestRunner();

    choice = Menu();

    if (choice > 0) {
        String channelType = queueOrStack();
        int numberOftrials = numberOfTrials();
        producer.Init(channelType, numberOftrials);
    }

    switch (choice)
    {
        case 1:
            Console.WriteLine("Press ENTER to complete the Menu test for the Producers");
            Console.ReadLine();

            // Console.WriteLine("Press ENTER to execute Test _01_testEmptyWhenConstructed();");
            // Console.ReadLine();
            // producer._01_testEmptyWhenConstructed();
            // Console.WriteLine("Press ENTER to EXIT Test _01_testEmptyWhenConstructed();");
            // Console.ReadLine();
            break;

        //case 2:
        //    Console.WriteLine("Press ENTER to execute Test _02_testIsFullAfterPutsAndEmptyAfterTakes();");
        //    Console.ReadLine();
        //    producer._02_testIsFullAfterPutsAndEmptyAfterTakes();
        //    Console.WriteLine("Press ENTER to EXIT Test _02_testIsFullAfterPutsAndEmptyAfterTakes();");
        //    Console.ReadLine();
        //    break;
// case 3:

case 4:
    TEST = false;
    Console.WriteLine("Press ENTER to execute the producer component of Test _04_testTakeIsUnblockedWhenElementAdded();");
    Console.ReadLine();
    producer._04_testTakeIsUnblockedWhenElementAdded();
    Console.WriteLine("Press ENTER to EXIT the producer component of Test _04_testTakeIsUnblockedWhenElementAdded();");
    Console.ReadLine();
    break;

case 5:
    TEST = true;
    Console.WriteLine("Press ENTER to execute the producer component of Test _05_testPutTakeInt();");
    Console.ReadLine();
    producer._05_testPutTakeInt();
    Console.WriteLine("Press ENTER to EXIT the producer component of Test _05_testPutTakeInt();");
    Console.ReadLine();
    break;

case 6:
    TEST = true;
    Console.WriteLine("Press ENTER to execute the producer component of Test _05_testPutTakeLong();");
    Console.ReadLine();
    producer._05_testPutTakeLong();
    Console.WriteLine("Press ENTER to EXIT the producer component of Test _05_testPutTakeLong();");
    Console.ReadLine();
    break;

case 7:
    TEST = false;
    Console.WriteLine("Press ENTER to execute the producer component of Test_07_testPutTakeString");
    Console.ReadLine();
    producer._07_testPutTakeString();
    Console.WriteLine("Press ENTER to EXIT the producer component of Test _07_testPutTakeString");
    Console.ReadLine();
    break;

case 8:
    TEST = false;
    Console.WriteLine("Press ENTER to execute the producer component of Test _08_testPutTake_fixed");
    Console.ReadLine();
    producer._08_testPutTake_fixed();
    Console.WriteLine("Press ENTER to EXIT the producer component of Test _08_testPutTake_fixed");
    Console.ReadLine();
    break;

case 10:
    TEST = true;
    Console.WriteLine("Press ENTER to execute the producer component of Test Group 00 - Integers");
    Console.ReadLine();
    producer.test_group_00();
    Console.WriteLine("Press ENTER to EXIT the producer component of Test Group 00 - Integers");
    Console.ReadLine();
    break;

case 11:
    TEST = true;
    Console.WriteLine("Press ENTER to execute the producer component of Test Group 01 - Longs");
    Console.ReadLine();
    producer.test_group_01();
    Console.WriteLine("Press ENTER to EXIT the producer component of Test Group 01 = Longs");
    Console.ReadLine();
    break;

case 12:
    TEST = true;
    Console.WriteLine("Press ENTER to execute the producer component of Test Group 02 - Strings");
    Console.ReadLine();
    producer.test_group_02();
    Console.WriteLine("Press ENTER to EXIT the producer component of Test Group 02 - Strings");
    Console.ReadLine();
    break;

case 13:
    TEST = true;
    Console.WriteLine("Press ENTER to execute the producer component of Test Group 03 - Structs");
    Console.ReadLine();
    producer.test_group_03();
    Console.WriteLine("Press ENTER to EXIT the producer component of Test Group 03 - Structs");
    Console.ReadLine();
    break;

case 14:
    TEST = true;
    Console.WriteLine("Press ENTER to execute the consumer component of Test Groups 00, 01, 02 and 03");
    Console.ReadLine();
    producer.test_group_00();
    producer.test_group_01();
```

```
                    producer.test_group_02();
                    producer.test_group_03();
                    Console.WriteLine("Press ENTER to EXIT the consumer component of Test Groups 00, 01, 02 and 03");
                    Console.ReadLine();
                    break;

                default:
                    Console.WriteLine("No valid test selection was made. Shutting down...");
                    break;
            }
        }
        while (choice > 0);
    }
    catch (Exception ex)
    {
        // Ignore ex - We should have displayed it in the individual TEST that failed
        Console.Write(ex.Message);
    }
}

public void Init(string channelType, int numberOfTrials)
{
    // Configure all tests to be run on a queue or a stack type channel

    if (channelType.ToUpper() == "S")
    {
        initTestDataStructureType = DataStructureType.Stack;
    }
    else
    {
        initTestDataStructureType = DataStructureType.Queue;
    }

    initNoOfTrials = numberOfTrials;

    // These values are not used in the Consumers
    maxIntRandomSeed = 1000;
    maxLongRandomSeed = 1000000;
}

private static void CurrentDomain_UnhandledException(object sender, UnhandledExceptionEventArgs e)
{
    // Set up uncaught exception handler in case some dodgy code throws a RunTimeException
    // This won't work if the exception is passed to some even more dodgy 3rd psrty code that swallows
    // the exception. Does work in the case of dodgy 3rd party rogue code like ActiveMQ which kindly throws
```

```
        // some kind of runtime exception if you don't have a 'geronimo' jar in your classpath when you try to
        // instantiate a connectionFactory or ActiveMQConnectionFactory
        // Java version looks like this - ASExceptionHandler UEH = new ASExceptionHandler();
        //                              Thread.setDefaultUncaughtExceptionHandler(UEH);
        // Java also has per-thread scheduler handlers set up using the same class
        Console.Write(e.ExceptionObject.ToString());
    }

    public void _01_testEmptyWhenConstructed()
    {
        TEST = false;

        MMChannel mmMain = null;

        try
        {
            int initialCount = 0;
            string QueueName = "_01_testEmptyWhenConstructed";
            int viewSize = 1000;
            int fileSize = 1000000;
            int capacity = 500;

            // mmq = new BlockingCollection<string>(new ConcurrentQueue<string>(), capacity);
            // mmq = new MMQueueArrayType(QueueName, new MMFileValueType(QueueName, fileSize, viewSize, capacity));
            mmMain = MMChannel.GetInstance(QueueName, fileSize, viewSize, capacity, DEBUG, TEST, initTestDataStructureType);

            ControlData controlData = mmMain.MMFControlData;


            Console.WriteLine("Result of _01_testEmptyWhenConstructed() = (Count {1} == initialCount {2}) = {0}",
                controlData.totalItemsEnqueued == initialCount, controlData.totalItemsEnqueued, initialCount);
        }
        catch (Exception unexpected)
        {
            Console.Write(unexpected);
            throw;
        }
        finally
        {
            Thread.Sleep(1000);
            mmMain.Report();
            mmMain.Dispose();
        }
    }
```

```
public void _02_testIsFullAfterPutsAndEmptyAfterTakes()
{
    TEST = false;

    MMChannel mmMain = null;
    try
    {
        int initialCount = 0;
        string QueueName = "_02_testIsFullAfterPutsAndEmptyAfterTakes";
        int viewSize = 1000;
        int fileSize = 1000000;
        int capacity = 500;

        // mmq = new BlockingCollection<string>(new ConcurrentQueue<string>(), capacity);
        // mmq = new MMQueueArrayType(QueueName, new MMFileValueType(QueueName, fileSize, viewSize, capacity));
        mmMain = MMChannel.GetInstance(QueueName, fileSize, viewSize, capacity, DEBUG, TEST, initTestDataStructureType);

        // Fill the queue
        // for (int i = 0; i < capacity; i++) { mmq.Add(i.ToString()); }
        for (int i = 0; i < capacity; i++) { mmMain.Put((char)i); }

        // Verify that the queue is full
        ControlData controlData = mmMain.MMFControlData;

        Console.WriteLine("_02_testIsFullAfterPutsAndEmptyAfterTakes count = {0} capacity = {1}", controlData.totalItemsEnqueued, capacity);

        // Empty the queue
        for (int i = 0; i < capacity; i++) { mmMain.Take<char>(); }

        // Verify that the queue is empty
        controlData = mmMain.MMFControlData;

        Console.WriteLine("Result of _02_testIsFullAfterPutsAndEmptyAfterTakes() = (reservations {1} == initialCount {2}) = {0} after enqueuing and dequeueing {3} items",
            controlData.totalItemsEnqueued - controlData.totalItemsDequeued == initialCount, controlData.totalItemsEnqueued, initialCount, capacity);
    }
    catch (Exception unexpected)
    {
        Console.Write(unexpected);
        throw;
    }
    finally
    {
        Thread.Sleep(1000);
        mmMain.Report();
        mmMain.Dispose();
    }
}


// This is the default layout that the compiler would use anyway
[StructLayout(LayoutKind.Sequential, Pack = 1, CharSet = CharSet.Unicode)]
unsafe struct _04_MMData
{
    public int Value;
    public char Letter;
    public int NumbersLength;
    public fixed float Numbers[10];
    public int TextLength;
    public fixed char Text[100];
}
unsafe struct _04_args
{
    public MMChannel mQueue;
    public _04_MMData dData;
}
private void _04_EnqueueData(_04_args arg)
{
    try
    {
        // Local 'data' or its members cannot have their address taken and be used inside an anonymous
        // method or lambda expression - Error when trying to enqueue a struct
        arg.mQueue.Put(arg.dData);

        StringBuilder numbers = new StringBuilder();

        for (int i = 0; i < arg.dData.NumbersLength; i++)
        {
            unsafe
            {
                numbers.Append(arg.dData.Numbers[i] + ", ");
            }
        }

        char[] txt = new char[arg.dData.TextLength];
        for (int i = 0; i < arg.dData.TextLength; i++)
        {
            unsafe { txt[i] = arg.dData.Text[i]; }
        }
        string text = new String(txt);
```

```
            Console.WriteLine("_04_testTakeIsUnblockedWhenElementAdded() - data items enqueued \n'{0}', \n'{1}', \n'{2}', \n'{3}'",
                arg.dData.Value, arg.dData.Letter, numbers, text);
        }
        catch (Exception unexpected)
        {
            Console.Write(unexpected);
            throw;
        }
    }


    public void _04_testTakeIsUnblockedWhenElementAdded()
    {
        int LOCKUP_DETECT_TIMEOUT_MILLIS = 1000;
        int initialCount = 0;
        int viewSize = 1000;
        int fileSize = 1000000;
        int capacity = 500;

        TEST = false;

        MMChannel mmMain = null;

        _04_MMData data; // A struct containing data to be enqueued and dequeued
        _04_args arg;    // A struct containing the data struct and the Memory Mapped File View Accessor to be passed as a parameter
        // to a parameterized threadstart

        string QueueName = "_04_testTakeIsUnblockedWhenElementAdded";
        mmMain = MMChannel.GetInstance(QueueName, fileSize, viewSize, capacity, DEBUG, TEST, initTestDataStructureType);

        // Populate the struct of data to be enqueued and dequeued
        data.Value = 1;
        data.Letter = 'A';
        data.NumbersLength = 5;
        for (int i = 0; i < data.NumbersLength; i++) { unsafe { data.Numbers[i] = i; } }

        string msg = "EUR/GBP USD/JPY AUD/USD";
        char[] txt = msg.ToCharArray();
        // Store the length of the array for dequeueing later
        data.TextLength = txt.Length;
        // Copy the data to unmanaged memory char by char
        for (int i = 0; i < data.TextLength; i++) { unsafe { data.Text[i] = txt[i]; } }

        // Assign the data and the view accessor to the struct that we will use for the parameterized threadstart
        arg.mQueue = mmMain;
```

```
arg.dData = data;

// Create the Producer threads with lamda expression that refers to a method rather than anonymous
Thread Producer_1 = new Thread(() => _04_EnqueueData(arg));
Thread Producer_2 = new Thread(() => _04_EnqueueData(arg));

// perform the test from the main thread
try
{
    ControlData controlData = mmMain.MMFControlData;

    Console.WriteLine("Verify that the queue is empty");
    Console.WriteLine("_04_testTakeIsUnblockedWhenElementAdded() Queue is empty? = (Count {1} == initialCount {2}) = {0}\n",
        controlData.totalItemsEnqueued - controlData.totalItemsDequeued == initialCount,
        controlData.totalItemsEnqueued - controlData.totalItemsDequeued, initialCount);

    // Start a thread to enqueue an element
    Producer_1.Start();

    // Wait for a period for the thread to die
    Thread.Sleep(LOCKUP_DETECT_TIMEOUT_MILLIS);
    Producer_1.Join(LOCKUP_DETECT_TIMEOUT_MILLIS);
    // Verify the thread has died
    Console.WriteLine("_04_testTakeIsUnblockedWhenElementAdded() = Join main thread to Producer thread returned before {0} ms and Isalive = {1}",
        LOCKUP_DETECT_TIMEOUT_MILLIS, Producer_1.IsAlive);

    Console.WriteLine("Verify the queue now contains one element");
    controlData = mmMain.MMFControlData;

    Console.WriteLine("_04_testTakeIsUnblockedWhenElementAdded() = (Count {1} == initialCount {2}) = {0} after enqueuing {3} items\n",
        controlData.totalItemsEnqueued - controlData.totalItemsDequeued == initialCount,
        controlData.totalItemsEnqueued - controlData.totalItemsDequeued, initialCount, controlData.totalItemsEnqueued);

    Console.WriteLine("Run the consumer then press ENTER to continue with the producer test runner");
    Console.ReadLine();


    // Start a thread to enqueue an element
    Producer_2.Start();

    // Wait for a period for the thread to die
    Thread.Sleep(LOCKUP_DETECT_TIMEOUT_MILLIS);
    Producer_2.Join(LOCKUP_DETECT_TIMEOUT_MILLIS);
    Console.WriteLine("Verify the Producer thread has died");
    Console.WriteLine("_04_testTakeIsUnblockedWhenElementAdded() = Joining the main thread to the Producer thread returned before {0} ms and Isalive = {1}\n",
                LOCKUP_DETECT_TIMEOUT_MILLIS, Producer_2.IsAlive);

            Console.WriteLine("Press ENTER to FINISH");
            Console.ReadLine();

        }
        catch (Exception unexpected)
        {
            Console.Write(unexpected);
            throw;
        }
        finally
        {
            Thread.Sleep(1000);
            mmMain.Report();
            mmMain.Dispose();
        }
    }


    private static int xOrShift(int y)
    {
        // Java Concurrency in Practice page 253. Listing 12.4 Medium quality RNG suitable for testing
        // Java Version
        // y ^= (y << 6);   ^= means ' y = y XORshift 6 - does not exist in C#
        // y ^= (y >>> 21); unsigned right shift operator - does not exist in C#
        // y ^= (y << 7);

        y = y ^ (y << 6);
        y = y ^ (int)((uint)y >> 21); // Have to cast to uint to simulate unsigned right shift operator
        y = y ^ (y << 7);

        return y;
    }

    public void _05_testPutTakeInt()
    {
        // Test that the queue performs correctly under unpredictable concurrent access by using multiple threads to
        // to to perform Put and Take operations over a period of time and that nothing wnet wrong

        TEST = true;

        Console.WriteLine("\nStart of Test Run No. {0} in Test Suite No. {1}\n", ++initTestRunNumber, initTestSuiteNumber);

        MMChannel mmMain = null;
```

```
try
{
    int capacity = 10, fileSize = 1000000, viewSize = 1000;
    string QueueName = "_05_testPutTakeInt";

    // If only performing a small number of trials then GC could impact the timing tests so try and request it beforehand
    // In the case of a small number of trials, hopefully GC won't be required again before the end of the test
    System.GC.Collect();

    // INFO Cannot use the Property (get/set) with an Interlocked -
    // Store the value of the computed checksums here using Interlocked to ensure atomicty
    long putSum = 0;
    // Start and end times of the test run
    long timerStartTime = 0, timerEndTime = 0;

    // test parameters
    int nPairs = 10, nTrials = initNoOfTrials;

    Random rand = new Random();

    // Create the MMChannel which will instantiate the memory mapped files, mutexes, semaphores etc ...
    mmMain = MMChannel.GetInstance(QueueName, fileSize, viewSize, capacity, DEBUG, TEST, initTestDataStructureType);

    #region Barrier and Barrier Action declaration
    // The barrier will wait for the test runner thread plus a producer and consumer each for the number of pairs
    // Waits for them all to be ready at the start line and again at the finish
    Barrier _barrier = new Barrier(nPairs + 1,
        actionDelegate =>
        {
            // Check to see if the start time variable has been assigned or still = zero
            // If false then this is the first execution of the barrier action (at the start). Otherwise it is the
            // second execution 9at the finish)
            const long zeroFalse_1 = 0; // Not passed by ref so no need to be assignable
            bool started = Interlocked.Equals(timerStartTime, zeroFalse_1);
            started = !started;

            // Store the start time or the end time depending on which execution this is
            long t = DateTime.Now.Ticks;
            if (!started)
            {
                Interlocked.Exchange(ref timerStartTime, t);
            }
            else
            {
```

```
            Interlocked.Exchange(ref timerEndTime, t);
        }
    }
);
#endregion Barrier and Barrier Action declaration

// create pairs of threads to put and take items to/from the queue
// Including the test runner thread the barriers will wait for nPairs * 2 + 1 ther
for (int i = 0; i < nPairs; i++)
{
    #region Producer Lamda declaration

    new Thread(
        new ThreadStart(
        // Old way - replace lamda expression '() =>' with 'delegate'
        () =>
        {
            try
            {
                // B.Goetz's Java version used "this.hashCode()" and this method was in a Runnable inner class
                // Creating an inner (nested) class inside a method may be possible in C# but seems to me all we
                // need is an Object so we can get a hash code
                // http://msdn.microsoft.com/en-us/library/system.datetime.ticks.aspx
                // TimeSpan elapsedSpan = new TimeSpan(elapsedTicks);
                // Java = seed = (this.hashCode() ^ (int) System.nanoTime());
                DateTime centuryBegin = new DateTime(2001, 1, 1);
                DateTime currentDate = DateTime.Now;

                // Original RNG
                int elapsedTicks = (int)(currentDate.Ticks - centuryBegin.Ticks);

                #region  WARNING - THIS TEST IS FOR INTEGERS ONLY!
                // IT IS THE PROGRAMMER'S RESPONSIBILITY TO ENSURE THAT THE COMPUTED
                // RESULT DOES NOT EXCEED THE MAX SIZE OF AN INTEGER
                // The result depends on the product of number of trials and the max size of the random number generated
                // In this case nTrials and maxIntRandomSeed respectively.
                // Choose values that will not exceed the max size or you will get corrupted results
                // An example is the Put sum is positive and the Take sum is negative because the result
                // overflowed the integer size and wrote a 1 to the sign bit or the Put sum is very large but
                // the Take sum is orders of magnitude smaller because it overflowed but wrote a zero to the sign bit
                #endregion WARNING - THIS TEST IS FOR INTEGERS ONLY!

                int result = 0;

                // Console.WriteLine("producer wait at the start barrier");
    // Wait at the barrier (start line) until all test threads have been created and are ready to go
    _barrier.SignalAndWait();

    // The Producer's sum should equal the Consumenr's sum at the end of the test
    for (int j = nTrials; j > 0; --j)
    {
        // enqueue the random value
        // If the RNG is sound then this proves that the data enqueued was dequeued
        // mmq.Add(Convert.ToString(seed));
        // Original RNG
        int r = rand.Next(maxIntRandomSeed);

        // mmq.Add(Convert.ToString(r));

        // Original RNG
        mmMain.Put((long)r);
        // New RNG
        // mmMain.Put((long) elapsedTicks );

        // Original RNG
        result += r;
        // New RNG
        // result += elapsedTicks;

        // elapsedTicks = xOrShift(elapsedTicks);
    }
    // Atomically store the computed checksum
    // Comment out for Test 01 as we have already incremented it
    Interlocked.Add(ref putSum, result);

    // Wait at the barrier (finish line) until all test threads have been finished
    _barrier.SignalAndWait();

}
catch (Exception unexpected)
{
    Console.Write(unexpected);
    throw;
}
finally
{
    // No need to dispose of these thread local queues as they will be garbase collected
    // when they go out scope though could consider closing them without disposing of the
    // IPC artefacts. These must remain in existance until all the queue has been drained by the consumers
```

```
                    // mmq.Close();
                }
            }
        )).Start();

        #endregion Producer Lamda declaration

    }

    _barrier.SignalAndWait();   // Wait for all the threads to be ready
    _barrier.SignalAndWait();   // Wait for all the threads to finish

    // calculate the number of ticks elapsed during the test run
    long elapsedTime = Interlocked.Read(ref timerEndTime) - Interlocked.Read(ref timerStartTime);
    Console.WriteLine("Intermediate Result of _05_testPutTakeInt() - elapsed time = {0} timer ticks for {1} producer/consumer pairs and {2} Messages",
        elapsedTime, nPairs, nTrials);

    // Calculate the number of ticks per item enqueued and dequeued - the throughput of the queue
    // A single tick represents one hundred nanoseconds or one ten-millionth of a second.
    // There are 10,000 ticks in a millisecond.
    long ticksPerItem = elapsedTime / (nPairs * (long)nTrials);
    TimeSpan elapsedSpan = new TimeSpan(ticksPerItem);
    double milliSeconds = elapsedSpan.TotalMilliseconds;
    long nanoSeconds = ticksPerItem * 100;
    long throughput = 1000000000 / nanoSeconds;

    // Compares the checksum values computed to determine if the data enqueued was exactly the data dequeued
    Console.WriteLine("_05_testPutTakeInt() = (data enqueued = {0} after {1} trials each by {2} pairs of producers/consumers",
        Interlocked.Read(ref putSum), nTrials, nPairs);

    Console.WriteLine("_05_testPutTakeInt() = (Average latency = {0} timer ticks <= Threshold value {1}) = {2}",
        Interlocked.Read(ref ticksPerItem),
        AVERAGE_THROUGHPUT_THRESHOLD_TICKS,
        Interlocked.Read(ref ticksPerItem) <= AVERAGE_THROUGHPUT_THRESHOLD_TICKS);

    Console.WriteLine("_05_testPutTake Throughput = {0} messages per second ", throughput);

    Console.WriteLine("_05_testPutTake {0} timer ticks = {1} nanoseconds or {2} milliseconds",
        ticksPerItem, nanoSeconds, milliSeconds);

    Console.WriteLine("\nEnd of Test Run No. {0} in Test Suite No. {1}\n", initTestRunNumber, initTestSuiteNumber);
}
catch (Exception unexpected)
{
    Console.Write(unexpected);
```

```
        throw;
    }
    finally
    {
        // Temporarily delay disposing the queue and its IPC artefacts to allow the consumers to finish draining the queue
        // This will be fixed by waiting in an interrupible loop for the mutex inside the queue and checking if shutdown
        Thread.Sleep(1000);
        mmMain.Report();
        mmMain.Dispose();
    }
}


public void _05_testPutTakeLong()
{
    // Test that the queue performs correctly under unpredictable concurrent access by using multiple threads to
    // to to perform Put and Take operations over a period of time and that nothing wnet wrong

    TEST = true;

    Console.WriteLine("\nStart of Test Run No. {0} in Test Suite No. {1}\n", ++initTestRunNumber, initTestSuiteNumber);

    MMChannel mmMain = null;

    try
    {
        int capacity = 10, fileSize = 1000000, viewSize = 1000;
        string QueueName = "_05_testPutTakeLong";

        // If only performing a small number of trials then GC could impact the timing tests so try and request it beforehand
        // In the case of a small number of trials, hopefully GC won't be required again before the end of the test
        System.GC.Collect();

        // INFO Cannot use the Property (get/set) with an Interlocked -
        // Store the value of the computed checksums here using Interlocked to ensure atomicty
        long putSum = 0;
        // Start and end times of the test run
        long timerStartTime = 0, timerEndTime = 0;

        // test parameters
        int nPairs = 10, nTrials = initNoOfTrials;

        Random rand = new Random();

        // Create the MMChannel which will instantiate the memory mapped files, mutexes, semaphores etc ...
```

```
mmMain = MMChannel.GetInstance(QueueName, fileSize, viewSize, capacity, DEBUG, TEST, initTestDataStructureType);

#region Barrier and Barrier Action declaration
// The barrier will wait for the test runner thread plus a producer and consumer each for the number of pairs
// Waits for them all to be ready at the start line and again at the finish
Barrier _barrier = new Barrier(nPairs + 1,
    actionDelegate =>
    {
        // Check to see if the start time variable has been assigned or still = zero
        // If false then this is the first execution of the barrier action (at the start). Otherwise it is the
        // second execution 9at the finish)
        const long zeroFalse_1 = 0; // Not passed by ref so no need to be assignable
        bool started = Interlocked.Equals(timerStartTime, zeroFalse_1);
        started = !started;

        // Store the start time or the end time depending on which execution this is
        long t = DateTime.Now.Ticks;
        if (!started)
        {
            Interlocked.Exchange(ref timerStartTime, t);
        }
        else
        {
            Interlocked.Exchange(ref timerEndTime, t);
        }
    }
);
#endregion Barrier and Barrier Action declaration

// create pairs of threads to put and take items to/from the queue
// Including the test runner thread the barriers will wait for nPairs * 2 + 1 ther
for (int i = 0; i < nPairs; i++)
{
    #region Producer Lamda declaration

    new Thread(
        new ThreadStart(
        // Old way - replace lamda expression '() =>' with 'delegate'
        () =>
        {
            try
            {
                // B.Goetz's Java version used "this.hashCode()" and this method was in a Runnable inner class
                // Creating an inner (nested) class inside a method may be possible in C# but seems to me all we
                // need is an Object so we can get a hash code
    // http://msdn.microsoft.com/en-us/library/system.datetime.ticks.aspx
    // TimeSpan elapsedSpan = new TimeSpan(elapsedTicks);
    // Java = seed = (this.hashCode() ^ (int) System.nanoTime());
    DateTime centuryBegin = new DateTime(2001, 1, 1);
    DateTime currentDate = DateTime.Now;

    int elapsedTicks = (int)(currentDate.Ticks - centuryBegin.Ticks);

    long result = 0;

    // Console.WriteLine("producer wait at the start barrier");

    // Wait at the barrier (start line) until all test threads have been created and are ready to go
    _barrier.SignalAndWait();

    // The Producer's sum should equal the Consumenr's sum at the end of the test
    for (int j = nTrials; j > 0; --j)
    {
        // enqueue the random value
        // If the RNG is sound then this proves that the data enqueued was dequeued
        // mmq.Add(Convert.ToString(seed));
        long r = rand.Next(maxLongRandomSeed);
        // mmq.Add(Convert.ToString(r));
        mmMain.Put((long)r);
        result += r;
    }
    // Atomically store the computed checksum
    // Comment out for Test 01 as we have already incremented it
    Interlocked.Add(ref putSum, result);

    // Wait at the barrier (finish line) until all test threads have been finished
    _barrier.SignalAndWait();

}
catch (Exception unexpected)
{
    Console.Write(unexpected);
    throw;
}
finally
{
    // No need to dispose of these thread local queues as they will be garbase collected
    // when they go out scope though could consider closing them without disposing of the
    // IPC artefacts. These must remain in existance until all the queue has been drained by the consumers
    // mmq.Close();
```

```
                }
            }
        )).Start();

        #endregion Producer Lamda declaration

    }

    _barrier.SignalAndWait();   // Wait for all the threads to be ready
    _barrier.SignalAndWait();   // Wait for all the threads to finish

    // calculate the number of ticks elapsed during the test run
    long elapsedTime = Interlocked.Read(ref timerEndTime) - Interlocked.Read(ref timerStartTime);
    Console.WriteLine("Intermediate Result of _05_testPutTakeLong() - elapsed time = {0} timer ticks for {1} producer/consumer pairs and {2} Messages",
        elapsedTime, nPairs, nTrials);

    // Calculate the number of ticks per item enqueued and dequeued - the throughput of the queue
    // A single tick represents one hundred nanoseconds or one ten-millionth of a second.
    // There are 10,000 ticks in a millisecond.
    long ticksPerItem = elapsedTime / (nPairs * (long)nTrials);
    TimeSpan elapsedSpan = new TimeSpan(ticksPerItem);
    double milliSeconds = elapsedSpan.TotalMilliseconds;
    long nanoSeconds = ticksPerItem * 100;
    long throughput = 1000000000 / nanoSeconds;

    // Compares the checksum values computed to determine if the data enqueued was exactly the data dequeued
    Console.WriteLine("_05_testPutTakeLong() = (data enqueued = {0} after {1} trials each by {2} pairs of producers/consumers",
        Interlocked.Read(ref putSum), nTrials, nPairs);

    Console.WriteLine("_05_testPutTakeLong() = (Average latency = {0} timer ticks <= Threshold value {1}) = {2}",
        Interlocked.Read(ref ticksPerItem),
        AVERAGE_THROUGHPUT_THRESHOLD_TICKS,
        Interlocked.Read(ref ticksPerItem) <= AVERAGE_THROUGHPUT_THRESHOLD_TICKS);

    Console.WriteLine("_05_testPutTakeLong Throughput = {0} messages per second ", throughput);

    Console.WriteLine("_05_testPutTakeLong {0} timer ticks = {1} nanoseconds or {2} milliseconds",
        ticksPerItem, nanoSeconds, milliSeconds);

    Console.WriteLine("\nEnd of Test Run No. {0} in Test Suite No. {1}\n", initTestRunNumber, initTestSuiteNumber);
}
catch (Exception unexpected)
{
    Console.Write(unexpected);
    throw;
    }
    finally
    {
        // Temporarily delay disposing the queue and its IPC artefacts to allow the consumers to finish draining the queue
        // This will be fixed by waiting in an interrupible loop for the mutex inside the queue and checking if shutdown
        Thread.Sleep(1000);
        mmMain.Report();
        mmMain.Dispose();
    }
}


public void _07_testPutTakeString()
{
    // Test that the queue performs correctly under unpredictable concurrent access by using multiple threads to
    // to to perform Put and Take operations over a period of time and that nothing went wrong

    TEST = false;
    MMChannel mmMain = null;

    Console.WriteLine("\nStart of Test Run No. {0} in Test Suite No. {1}\n", ++initTestRunNumber, initTestSuiteNumber);

    String applicationInstance = UTCutil.GetInstanceNameForProcessId(Process.GetCurrentProcess().Id);
    Dictionary<String, PerformanceCounter> map = UTCutil.ReadKeyMemoryAndHandlePerformanceCounters(applicationInstance);

    PerformanceCounter all_heaps_counter;
    map.TryGetValue(UTCutil.performanceCounter_bytes_in_all_heaps, out all_heaps_counter);
    String name = all_heaps_counter.CounterName.ToString();

    try
    {
        int capacity = 500, fileSize = 1000000, viewSize = 1000;
        string QueueName = "_07_testPutTakeString";

        // If only performing a small number of trials then GC could impact the timing tests so try and request it beforehand
        // In the case of a small number of trials, hopefully GC won't be required again before the end of the test
        System.GC.Collect();

        // INFO Cannot use the Property (get/set) with an Interlocked -
        // Store the value of the computed checksums here using Interlocked to ensure atomicty
        long putSum = 0;
        // Start and end times of the test run
        long timerStartTime = 0, timerEndTime = 0;

        // test parameters
```

```
int nPairs = 10, nTrials = initNoOfTrials;

Random rand = new Random();

// Create the MMChannel which will instantiate the memory mapped files, mutexes, semaphores etc ...
mmMain = MMChannel.GetInstance(QueueName, fileSize, viewSize, capacity, DEBUG, TEST, initTestDataStructureType);

#region Barrier and Barrier Action declaration
// The barrier will wait for the test runner thread plus a producer and consumer each for the number of pairs
// Waits for them all to be ready at the start line and again at the finish
Barrier _barrier = new Barrier(nPairs + 1,
    actionDelegate =>
    {
        // Check to see if the start time variable has been assigned or still = zero
        // If false then this is the first execution of the barrier action (at the start). Otherwise it is the
        // second execution at the finish)
        const long zeroFalse_1 = 0; // Not passed by ref so no need to be assignable
        bool started = Interlocked.Equals(timerStartTime, zeroFalse_1);
        started = !started;

        // Store the start time or the end time depending on which execution this is
        long t = DateTime.Now.Ticks;
        if (!started)
        {
            Interlocked.Exchange(ref timerStartTime, t);
        }
        else
        {
            Interlocked.Exchange(ref timerEndTime, t);
        }
    }
);
#endregion Barrier and Barrier Action declaration

// create pairs of threads to put and take items to/from the queue
// Including the test runner thread the barriers will wait for nPairs * 2 + 1 there
for (int i = 0; i < nPairs; i++)
{
    #region Producer Lamda declaration

    new Thread(
        new ThreadStart(
        () =>
        {
            try
```

```
{
    // B.Goetz's Java version used "this.hashCode()" and this method was in a Runnable inner class
    // Creating an inner (nested) class inside a method may be possible in C# but seems to me all we
    // need is an Object so we can get a hash code
    // http://msdn.microsoft.com/en-us/library/system.datetime.ticks.aspx
    // TimeSpan elapsedSpan = new TimeSpan(elapsedTicks);
    // Java = seed = (this.hashCode() ^ (int) System.nanoTime());
    DateTime centuryBegin = new DateTime(2001, 1, 1);
    DateTime currentDate = DateTime.Now;

    int elapsedTicks = (int)(currentDate.Ticks - centuryBegin.Ticks);
    // int seed = (computedHashCode ^ elapsedTicks);
    // int seed = (int)(new Object().GetHashCode() ^ elapsedTicks);
    // int seed = (int)(new Object().GetHashCode());
    long result = 0;

    // Wait at the barrier (start line) until all test threads have been created and are ready to go
    _barrier.SignalAndWait();

    // Put the data into the queue as Strings, generating a new random number each time
    // The consumer will convert back to integers and sum them
    // The Producer's sum should equal the Consumenr's sum at the end of the test
    for (int j = nTrials; j > 0; --j)
    {

        // Test 03 - enqueue the random value
        // If the RNG is sound then this proves that the data enqueued was dequeued
        // mmq.Add(Convert.ToString(seed));
        int r = rand.Next(maxLongRandomSeed);
        // byte[] encodedData = MMChannel.StringToByteArray(Convert.ToString(r));
        char[] encodedData = Convert.ToString(r).ToCharArray();
        mmMain.Put(encodedData);
        result += r;

        // re-compute the random number
        // seed = MMQueue<string>.xorShift(seed);
    }
    // Atomically store the computed checksum
    // Comment out for Test 01 as we have already incremented it
    Interlocked.Add(ref putSum, result);

    // Wait at the barrier (finish line) until all test threads have been finished
    _barrier.SignalAndWait();

}
```

```
            catch (Exception unexpected)
            {
                Console.Write(unexpected);
                throw;
            }
        }
    )).Start();


    #endregion Producer Lamda declaration
}

int THRESHOLD = 1000; long diff;

#region heap profiling testing notes
// I got this off Java Concurrency in Practice, chap 12 Testing Concurrent Programs Page 258
// It doesn't work as written though!
// Generally, the heap size after testing was fraction of the size before the test
// Obviously, processing these huge messages has triggered a GC during the test
// Even then you would expect this to result in a false positive where the two snapshot were similar even if
// your code was leaking memory so the most likely explanation seems to be that NUnit itself is creating objects
// which have not yet been reclaimed before the test starts
// Requesting a GC before the initial snapshot solves the problem but you have to accept that you cannot
// completely control managed memory allocation and at some point the GC will probably ignore your request
// and the test will fail
#endregion heap profiling testing notes

System.GC.Collect();
long heapSizeBeforeTest = Convert.ToInt64(UTCutil.GetCounterValue(all_heaps_counter));

_barrier.SignalAndWait();   // Wait for all the threads to be ready
_barrier.SignalAndWait();   // Wait for all the threads to finish

System.GC.Collect();
long heapSizeAfterTest = Convert.ToInt64(UTCutil.GetCounterValue(all_heaps_counter));
diff = Math.Abs(heapSizeBeforeTest - heapSizeAfterTest);

Console.WriteLine("Result of TestLeak() Heap size at end of run = {0}, Heap size at start of run = {1} Difference = {2}, Passed = {3}",
    heapSizeAfterTest, heapSizeBeforeTest, diff, diff <= THRESHOLD);

// calculate the number of ticks elapsed during the test run
long elapsedTime = Interlocked.Read(ref timerEndTime) - Interlocked.Read(ref timerStartTime);
Console.WriteLine("Intermediate Result of _07_testPutTakeString() - elapsed time = {0} timer ticks for {1} producer/consumer pairs and {2} Messages",
    elapsedTime, nPairs, nTrials);
```

```
        // Calculate the number of ticks per item enqueued and dequeued - the throughput of the queue
        // A single tick represents one hundred nanoseconds or one ten-millionth of a second.
        // There are 10,000 ticks in a millisecond.
        long ticksPerItem = elapsedTime / (nPairs * (long)nTrials);
        TimeSpan elapsedSpan = new TimeSpan(ticksPerItem);
        double milliSeconds = elapsedSpan.TotalMilliseconds;
        long nanoSeconds = ticksPerItem * 100;
        long throughput = 1000000000 / nanoSeconds;

        // Compares the checksum values computed to determine if the data enqueued was exactly the data dequeued
        Console.WriteLine("1st Result of _07_testPutTakeString() = (data enqueued = {0} after {1} trials each by {2} pairs of producers/consumers",
            Interlocked.Read(ref putSum), nTrials, nPairs);

        Console.WriteLine("2nd Result of _07_testPutTakeString() = (Average latency = {0} timer ticks <= Threshold value {1}) = {2}",
            Interlocked.Read(ref ticksPerItem),
            AVERAGE_THROUGHPUT_THRESHOLD_TICKS,
            Interlocked.Read(ref ticksPerItem) <= AVERAGE_THROUGHPUT_THRESHOLD_TICKS);

        Console.WriteLine("_07_testPutTakeString Throughput = {0} messages per second ", throughput);

        Console.WriteLine("_07_testPutTakeString n.b. {0} timer ticks = {1} nanoseconds or {2} milliseconds",
            ticksPerItem, nanoSeconds, milliSeconds);

        Console.WriteLine("\nEnd of Test Run No. {0} in Test Suite No. {1}\n", initTestRunNumber, initTestSuiteNumber);
    }
    catch (Exception unexpected)
    {
        Console.Write(unexpected);
        throw;
    }
    finally
    {
        // Temporarily delay disposing the queue and its IPC artefacts to allow the consumers to finish draining the queue
        // This will be fixed by waiting in an interrupible loop for the mutex inside the queue and checking if shutdown
        Thread.Sleep(1000);
        mmMain.Report();
        mmMain.Dispose();
    }
}



// This is the default layout that the compiler would use anyway
[StructLayout(LayoutKind.Sequential, Pack = 1, CharSet = CharSet.Unicode)]
unsafe struct _08_MMData
```

```
{
    public int TextLength;
    public fixed char Text[100];
}

public void _08_testPutTake_fixed()
{
    TEST = false;

    // Test that the queue performs correctly under unpredictable concurrent access by using multiple threads to
    // to to perform Put and Take operations over a period of time and that nothing wnet wrong

    MMChannel mmMain = null;

    Console.WriteLine("\nStart of Test Run No. {0} in Test Suite No. {1}\n", ++initTestRunNumber, initTestSuiteNumber);

    try
    {
        int capacity = 500, fileSize = 1000000, viewSize = 1000;
        string QueueName = "_08_testPutTake_fixed";

        // If only performing a small number of trials then GC could impact the timing tests so try and request it beforehand
        // In the case of a small number of trials, hopefully GC won't be required again before the end of the test
        System.GC.Collect();

        // INFO Cannot use the Property (get/set) with an Interlocked -
        // Store the value of the computed checksums here using Interlocked to ensure atomicty
        long putSum = 0;
        // Start and end times of the test run
        long timerStartTime = 0, timerEndTime = 0;

        // test parameters
        // Performance nPairs = 10, capacity = 10, nTrials = 1,000,000 = BlockingCollection = 60s, MMQueue = 254s
        int nPairs = 10, nTrials = initNoOfTrials;

        Random rand = new Random();

        // Create the MMChannel which will instantiate the memory mapped files, mutexes, semaphores etc ...
        mmMain = MMChannel.GetInstance(QueueName, fileSize, viewSize, capacity, DEBUG, TEST, initTestDataStructureType);

        #region Barrier and Barrier Action declaration
        // The barrier will wait for the test runner thread plus a producer and consumer each for the number of pairs
        // Waits for them all to be ready at the start line and again at the finish
        Barrier _barrier = new Barrier(nPairs + 1,
            actionDelegate =>
    {
        // Check to see if the start time variable has been assigned or still = zero
        // If false then this is the first execution of the barrier action (at the start). Otherwise it is the
        // second execution 9at the finish)
        const long zeroFalse_1 = 0; // Not passed by ref so no need to be assignable
        bool started = Interlocked.Equals(timerStartTime, zeroFalse_1);
        started = !started;

        // Store the start time or the end time depending on which execution this is
        long t = DateTime.Now.Ticks;
        if (!started)
        {
            Interlocked.Exchange(ref timerStartTime, t);
        }
        else
        {
            Interlocked.Exchange(ref timerEndTime, t);
        }
    }
);
#endregion Barrier and Barrier Action declaration

// create pairs of threads to put and take items to/from the queue
// Including the test runner thread the barriers will wait for nPairs * 2 + 1 ther
for (int i = 0; i < nPairs; i++)
{
    #region Producer Lamda declaration

    new Thread(
        new ThreadStart(
        // Old way - replace lamda expression '() =>' with 'delegate'
        () =>
        {
            try
            {
                DateTime centuryBegin = new DateTime(2001, 1, 1);
                DateTime currentDate = DateTime.Now;

                int elapsedTicks = (int)(currentDate.Ticks - centuryBegin.Ticks);
                long result = 0;

                // Wait at the barrier (start line) until all test threads have been created and are ready to go
                _barrier.SignalAndWait();

                // Put the data into the queue as Strings, generating a new random number each time
                // The consumer will convert back to integers and sum them
                // The Producer's sum should equal the Consumenr's sum at the end of the test
                for (int j = nTrials; j > 0; --j)
                {
                    // If the RNG is sound then this proves that the data enqueued was dequeued
                    int r = rand.Next(maxLongRandomSeed);

                    _08_MMData data;

                    // Test data string to enqueue and dequeue. Convert to a byte array. This array is a reference type so cannot be directly
                    // passed to the View Accessor
                    char[] encodedData = Convert.ToString(r).ToCharArray();
                    // Store the length of the array for dequeueing later
                    data.TextLength = encodedData.Length;
                    // Copy the data to unmanaged memory char by char
                    for (int k = 0; k < data.TextLength; k++) { unsafe { data.Text[k] = encodedData[k]; } }

                    mmMain.Put(data);
                    result += r;

                }
                // Atomically store the computed checksum
                // Comment out for Test 01 as we have already incremented it
                Interlocked.Add(ref putSum, result);

                // Wait at the barrier (finish line) until all test threads have been finished
                _barrier.SignalAndWait();

            }
            catch (Exception unexpected)
            {
                Console.Write(unexpected);
                throw;
            }
        }
    )).Start();


    // Start a thread to enqueue an element
    // Producer.Start();

    #endregion Producer Lamda declaration

}
```

```
    _barrier.SignalAndWait();   // Wait for all the threads to be ready
    _barrier.SignalAndWait();   // Wait for all the threads to finish

    // calculate the number of ticks elapsed during the test run
    long elapsedTime = Interlocked.Read(ref timerEndTime) - Interlocked.Read(ref timerStartTime);
    Console.WriteLine("Intermediate Result of _08_testPutTake_fixed() - elapsed time = {0} timer ticks for {1} producer/consumer pairs and {2} Messages",
        elapsedTime, nPairs, nTrials);

    // Calculate the number of ticks per item enqueued and dequeued - the throughput of the queue
    // A single tick represents one hundred nanoseconds or one ten-millionth of a second.
    // There are 10,000 ticks in a millisecond.
    long ticksPerItem = elapsedTime / (nPairs * (long)nTrials);
    TimeSpan elapsedSpan = new TimeSpan(ticksPerItem);
    double milliSeconds = elapsedSpan.TotalMilliseconds;
    long nanoSeconds = ticksPerItem * 100;
    long throughput = 1000000000 / nanoSeconds;

    // Compares the checksum values computed to determine if the data enqueued was exactly the data dequeued
    Console.WriteLine("1st Result of _08_testPutTake_fixed() = (data enqueued = {0} after {1} trials each by {2} pairs of producers/consumers",
        Interlocked.Read(ref putSum), nTrials, nPairs);

    Console.WriteLine("2nd Result of _08_testPutTake_fixed() = (Average latency = {0} timer ticks <= Threshold value {1}) = {2}",
        Interlocked.Read(ref ticksPerItem),
        AVERAGE_THROUGHPUT_THRESHOLD_TICKS,
        Interlocked.Read(ref ticksPerItem) <= AVERAGE_THROUGHPUT_THRESHOLD_TICKS);

    Console.WriteLine("_08_testPutTake_fixed Throughput = {0} messages per second ", throughput);

    Console.WriteLine("_08_testPutTake_fixed {0} timer ticks = {1} nanoseconds or {2} milliseconds",
        ticksPerItem, nanoSeconds, milliSeconds);

    Console.WriteLine("\nEnd of Test Run No. {0} in Test Suite No. {1}\n", initTestRunNumber, initTestSuiteNumber);

}
catch (Exception unexpected)
{
    Console.Write(unexpected);
    throw;
}
finally
{
    // Temporarily delay disposing the queue and its IPC artefacts to allow the consumers to finish draining the queue
    // This will be fixed by waiting in an interrupible loop for the mutex inside the queue and checking if shutdown
    Thread.Sleep(1000);
    mmMain.Report();
```

```
        mmMain.Dispose();
    }
}

#region TEST Groups

public void test_group_00()
{
    int numTestRuns = 100;

    Console.WriteLine("Start Test Suite No. {0} using Integers - {1} test runs\n", ++initTestSuiteNumber, numTestRuns);

    for (int i = 0; i < numTestRuns; i++)
    {
        _05_testPutTakeInt();
    }

    Console.WriteLine("End Test Suite No. {0}\n", initTestSuiteNumber);
}

public void test_group_01()
{
    int numTestRuns = 100;

    Console.WriteLine("Start Test Suite No. {0} using Long Integers - {1} test runs\n", ++initTestSuiteNumber, numTestRuns);

    for (int i = 0; i < numTestRuns; i++)
    {
        _05_testPutTakeLong();
    }

    Console.WriteLine("End Test Suite No. {0}\n", initTestSuiteNumber);
}

public void test_group_02()
{
    int numTestRuns = 100;

    Console.WriteLine("Start Test Suite No. {0} using Strings - {1} test runs\n", ++initTestSuiteNumber, numTestRuns);

    for (int i = 0; i < numTestRuns; i++)
    {
        _07_testPutTakeString();
    }
```

```
            Console.WriteLine("End Test Suite No. {0}\n", initTestSuiteNumber);
        }

        public void test_group_03()
        {
            int numTestRuns = 100;

            Console.WriteLine("Start Test Suite No. {0} using Structs inc. Fixed Arrays - {1} test runs\n", ++initTestSuiteNumber, numTestRuns);

            for (int i = 0; i < numTestRuns; i++)
            {
                _08_testPutTake_fixed();
            }

            Console.WriteLine("End Test Suite No. {0}\n", initTestSuiteNumber);
        }

        #endregion TEST groups
    }
}
```

Inter Process producer-Consumer Queue Test Harness

ShutdownTestRunner.cs

```
/****************************************************************************************************************************************************
 * The goal of the AlphaSystematics Project is create an open-source system for forward-testing systematic strategies with live market data and trade feeds.
 * It enables strategies developed in Excel to be connected to trading venues via industry standard FIX messaging.
 *
 * Copyright (C) 2009  Antonio Tapper. www.alphasystematics.org

 * This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

 * This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more details.
 * You should have received a copy of the GNU General Public License along with this program.  If not, see <http://www.gnu.org/licenses/>.
******************************************************************************************************************************************************/

using System;
using System.Collections.Generic;
using System.IO;
using System.IO.MemoryMappedFiles;
using System.Text;
using System.Threading;
using com.alphaSystematics.concurrency;
using System.Linq;
using System.Threading.Tasks;
using System.Collections.Concurrent;
using System.Diagnostics;
using System.Runtime.InteropServices;

namespace TestMMFile_Shutdown
{
    public class ShutdownTestRunner
    {

        // The hardest part of writing tests is that when they fail you don't know if it is the test or the application
        // thats broken unless you have confidence that the tests themselves have been tested thoroughly
        // in this case we are lucky in that we are trying to mimic the functionality of an existing library class but extend
        // it to use inter process.
        // We can drop in the library class here in order to test the test because we have confidence that the library class
        // works so if the tests fail when using library class then the tests are broken - For NUnit tests in a single process

        static DataStructureType initTestDataStructureType = default(DataStructureType);
        const long AVERAGE_THROUGHPUT_THRESHOLD_TICKS = 1000;
        int initNoOfTrials = 0;
        const int defaultNoOfTrials = 1000000;
        const bool DEBUG = true; static bool TEST = false;

        static int Menu()
        {
            string result = ""; int choice = 0; bool valid = false;

            while (!valid)
            {
                Console.Clear();
                Console.WriteLine("Memory Mapped Message Channel test suite (Shutdown). Please choose from the following options:\n");

                Console.WriteLine("1: Test menu for the Shutdown\n");
                Console.WriteLine("3: Shutdown the Channel. Currently only implemented for Test No. 8: \nTest Put and Take methods with struct data and equal numbers of producers and consumers\n");

                Console.WriteLine("Q: Quit\n");

                // get the 1st character of input and quit if it is "Q"
                result = Console.ReadLine();
                if (result.ToUpper().Equals("Q")) { result = "0"; }

                try
                {
                    choice = int.Parse(result);
                }
                catch (ArgumentException) { }
                catch (FormatException) { }

                switch (choice)
                {
                    case 0:
                        Console.WriteLine("Quitting test harness {0} please wait...", result);
                        valid = true;
                        break;

                    case 1:
                        Console.WriteLine("Press ENTER to test the Menu for Shutdown");
                        Console.ReadLine();
                        valid = true;
                        break;

                    case 3:
                        Console.WriteLine(" Shutting down Channel (Option {0}). Please wait...", result);
                        valid = true;
                        break;

                    default:
                        Console.WriteLine("Invalid selection {0}. Please select 1 or 3 or Quit.\n\n\n\n\n", result);
                        break;
                }
            }
            return choice;
        }

        static string queueOrStack()
        {
```

```
    Console.WriteLine("Please choose to test a Queue or a Stack (Default = Queue)");
    string result = Console.ReadLine();
    if (result.ToUpper().Equals("S"))
    {
        result = "S";
    }
    else {
        // if ( ! result.ToUpper().Equals("Q")) { Console.WriteLine(result + " is invalid. Defaulting to Queue)"); }
        result = "Q";
    }

    return result;
}

static int numberOfTrials()
{
    Console.WriteLine("Please a number of trials, between 1 and 1,000,000, to test (Default = 1,000,000)");
    string result = Console.ReadLine();
    int choice = 0;

    try
    {
        choice = int.Parse(result);
    }
    catch (ArgumentException) { }
    catch (FormatException) { }

    if (!(choice > 0 && choice < 1000000))
    {
        Console.WriteLine(choice + " is invalid. Defaulting to 1,000,000)");
        choice = defaultNoOfTrials;
    }
    return choice;
}

static void Main(String[] args)
{
    try
    {
        // Add the event handler for handling UI thread exceptions to the event.
        // Application.ThreadException += new
        //    ThreadExceptionEventHandler(ErrorHandlerForm.Form1_UIThreadException);
        // Set the unhandled exception mode to force all Windows Forms
        // errors to go through our handler.
        // Application.SetUnhandledExceptionMode(UnhandledExceptionMode.CatchException);
    // Add the event handler for handling non-UI thread exceptions to the event.
    AppDomain.CurrentDomain.UnhandledException +=
        new UnhandledExceptionEventHandler(CurrentDomain_UnhandledException);

    int choice = 0;

    do {
        ShutdownTestRunner shutdown = null;
        shutdown = new ShutdownTestRunner();

        choice = Menu();

        if (choice > 0) {
            String channelType = queueOrStack();
            int numberOftrials = numberOfTrials();
            shutdown.Init(channelType, numberOftrials);
        }

        switch (choice)
        {
            case 1:
                Console.WriteLine("Press ENTER to complete the Menu test for Shutdown");
            Console.ReadLine();
            break;

            case 3:
            TEST = true;
            shutdown._03_shutdown();
            Console.WriteLine("Press ENTER to EXIT the shutdown component");
            Console.ReadLine();
            break;

            default:
            Console.WriteLine("No valid test selection was made. Shutting down...");
            break;
        }
    }
    while (choice > 1);
}
catch (Exception ex)
{
    // Ignore ex - We should have displayed it in the individual TEST that failed
    Console.Write(ex);
}
```

```
}

private static void CurrentDomain_UnhandledException(object sender, UnhandledExceptionEventArgs e)
{
    // Set up uncaught exception handler in case some dodgy code throws a RunTimeException
    // This won't work if the exception is passed to some even more dodgy 3rd psrty code that swallows
    // the exception. Does work in the case of dodgy 3rd party rogue code like ActiveMQ which kindly throws
    // some kind of runtime exception if you don't have a 'geronimo' jar in your classpath when you try to
    // instantiate a connectionFactory or ActiveMQConnectionFactory
    // Java version looks like this - ASExceptionHandler UEH = new ASExceptionHandler();
    //                               Thread.setDefaultUncaughtExceptionHandler(UEH);
    // Java also has per-thread scheduler handlers set up using the same class
    Console.Write(e.ExceptionObject);
}

    public void Init(string channelType, int numberOfTrials)
    {
        // Configure all tests to be run on a queue or a stack type channel

        if (channelType.ToUpper() == "S")
        {
            initTestDataStructureType = DataStructureType.Stack;
        } else {
            initTestDataStructureType = DataStructureType.Queue;
        }

        initNoOfTrials = numberOfTrials;
        // These values are not used in the Consumers
        // maxIntRandomSeed = 1000;
        // maxLongRandomSeed = 1000000;
    }

    public void _03_shutdown()
    {
        int initialCount = 0;
        int viewSize = 1000;
        int fileSize = 1000000;
        int capacity = 500;

        TEST = false;
        MMChannel mmMain = null;

        string QueueName = "_08_testPutTake_fixed";

        // Create the MMChannel which will instantiate the memory mapped files, mutexes, semaphores etc ...
```

```
                mmMain = MMChannel.GetInstance(QueueName, fileSize, viewSize, capacity, DEBUG, TEST, initTestDataStructureType);

                // perform the test from the main thread
                try
                {
                    ControlData controlData = mmMain.MMFControlData;

                    // verify that the queue is empty
                    Console.WriteLine("_03_shutdown() Queue is empty? = (Count {1} == initialCount {2}) = {0}",
                          controlData.totalItemsEnqueued - controlData.totalItemsDequeued == initialCount,
                          controlData.totalItemsEnqueued - controlData.totalItemsDequeued, initialCount);

                    Console.WriteLine("Press ENTER to shutdown the Channel");
                    Console.ReadLine();
                    mmMain.shutdown();

                    Console.WriteLine("Press ENTER to FINISH");
                    Console.ReadLine();

                }
                catch (Exception unexpected)
                {
                    Console.Write(unexpected);
                    throw;
                }
                finally
                {
                    Console.WriteLine("\n");
                    mmMain.Report();
                    mmMain.Dispose();
                }
            }

    }
}
```