# Fifo Stream

```
//////////////////////////////////////////////////////////////////////////////
//
// This File is Part of the CallButler Open Source PBX (http://www.codeplex.com/callbutler
//
// Copyright (c) 2005-2008, Jim Heising
// All rights reserved.
//
// Redistribution and use in source and binary forms, with or without modification,
// are permitted provided that the following conditions are met:
//
// * Redistributions of source code must retain the above copyright notice,
// this list of conditions and the following disclaimer.
//
// * Redistributions in binary form must reproduce the above copyright notice,
// this list of conditions and the following disclaimer in the documentation and/or
// other materials provided with the distribution.
//
// * Neither the name of Jim Heising nor the names of its contributors may be
// used to endorse or promote products derived from this software without specific prior
// written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND
// ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
```

```
LIMITED TO, THE IMPLIED
// WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED.
// IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT,
// INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT
// NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR
// PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
ANY THEORY OF LIABILITY,
// WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE)
// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
ADVISED OF THE
// POSSIBILITY OF SUCH DAMAGE.
//
////////////////////////////////////////////////////////////////
////////////////////////////////

using System;
using System.IO;
using System.Collections;

namespace WOSI.Utilities
{
public class FifoStream : Stream
{
private const int BlockSize = 65536;
private const int MaxBlocksInCache = (3 * 1024 * 1024) /
BlockSize;

private int m_Size;
private int m_RPos;
private int m_WPos;
private Stack m_UsedBlocks = new Stack();
private ArrayList m_Blocks = new ArrayList();
```

```
private byte[] AllocBlock()
{
byte[] Result = null;
Result = m_UsedBlocks.Count > 0 ? (byte[])m_UsedBlocks.Pop() :
new byte[BlockSize];
return Result;
}
private void FreeBlock(byte[] block)
{
if (m_UsedBlocks.Count < MaxBlocksInCache)
m_UsedBlocks.Push(block); } private byte[] GetWBlock() {
byte[] Result = null; if (m_WPos < BlockSize && m_Blocks.Count
> 0)
Result = (byte[])m_Blocks[m_Blocks.Count – 1];
else
{
Result = AllocBlock();
m_Blocks.Add(Result);
m_WPos = 0;
}
return Result;
}

// Stream members
public override bool CanRead
{
get { return true; }
}
public override bool CanSeek
{
get { return false; }
}
public override bool CanWrite
{
get { return true; }
}
public override long Length
```

```
{
get
{
lock (this)
return m_Size;
}
}
public override long Position
{
get { throw new InvalidOperationException(); }
set { throw new InvalidOperationException(); }
}
public override void Close()
{
Flush();
}
public override void Flush()
{
lock (this)
{
foreach (byte[] block in m_Blocks)
FreeBlock(block);
m_Blocks.Clear();
m_RPos = 0;
m_WPos = 0;
m_Size = 0;
}
}
public override void SetLength(long len)
{
throw new InvalidOperationException();
}
public override long Seek(long pos, SeekOrigin o)
{
throw new InvalidOperationException();
}
public override int Read(byte[] buf, int ofs, int count)
```

```
{
lock (this)
{
int Result = Peek(buf, ofs, count);
Advance(Result);
return Result;
}
}
public override void Write(byte[] buf, int ofs, int count)
{
lock (this)
{
int Left = count;
while (Left > 0)
{
int ToWrite = Math.Min(BlockSize – m_WPos, Left);
Array.Copy(buf, ofs + count – Left, GetWBlock(), m_WPos,
ToWrite);
m_WPos += ToWrite;
Left -= ToWrite;
}
m_Size += count;
}
}

// extra stuff
public int Advance(int count)
{
lock (this)
{
int SizeLeft = count;
while (SizeLeft > 0 && m_Size > 0)
{
if (m_RPos == BlockSize)
{
m_RPos = 0;
FreeBlock((byte[])m_Blocks[0]);
```

```
m_Blocks.RemoveAt(0);
}
int ToFeed = m_Blocks.Count == 1 ? Math.Min(m_WPos – m_RPos,
SizeLeft) : Math.Min(BlockSize – m_RPos, SizeLeft);
m_RPos += ToFeed;
SizeLeft -= ToFeed;
m_Size -= ToFeed;
}
return count – SizeLeft;
}
}
public int Peek(byte[] buf, int ofs, int count)
{
lock (this)
{
int SizeLeft = count;
int TempBlockPos = m_RPos;
int TempSize = m_Size;

int CurrentBlock = 0;
while (SizeLeft > 0 && TempSize > 0)
{
if (TempBlockPos == BlockSize)
{
TempBlockPos = 0;
CurrentBlock++;
}
int Upper = CurrentBlock < m_Blocks.Count - 1 ? BlockSize :
m_WPos; int ToFeed = Math.Min(Upper - TempBlockPos, SizeLeft);
Array.Copy((byte[])m_Blocks[CurrentBlock], TempBlockPos, buf,
ofs + count - SizeLeft, ToFeed); SizeLeft -= ToFeed;
TempBlockPos += ToFeed; TempSize -= ToFeed; } return count -
SizeLeft; } } } } [/csharp]
```