Computes Adler32 checksum for a stream of data

```
// Adler32.cs – Computes Adler32 data checksum of a data stream
// Copyright (C) 2001 Mike Krueger
//
// This file was translated from java, it was part of the GNU Classpath
// Copyright (C) 1999, 2000, 2001 Free Software Foundation, Inc.
//
// 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 2
// 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, write to the Free Software
// Foundation, Inc., 59 Temple Place – Suite 330, Boston, MA 02111-1307, USA.
//
// Linking this library statically or dynamically with other
```

```
modules is
// making a combined work based on this library. Thus, the terms and
// conditions of the GNU General Public License cover the whole
// combination.
//
// As a special exception, the copyright holders of this library give you
// permission to link this library with independent modules to produce an
// executable, regardless of the license terms of these independent
// modules, and to copy and distribute the resulting executable under
// terms of your choice, provided that you also meet, for each linked
// independent module, the terms and conditions of the license of that
// module. An independent module is a module which is not derived from
// or based on this library. If you modify this library, you may extend
// this exception to your version of the library, but you are not
// obligated to do so. If you do not wish to do so, delete this
// exception statement from your version.

using System;

namespace ICSharpCode.SharpZipLib.Checksums
{

///

/// Computes Adler32 checksum for a stream of data. An Adler32
/// checksum is not as reliable as a CRC32 checksum, but a lot
```

```
faster to
/// compute.
///
/// The specification for Adler32 may be found in RFC 1950.
/// ZLIB Compressed Data Format Specification version 3.3)
///
///
/// From that document:
///
/// “ADLER32 (Adler-32 checksum)
/// This contains a checksum value of the uncompressed data
/// (excluding any dictionary data) computed according to
Adler-32
/// algorithm. This algorithm is a 32-bit extension and
improvement
/// of the Fletcher algorithm, used in the ITU-T X.224 / ISO
8073
/// standard.
///
/// Adler-32 is composed of two sums accumulated per byte: s1
is
/// the sum of all bytes, s2 is the sum of all s1 values. Both
sums
/// are done modulo 65521. s1 is initialized to 1, s2 to zero.
The
/// Adler-32 checksum is stored as s2*65536 + s1 in most-
/// significant-byte first (network) order.”
///
/// “8.2. The Adler-32 algorithm
///
/// The Adler-32 algorithm is much faster than the CRC32
algorithm yet
/// still provides an extremely low probability of undetected
errors.
///
/// The modulo on unsigned long accumulators can be delayed
for 5552
/// bytes, so the modulo operation time is negligible. If the
bytes
/// are a, b, c, the second sum is 3a + 2b + c + 3, and so is
position
```

```
/// and order sensitive, unlike the first sum, which is just a
/// checksum. That 65521 is prime is important to avoid a
possible
/// large class of two-byte errors that leave the check
unchanged.
/// (The Fletcher checksum uses 255, which is not prime and
which also
/// makes the Fletcher check insensitive to single byte
changes 0 –
/// 255.)
///
/// The sum s1 is initialized to 1 instead of zero to make the
length
/// of the sequence part of s2, so that the length does not
have to be
/// checked separately. (Any sequence of zeroes has a Fletcher
/// checksum of zero.)”
///
///
///
public sealed class Adler32
{
///

/// largest prime smaller than 65536
///
const uint BASE = 65521;

///

/// Returns the Adler32 data checksum computed so far.
///
public long Value {
get {
return checksum;
}
}

///

/// Creates a new instance of the Adler32 class.
```

```
/// The checksum starts off with a value of 1.
///
public Adler32()
{
Reset();
}

///

/// Resets the Adler32 checksum to the initial value.
///
public void Reset()
{
checksum = 1;
}

///

/// Updates the checksum with a byte value.
///
///  /// The data value to add. The high byte of the int is
ignored.
///  public void Update(int value)
{
// We could make a length 1 byte array and call update again,
but I
// would rather not have that overhead
uint s1 = checksum & 0xFFFF;
uint s2 = checksum >> 16;

s1 = (s1 + ((uint)value & 0xFF)) % BASE;
s2 = (s1 + s2) % BASE;

checksum = (s2 < /// Updates the checksum with an array of
bytes.
///

///  /// The source of the data to update with.
///  public void Update(byte[] buffer)
{
```

```
if ( buffer == null ) {
throw new ArgumentNullException(“buffer”);
}

Update(buffer, 0, buffer.Length);
}

///

/// Updates the checksum with the bytes taken from the array.
///
///  /// an array of bytes
///  ///  /// the start of the data used for this update
///  ///  /// the number of bytes to use for this update
///  public void Update(byte[] buffer, int offset, int count)
{
if (buffer == null) {
throw new ArgumentNullException(“buffer”);
}

if (offset < 0) { #if NETCF_1_0 throw new ArgumentOutOfRangeException("offset"); #else throw new ArgumentOutOfRangeException("offset", "cannot be negative"); #endif } if ( count < 0 ) { #if NETCF_1_0 throw new ArgumentOutOfRangeException("count"); #else throw new ArgumentOutOfRangeException("count", "cannot be negative"); #endif } if (offset >= buffer.Length)
{
#if NETCF_1_0
throw new ArgumentOutOfRangeException(“offset”);
#else
throw new ArgumentOutOfRangeException(“offset”, “not a valid index into buffer”);
#endif
}

if (offset + count > buffer.Length)
{
```

```
#if NETCF_1_0
throw new ArgumentOutOfRangeException("count");
#else
throw new ArgumentOutOfRangeException("count", "exceeds buffer
size");
#endif
}

//(By Per Bothner)
uint s1 = checksum & 0xFFFF;
uint s2 = checksum >> 16;

while (count > 0) {
// We can defer the modulo operation:
// s1 maximally grows from 65521 to 65521 + 255 * 3800
// s2 maximally grows by 3800 * median(s1) = 2090079800 < 2^31
int n = 3800; if (n > count) {
n = count;
}
count -= n;
while (–n >= 0) {
s1 = s1 + (uint)(buffer[offset++] & 0xff);
s2 = s2 + s1;
}
s1 %= BASE;
s2 %= BASE;
}

checksum = (s2 <
```