How do you convert a byte array to a hexadecimal string, and vice versa?

Ask Question

1230

312

How can you convert a byte array to a hexadecimal string, and vice versa?

c# arrays hex

edited Jun 21 '17 at 22:49

community wiki
7 revs, 4 users 67%
alextansc

5 The accepted answer below appear to allocate a horrible amount of strings in the string to bytes conversion. I'm wondering how this impacts performance – Wim Coenen Mar 6 '09 at 16:41

7 The SoapHexBinary class does exactly what you want I think. – Mykroft Mar 31 '10 at 20:44

39 Answers

1 2 next

Home

1194

Either:

```
public static string ByteArrayToString(byte[] ba)
{
  StringBuilder hex = new StringBuilder(ba.Length * 2);
  foreach (byte b in ba)
    hex.AppendFormat("{0:x2}", b);
  return hex.ToString();
}
```

or:

```
public static string ByteArrayToString(byte[] ba)
{
  return BitConverter.ToString(ba).Replace("-","");
}
```

There are even more variants of doing it, for example here.

The reverse conversion would go like this:

```
public static byte[] StringToByteArray(String hex)
{
  int NumberChars = hex.Length;
  byte[] bytes = new byte[NumberChars / 2];
  for (int i = 0; i < NumberChars; i += 2)
    bytes[i / 2] = Convert.ToByte(hex.Substring(i, 2), 16);
  return bytes;
}
```

Using `Substring` is the best option in combination with `Convert.ToByte`. See this answer for more information. If you need better performance, you must avoid `Convert.ToByte` before you can drop `SubString`.

edited Sep 27 '18 at 9:23

community wiki
12 revs, 7 users 54%
Tomalak

17 You're using SubString. Doesn't this loop allocate a horrible amount of string objects? – Wim Coenen Mar 6 '09 at 16:36

26 Honestly - until it tears down performance dramatically, I would tend to ignore this and trust the Runtime and the GC to take care of it. – Tomalak Mar 6 '09 at 17:11

83 Because a byte is two nibbles, any hex string that validly represents a byte array must have an even character count. A 0 should not be added anywhere - to add one would be making an assumption about invalid data that is potentially dangerous. If anything, the StringToByteArray method should throw a FormatException if the hex string contains an odd number of characters. – David Boike Mar 9 '10 at 19:01

6 @00jt You must make an assumption that F == 0F. Either it is the same as 0F, or the input was clipped and F is actually the start of something you have not received. It is up to your context to make those assumptions, but I believe a general purpose function should reject odd characters as invalid instead of making that assumption for the calling code. – David Boike Jan 28 '13 at 15:35

9 @DavidBoike The question had NOTHING to do with "how to handle possibly clipped stream values" Its talking about a String. String myValue = 10.ToString("X"); myValue is "A" not "0A". Now go read that string back into bytes, oops you broke it. – 00jt Jan 30 '13 at 19:25

Performance Analysis

440

Note: new leader as of 2015-08-20.

I ran each of the various conversion methods through some crude `Stopwatch` performance testing, a run with a random sentence (n=61, 1000 iterations) and a run with a Project Gutenburg text

(n=1,238,957, 150 iterations). Here are the results, roughly from fastest to slowest. All measurements are in ticks (10,000 ticks = 1 ms) and all relative notes are compared to the [slowest] `StringBuilder` implementation. For the code used, see below or the test framework repo where I now maintain the code for running this.

Disclaimer

WARNING: Do not rely on these stats for anything concrete; they are simply a sample run of sample data. If you really need top-notch performance, please test these methods in an environment representative of your production needs with data representative of what you will use.

Results

- Lookup by byte `unsafe` (via CodesInChaos) (added to test repo by airbreather)
  - Text: 4,727.85 (105.2X)
  - Sentence: 0.28 (99.7X)
- Lookup by byte (via CodesInChaos)
  - Text: 10,853.96 (45.8X faster)
  - Sentence: 0.65 (42.7X faster)
- Byte Manipulation 2 (via CodesInChaos)
  - Text: 12,967.69 (38.4X faster)
  - Sentence: 0.73 (37.9X faster)
- Byte Manipulation (via Waleed Eissa)
  - Text: 16,856.64 (29.5X faster)
  - Sentence: 0.70 (39.5X faster)
- Lookup/Shift (via Nathan Moinvaziri)
  - Text: 23,201.23 (21.4X faster)
  - Sentence: 1.24 (22.3X faster)

- Lookup by nibble (via Brian Lambert)
  - Text: 23,879.41 (20.8X faster)
  - Sentence: 1.15 (23.9X faster)
- `BitConverter` (via Tomalak)
  - Text: 113,269.34 (4.4X faster)
  - Sentence: 9.98 (2.8X faster)
- `{SoapHexBinary}.ToString` (via Mykroft)
  - Text: 178,601.39 (2.8X faster)
  - Sentence: 10.68 (2.6X faster)
- `{byte}.ToString("X2")` (using `foreach`) (derived from Will Dean's answer)
  - Text: 308,805.38 (2.4X faster)
  - Sentence: 16.89 (2.4X faster)
- `{byte}.ToString("X2")` (using `{IEnumerable}.Aggregate`, requires System.Linq) (via Mark)
  - Text: 352,828.20 (2.1X faster)
  - Sentence: 16.87 (2.4X faster)
- `Array.ConvertAll` (using `string.Join`) (via Will Dean)
  - Text: 675,451.57 (1.1X faster)
  - Sentence: 17.95 (2.2X faster)
- `Array.ConvertAll` (using `string.Concat`, requires .NET 4.0) (via Will Dean)
  - Text: 752,078.70 (1.0X faster)
  - Sentence: 18.28 (2.2X faster)
- `{StringBuilder}.AppendFormat` (using `foreach`) (via Tomalak)
  - Text: 672,115.77 (1.1X faster)
  - Sentence: 36.82 (1.1X faster)

- `{StringBuilder}.AppendFormat` (using `{IEnumerable}.Aggregate`, requires System.Linq) (derived from Tomalak's answer)
  - Text: 718,380.63 (1.0X faster)
  - Sentence: 39.71 (1.0X faster)

Lookup tables have taken the lead over byte manipulation. Basically, there is some form of precomputing what any given nibble or byte will be in hex. Then, as you rip through the data, you simply look up the next portion to see what hex string it would be. That value is then added to the resulting string output in some fashion. For a long time byte manipulation, potentially harder to read by some developers, was the top-performing approach.

Your best bet is still going to be finding some representative data and trying it out in a production-like environment. If you have different memory constraints, you may prefer a method with fewer allocations to one that would be faster but consume more memory.

Testing Code

Feel free to play with the testing code I used. A version is included here but feel free to clone the repo and add your own methods. Please submit a pull request if you find anything interesting or want to help improve the testing framework it uses.

1. Add the new static method (`Func<byte[], string>`) to /Tests/ConvertByteArrayToHexString/Test.cs.
2. Add that method's name to the `TestCandidates` return value in that same class.
3. Make sure you are running the input version you want, sentence or text, by toggling the comments in `GenerateTestInput` in that same class.
4. Hit `F5` and wait for the output (an HTML dump is also generated in the /bin folder).

```
static string ByteArrayToHexStringViaStringJoinArrayConvertAll(byte[
    return string.Join(string.Empty, Array.ConvertAll(bytes, b => b.
}
static string ByteArrayToHexStringViaStringConcatArrayConvertAll(byt
    return string.Concat(Array.ConvertAll(bytes, b => b.ToString("X2
}
static string ByteArrayToHexStringViaBitConverter(byte[] bytes) {
    string hex = BitConverter.ToString(bytes);
    return hex.Replace("-", "");
}
static string ByteArrayToHexStringViaStringBuilderAggregateByteToStr
    return bytes.Aggregate(new StringBuilder(bytes.Length * 2), (sb,
sb.Append(b.ToString("X2"))).ToString();
}
static string ByteArrayToHexStringViaStringBuilderForEachByteToStrin
    StringBuilder hex = new StringBuilder(bytes.Length * 2);
    foreach (byte b in bytes)
        hex.Append(b.ToString("X2"));
    return hex.ToString();
}
static string ByteArrayToHexStringViaStringBuilderAggregateAppendFor
    return bytes.Aggregate(new StringBuilder(bytes.Length * 2), (sb,
sb.AppendFormat("{0:X2}", b)).ToString();
}
static string ByteArrayToHexStringViaStringBuilderForEachAppendForma
    StringBuilder hex = new StringBuilder(bytes.Length * 2);
    foreach (byte b in bytes)
        hex.AppendFormat("{0:X2}", b);
    return hex.ToString();
}
static string ByteArrayToHexViaByteManipulation(byte[] bytes) {
    char[] c = new char[bytes.Length * 2];
    byte b;
    for (int i = 0; i < bytes.Length; i++) {
        b = ((byte)(bytes[i] >> 4));
        c[i * 2] = (char)(b > 9 ? b + 0x37 : b + 0x30);
        b = ((byte)(bytes[i] & 0xF));
        c[i * 2 + 1] = (char)(b > 9 ? b + 0x37 : b + 0x30);
    }
    return new string(c);
}
static string ByteArrayToHexViaByteManipulation2(byte[] bytes) {
    char[] c = new char[bytes.Length * 2];
    int b;
    for (int i = 0; i < bytes.Length; i++) {
        b = bytes[i] >> 4;
```

```
            c[i * 2] = (char)(55 + b + (((b - 10) >> 31) & -7));
            b = bytes[i] & 0xF;
            c[i * 2 + 1] = (char)(55 + b + (((b - 10) >> 31) & -7));
        }
        return new string(c);
    }
    static string ByteArrayToHexViaSoapHexBinary(byte[] bytes) {
        SoapHexBinary soapHexBinary = new SoapHexBinary(bytes);
        return soapHexBinary.ToString();
    }
    static string ByteArrayToHexViaLookupAndShift(byte[] bytes) {
        StringBuilder result = new StringBuilder(bytes.Length * 2);
        string hexAlphabet = "0123456789ABCDEF";
        foreach (byte b in bytes) {
            result.Append(hexAlphabet[(int)(b >> 4)]);
            result.Append(hexAlphabet[(int)(b & 0xF)]);
        }
        return result.ToString();
    }
    static readonly uint* _lookup32UnsafeP = (uint*)GCHandle.Alloc(_Look
    GCHandleType.Pinned).AddrOfPinnedObject();
    static string ByteArrayToHexViaLookup32UnsafeDirect(byte[] bytes) {
        var lookupP = _lookup32UnsafeP;
        var result = new string((char)0, bytes.Length * 2);
        fixed (byte* bytesP = bytes)
        fixed (char* resultP = result) {
            uint* resultP2 = (uint*)resultP;
            for (int i = 0; i < bytes.Length; i++) {
                resultP2[i] = lookupP[bytesP[i]];
            }
        }
        return result;
    }
    static uint[] _Lookup32 = Enumerable.Range(0, 255).Select(i => {
        string s = i.ToString("X2");
        return ((uint)s[0]) + ((uint)s[1] << 16);
    }).ToArray();
    static string ByteArrayToHexViaLookupPerByte(byte[] bytes) {
        var result = new char[bytes.Length * 2];
        for (int i = 0; i < bytes.Length; i++)
        {
            var val = _Lookup32[bytes[i]];
            result[2*i] = (char)val;
            result[2*i + 1] = (char) (val >> 16);
        }
        return new string(result);
    }
```

```
static string ByteArrayToHexViaLookup(byte[] bytes) {
    string[] hexStringTable = new string[] {
        "00", "01", "02", "03", "04", "05", "06", "07", "08", "09",
"0D", "0E", "0F",
        "10", "11", "12", "13", "14", "15", "16", "17", "18", "19",
"1D", "1E", "1F",
        "20", "21", "22", "23", "24", "25", "26", "27", "28", "29",
"2D", "2E", "2F",
        "30", "31", "32", "33", "34", "35", "36", "37", "38", "39",
"3D", "3E", "3F",
        "40", "41", "42", "43", "44", "45", "46", "47", "48", "49",
"4D", "4E", "4F",
        "50", "51", "52", "53", "54", "55", "56", "57", "58", "59",
"5D", "5E", "5F",
        "60", "61", "62", "63", "64", "65", "66", "67", "68", "69",
"6D", "6E", "6F",
        "70", "71", "72", "73", "74", "75", "76", "77", "78", "79",
"7D", "7E", "7F",
        "80", "81", "82", "83", "84", "85", "86", "87", "88", "89",
"8D", "8E", "8F",
        "90", "91", "92", "93", "94", "95", "96", "97", "98", "99",
"9D", "9E", "9F",
        "A0", "A1", "A2", "A3", "A4", "A5", "A6", "A7", "A8", "A9",
"AD", "AE", "AF",
        "B0", "B1", "B2", "B3", "B4", "B5", "B6", "B7", "B8", "B9",
"BD", "BE", "BF",
        "C0", "C1", "C2", "C3", "C4", "C5", "C6", "C7", "C8", "C9",
"CD", "CE", "CF",
        "D0", "D1", "D2", "D3", "D4", "D5", "D6", "D7", "D8", "D9",
"DD", "DE", "DF",
        "E0", "E1", "E2", "E3", "E4", "E5", "E6", "E7", "E8", "E9",
"ED", "EE", "EF",
        "F0", "F1", "F2", "F3", "F4", "F5", "F6", "F7", "F8", "F9",
"FD", "FE", "FF",
    };
    StringBuilder result = new StringBuilder(bytes.Length * 2);
    foreach (byte b in bytes) {
        result.Append(hexStringTable[b]);
    }
    return result.ToString();
}
```

Update (2010-01-13)

Added Waleed's answer to analysis. Quite fast.

Update (2011-10-05)

Added `string.Concat Array.ConvertAll` variant for completeness (requires .NET 4.0). On par with `string.Join` version.

Update (2012-02-05)

Test repo includes more variants such as `StringBuilder.Append(b.ToString("X2"))` . None upset the results any. `foreach` is faster than `{IEnumerable}.Aggregate` , for instance, but `BitConverter` still wins.

Update (2012-04-03)

Added Mykroft's `SoapHexBinary` answer to analysis, which took over third place.

Update (2013-01-15)

Added CodesInChaos's byte manipulation answer, which took over first place (by a large margin on large blocks of text).

Update (2013-05-23)

Added Nathan Moinvaziri's lookup answer and the variant from Brian Lambert's blog. Both rather fast, but not taking the lead on the test machine I used (AMD Phenom 9750).

Update (2014-07-31)

Added @CodesInChaos's new byte-based lookup answer. It appears to have taken the lead on both the sentence tests and the full-text tests.

Update (2015-08-20)

Added airbreather's optimizations and `unsafe` variant to this answer's repo. If you want to play in the unsafe game, you can get

some huge performance gains over any of the prior top winners on both short strings and large texts.

edited Jul 5 '18 at 5:15

Kolappan Nathan
606 1 15 20

answered Mar 8 '09 at 21:56

patridge
20.9k 15 76 116

4 Despite making the code available for you to do the very thing you requested on your own, I updated the testing code to include Waleed answer. All grumpiness aside, it is much faster. – patridge Jan 13 '10 at 16:29

1 @CodesInChaos Done. And it won in my tests by quite a bit as well. I don't pretend to fully understand either of the top methods yet, but they are easily hidden from direct interaction. – patridge Jan 15 '13 at 18:01

5 This answer has no intention of answering the question of what is "natural" or commonplace. The goal is to give people some basic performance benchmarks since, when you need to do these conversion, you tend to do them a lot. If someone needs raw speed, they just run the benchmarks with some appropriate test data in their desired computing environment. Then, tuck that method away into an extension method where you never look its implementation again (e.g., `bytes.ToHexStringAtLudicrousSpeed()`). – patridge Apr 8 '13 at 20:37

1 Just produced a high performance lookup table based implementation. Its safe variant is about 30% faster than the current leader on my CPU. The unsafe variants are even faster. stackoverflow.com/a/24343727/445517 – CodesInChaos Jun 21 '14 at 17:12

0

There is also `XmlWriter.WriteBinHex` (see the MSDN page). This is very useful if you need to put the hexadecimal string into an XML stream.

Here is a standalone method to see how it works:

```
public static string ToBinHex(byte[] bytes)
{
    XmlWriterSettings xmlWriterSettings = new XmlWriterSettings(
    xmlWriterSettings.ConformanceLevel = ConformanceLevel.Fragme
    xmlWriterSettings.CheckCharacters = false;
    xmlWriterSettings.Encoding = ASCIIEncoding.ASCII;
    MemoryStream memoryStream = new MemoryStream();
    using (XmlWriter xmlWriter = XmlWriter.Create(memoryStream,
    {
        xmlWriter.WriteBinHex(bytes, 0, bytes.Length);
    }
    return Encoding.ASCII.GetString(memoryStream.ToArray());
}
```

edited Jul 5 '18 at 4:12

community wiki
2 revs, 2 users 92%
astrada

15

This is an answer to revision 4 of Tomalak's highly popular answer (and subsequent edits).

I'll make the case that this edit is wrong, and explain why it could be reverted. Along the way, you might learn a thing or two about some internals, and see yet another example of what premature optimization really is and how it can bite you.

tl;dr: Just use `Convert.ToByte` and `String.Substring` if you're in a hurry ("Original code" below), it's the best combination if you don't want to re-implement `Convert.ToByte`. Use something more advanced (see other answers) that doesn't use `Convert.ToByte` if you *need* performance. Do **not** use anything else other than

`String.Substring` in combination with `Convert.ToByte`, unless someone has something interesting to say about this in the comments of this answer.

warning: This answer may become obsolete *if* a `Convert.ToByte(char[], Int32)` overload is implemented in the framework. This is unlikely to happen soon.

*As a general rule, I don't much like to say "don't optimize prematurely", because nobody knows when "premature" is. The only thing you must consider when deciding whether to optimize or not is: "Do I have the time and resources to investigate optimization approaches properly?". If you don't, then it's too soon, wait until your project is more mature or until you need the performance (if there is a real need, then you will **make** the time). In the meantime, do the simplest thing that could possibly work instead.*

Original code:

```
public static byte[] HexadecimalStringToByteArray_Original(strin
{
    var outputLength = input.Length / 2;
    var output = new byte[outputLength];
    for (var i = 0; i < outputLength; i++)
        output[i] = Convert.ToByte(input.Substring(i * 2, 2), 16
    return output;
}
```

Revision 4:

```
public static byte[] HexadecimalStringToByteArray_Rev4(string in
{
    var outputLength = input.Length / 2;
    var output = new byte[outputLength];
    using (var sr = new StringReader(input))
    {
        for (var i = 0; i < outputLength; i++)
            output[i] = Convert.ToByte(new string(new char[2] {
(char)sr.Read() }), 16);
    }
```

```
    return output;
}
```

The revision avoids `String.Substring` and uses a `StringReader` instead. The given reason is:

> Edit: you can improve performance for long strings by using a single pass parser, like so:

Well, looking at the reference code for `String.Substring`, it's clearly "single-pass" already; and why shouldn't it be? It operates at byte-level, not on surrogate pairs.

It does allocate a new string however, but then you need to allocate one to pass to `Convert.ToByte` anyway. Furthermore, the solution provided in the revision allocates yet another object on every iteration (the two-char array); you can safely put that allocation outside the loop and reuse the array to avoid that.

```
public static byte[] HexadecimalStringToByteArray(string input)
{
    var outputLength = input.Length / 2;
    var output = new byte[outputLength];
    var numeral = new char[2];
    using (var sr = new StringReader(input))
    {
        for (var i = 0; i < outputLength; i++)
        {
            numeral[0] = (char)sr.Read();
            numeral[1] = (char)sr.Read();
            output[i] = Convert.ToByte(new string(numeral), 16);
        }
    }
    return output;
}
```

Each hexadecimal `numeral` represents a single octet using two digits (symbols).

But then, why call `StringReader.Read` twice? Just call its second overload and ask it to read two characters in the two-char array at once; and reduce the amount of calls by two.

```
public static byte[] HexadecimalStringToByteArray(string input)
{
    var outputLength = input.Length / 2;
    var output = new byte[outputLength];
    var numeral = new char[2];
    using (var sr = new StringReader(input))
    {
        for (var i = 0; i < outputLength; i++)
        {
            var read = sr.Read(numeral, 0, 2);
            Debug.Assert(read == 2);
            output[i] = Convert.ToByte(new string(numeral), 16);
        }
    }
    return output;
}
```

What you're left with is a string reader whose only added "value" is a parallel index (internal `_pos`) which you could have declared yourself (as `j` for example), a redundant length variable (internal `_length`), and a redundant reference to the input string (internal `_s`). In other words, it's useless.

If you wonder how `Read` "reads", just look at the code, all it does is call `String.CopyTo` on the input string. The rest is just book-keeping overhead to maintain values we don't need.

So, remove the string reader already, and call `CopyTo` yourself; it's simpler, clearer, and more efficient.

```
public static byte[] HexadecimalStringToByteArray(string input)
{
    var outputLength = input.Length / 2;
    var output = new byte[outputLength];
    var numeral = new char[2];
    for (int i = 0, j = 0; i < outputLength; i++, j += 2)
    {
        input.CopyTo(j, numeral, 0, 2);
```

```
            output[i] = Convert.ToByte(new string(numeral), 16);
        }
        return output;
    }
```

Do you really need a `j` index that increments in steps of two parallel to `i`? Of course not, just multiply `i` by two (which the compiler should be able to optimize to an addition).

```
public static byte[] HexadecimalStringToByteArray_BestEffort(str
{
    var outputLength = input.Length / 2;
    var output = new byte[outputLength];
    var numeral = new char[2];
    for (int i = 0; i < outputLength; i++)
    {
        input.CopyTo(i * 2, numeral, 0, 2);
        output[i] = Convert.ToByte(new string(numeral), 16);
    }
    return output;
}
```

What does the solution look like now? Exactly like it was at the beginning, only instead of using `String.Substring` to allocate the string and copy the data to it, you're using an intermediary array to which you copy the hexadecimal numerals to, then allocate the string yourself and copy the data *again* from the array and into the string (when you pass it in the string constructor). The second copy might be optimized-out if the string is already in the intern pool, but then `String.Substring` will also be able to avoid it in these cases.

In fact, if you look at `String.Substring` again, you see that it uses some low-level internal knowledge of how strings are constructed to allocate the string faster than you could normally do it, and it inlines the same code used by `CopyTo` directly in there to avoid the call overhead.

`String.Substring`

- Worst-case: One fast allocation, one fast copy.

- Best-case: No allocation, no copy.

Manual method

- Worst-case: Two normal allocations, one normal copy, one fast copy.
- Best-case: One normal allocation, one normal copy.

Conclusion? **If you want to use `Convert.ToByte(String, Int32)`** (because you don't want to re-implement that functionality yourself), there doesn't seem to be a way to beat `String.Substring`; all you do is run in circles, re-inventing the wheel (only with sub-optimal materials).

Note that using `Convert.ToByte` and `String.Substring` is a perfectly valid choice if you don't need extreme performance. Remember: only opt for an alternative if you have the time and resources to investigate how it works properly.

If there was a `Convert.ToByte(char[], Int32)`, things would be different of course (it would be possible to do what I described above and completely avoid `String`).

I suspect that people who report better performance by "avoiding `String.Substring`" also avoid `Convert.ToByte(String, Int32)`, which you should really be doing if you need the performance anyway. Look at the countless other answers to discover all the different approaches to do that.

Disclaimer: I haven't decompiled the latest version of the framework to verify that the reference source is up-to-date, I assume it is.

Now, it all sounds good and logical, hopefully even obvious if you've managed to get so far. But is it true?

```
Intel(R) Core(TM) i7-3720QM CPU @ 2.60GHz
    Cores: 8
    Current Clock Speed: 2600
    Max Clock Speed: 2600
```

```
--------------------
Parsing hexadecimal string into an array of bytes
--------------------
HexadecimalStringToByteArray_Original: 7,777.09 average ticks (over 
HexadecimalStringToByteArray_BestEffort: 8,550.82 average ticks (ove
HexadecimalStringToByteArray_Rev4: 9,218.03 average ticks (over 1000
```

Yes!

Props to Partridge for the bench framework, it's easy to hack. The input used is the following SHA-1 hash repeated 5000 times to make a 100,000 bytes long string.

```
209113288F93A9AB8E474EA78D899AFDBB874355
```

Have fun! (But optimize with moderation.)

edited Jul 5 '18 at 4:10

community wiki
3 revs, 2 users 97%
tne

62

You can use the BitConverter.ToString method:

```
byte[] bytes = {0, 1, 2, 4, 8, 16, 32, 64, 128, 256}
Console.WriteLine( BitConverter.ToString(bytes));
```

Output:

> 00-01-02-04-08-10-20-40-80-FF

More information: *BitConverter.ToString Method (Byte[])*

edited Jul 5 '18 at 4:07

community wiki
4 revs, 4 users 71%
Baget

10	Only answers half the question. – Sly Gryphon Jun 28 '11 at 6:49
2	Where is the second part of the answer? – Sawan Dec 25 '12 at 9:12

225

There's a class called SoapHexBinary that does exactly what you want.

```
using System.Runtime.Remoting.Metadata.W3cXsd2001;

public static byte[] GetStringToBytes(string value)
{
    SoapHexBinary shb = SoapHexBinary.Parse(value);
    return shb.Value;
}

public static string GetBytesToString(byte[] value)
{
    SoapHexBinary shb = new SoapHexBinary(value);
    return shb.ToString();
}
```

edited Jul 5 '18 at 4:07

community wiki
4 revs, 4 users 89%
Mykroft

33 SoapHexBinary is available from .NET 1.0 and is in mscorlib. Despite it's funny namespace, it does exactly what the question asked. – Sly Gryphon Jun 28 '11 at 6:48

2 Great find! Note that you will need to pad odd strings with a leading 0 for GetStringToBytes, like the other solution. – Carter Medlin Oct 31 '11 at 17:10

6 Interesting to see the Mono implementation here: github.com/mono/mono/blob/master/mcs/class/corlib/… – Jeremy Apr 29 '12 at 4:40

-1

Basic Solution With Extension Support

```
public static class Utils
{
    public static byte[] ToBin(this string hex)
    {
        int NumberChars = hex.Length;
        byte[] bytes = new byte[NumberChars / 2];
        for (int i = 0; i < NumberChars; i += 2)
            bytes[i / 2] = Convert.ToByte(hex.Substring(i, 2), 16);
        return bytes;
    }
    public static string ToHex(this byte[] ba)
    {
        return  BitConverter.ToString(ba).Replace("-", "");
    }
}
```

And use this class like below

```
byte[] arr1 = new byte[] { 1, 2, 3 };
string hex1 = arr1.ToHex();
byte[] arr2 = hex1.ToBin();
```

answered May 14 '18 at 12:53

community wiki
cahit beyaz

15

This problem could also be solved using a look-up table. This would require a small amount of static memory for both the encoder and decoder. This method will however be fast:

- Encoder table 512 bytes or 1024 bytes (twice the size if both upper and lower case is needed)
- Decoder table 256 bytes or 64 KiB (either a single char look-up or dual char look-up)

My solution uses 1024 bytes for the encoding table, and 256 bytes for decoding.

Decoding

```
private static readonly byte[] LookupTable = new byte[] {
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0xFF, (
0xFF, 0xFF,
  0xFF, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
```

```
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF
};

private static byte Lookup(char c)
{
  var b = LookupTable[c];
  if (b == 255)
    throw new IOException("Expected a hex character, got " + c);
  return b;
}

public static byte ToByte(char[] chars, int offset)
{
  return (byte)(Lookup(chars[offset]) << 4 | Lookup(chars[offset + 1
}
```

Encoding

```
private static readonly char[][] LookupTableUpper;
private static readonly char[][] LookupTableLower;

static Hex()
{
  LookupTableLower = new char[256][];
  LookupTableUpper = new char[256][];
  for (var i = 0; i < 256; i++)
  {
    LookupTableLower[i] = i.ToString("x2").ToCharArray();
    LookupTableUpper[i] = i.ToString("X2").ToCharArray();
  }
}
```

```
public static char[] ToCharLower(byte[] b, int bOffset)
{
  return LookupTableLower[b[bOffset]];
}

public static char[] ToCharUpper(byte[] b, int bOffset)
{
  return LookupTableUpper[b[bOffset]];
}
```

Comparison

```
StringBuilderToStringFromBytes:   106148
BitConverterToStringFromBytes:     15783
ArrayConvertAllToStringFromBytes:  54290
ByteManipulationToCharArray:        8444
TableBasedToCharArray:              5651 *
```

* this solution

Note

During decoding IOException and IndexOutOfRangeException could occur (if a character has a too high value > 256). Methods for de/encoding streams or arrays should be implemented, this is just a proof of concept.

edited Jun 21 '17 at 23:34

community wiki
3 revs, 2 users 82%
drphrozen

1 Memory usage of 256 bytes is negligible when you run code on the CLR. – dolmen Aug 21 '13 at 0:05

1

For performance I would go with drphrozens solution. A tiny optimization for the decoder could be to use a table for either char to get rid of the "<< 4".

Clearly the two method calls are costly. If some kind of check is made either on input or output data (could be CRC, checksum or whatever) the `if (b == 255)...` could be skipped and thereby also the method calls altogether.

Using `offset++` and `offset` instead of `offset` and `offset + 1` might give some theoretical benefit but I suspect the compiler handles this better than me.

```
private static readonly byte[] LookupTableLow = new byte[] {
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0xFF, (
0xFF, 0xFF,
  0xFF, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF
};

private static readonly byte[] LookupTableHigh = new byte[] {
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0x00, 0x10, 0x20, 0x30, 0x40, 0x50, 0x60, 0x70, 0x80, 0x90, 0xFF, (
0xFF, 0xFF,
```

```
  0xFF, 0xA0, 0xB0, 0xC0, 0xD0, 0xE0, 0xF0, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0xA0, 0xB0, 0xC0, 0xD0, 0xE0, 0xF0, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF,
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, (
0xFF, 0xFF
};

private static byte LookupLow(char c)
{
  var b = LookupTableLow[c];
  if (b == 255)
    throw new IOException("Expected a hex character, got " + c);
  return b;
}

private static byte LookupHigh(char c)
{
  var b = LookupTableHigh[c];
  if (b == 255)
    throw new IOException("Expected a hex character, got " + c);
  return b;
}

public static byte ToByte(char[] chars, int offset)
{
  return (byte)(LookupHigh(chars[offset++]) | LookupLow(chars[offset
}
```

This is just off the top of my head and has not been tested or benchmarked.

edited Jun 21 '17 at 23:33

community wiki
3 revs, 2 users 73%
ClausAndersen

Another fast function...

1

```
private static readonly byte[] HexNibble = new byte[] {
    0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7,
    0x8, 0x9, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
    0x0, 0xA, 0xB, 0xC, 0xD, 0xE, 0xF, 0x0,
    0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
    0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
    0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
    0x0, 0xA, 0xB, 0xC, 0xD, 0xE, 0xF
};

public static byte[] HexStringToByteArray( string str )
{
    int byteCount = str.Length >> 1;
    byte[] result = new byte[byteCount + (str.Length & 1)];
    for( int i = 0; i < byteCount; i++ )
        result[i] = (byte) (HexNibble[str[i << 1] - 48] << 4 | HexNi
1] - 48]);
    if( (str.Length & 1) != 0 )
        result[byteCount] = (byte) HexNibble[str[str.Length - 1] - 4
    return result;
}
```

edited Jun 21 '17 at 23:24

community wiki
2 revs, 2 users 70%
spacepille

From Microsoft's developers, a nice, simple conversion:

3

```
public static string ByteArrayToString(byte[] ba)
{
    // Concatenate the bytes into one long string
```

```
    return ba.Aggregate(new StringBuilder(32),
                        (sb, b) => sb.Append(b.ToString("X2"))
                        ).ToString();
}
```

While the above is clean an compact, performance junkies will scream about it using enumerators. You can get peak performance with an improved version of Tomolak's original answer:

```
public static string ByteArrayToString(byte[] ba)
{
   StringBuilder hex = new StringBuilder(ba.Length * 2);

   for(int i=0; i < ga.Length; i++)       // <-- Use for loop is fas
       hex.Append(ba[i].ToString("X2"));   // <-- ToString is faster

   return hex.ToString();
}
```

This is the fastest of all the routines I've seen posted here so far. Don't just take my word for it... performance test each routine and inspect its CIL code for yourself.

edited Jun 21 '17 at 23:22

community wiki
2 revs, 2 users 92%
Mark

1 The iterator is not the main problem of this code. You should benchmark `b.ToSting("X2")` . – dolmen Aug 20 '13 at 23:49

8

Why make it complex? This is simple in Visual Studio 2008:

C#:

```
string hex = BitConverter.ToString(YourByteArray).Replace("-", "");
```

VB:

```
Dim hex As String = BitConverter.ToString(YourByteArray).Replace("-"
```

edited Jun 21 '17 at 23:20

community wiki
2 revs, 2 users 91%
Craig Poulton

2 the reason is performance, when you need high performance solution. :) – Ricky Aug 4 '16 at 6:28

7

Not to pile on to the many answers here, but I found a fairly optimal (~4.5x better than accepted), straightforward implementation of the hex string parser. First, output from my tests (the first batch is my implementation):

```
Give me that string:
04c63f7842740c77e545bb0b2ade90b384f119f6ab57b680b7aa575a2f40939f

Time to parse 100,000 times: 50.4192 ms
Result as base64: BMY/eEJ0DHflRbsLKt6Qs4TxGfarV7aAt6pXWi9Ak58=
BitConverter'd: 04-C6-3F-78-42-74-0C-77-E5-45-BB-0B-2A-DE-90-B3-84-F
7-B6-80-B7-AA-57-5A-2F-40-93-9F
```

```
Accepted answer: (StringToByteArray)
Time to parse 100000 times: 233.1264ms
Result as base64: BMY/eEJ0DHflRbsLKt6Qs4TxGfarV7aAt6pXWi9Ak58=
BitConverter'd: 04-C6-3F-78-42-74-0C-77-E5-45-BB-0B-2A-DE-90-B3-84-F
7-B6-80-B7-AA-57-5A-2F-40-93-9F

With Mono's implementation:
Time to parse 100000 times: 777.2544ms
Result as base64: BMY/eEJ0DHflRbsLKt6Qs4TxGfarV7aAt6pXWi9Ak58=
BitConverter'd: 04-C6-3F-78-42-74-0C-77-E5-45-BB-0B-2A-DE-90-B3-84-F
7-B6-80-B7-AA-57-5A-2F-40-93-9F

With SoapHexBinary:
Time to parse 100000 times: 845.1456ms
Result as base64: BMY/eEJ0DHflRbsLKt6Qs4TxGfarV7aAt6pXWi9Ak58=
BitConverter'd: 04-C6-3F-78-42-74-0C-77-E5-45-BB-0B-2A-DE-90-B3-84-F
7-B6-80-B7-AA-57-5A-2F-40-93-9F
```

The base64 and 'BitConverter'd' lines are there to test for correctness. Note that they are equal.

The implementation:

```
public static byte[] ToByteArrayFromHex(string hexString)
{
  if (hexString.Length % 2 != 0) throw new ArgumentException("String
length");
  var array = new byte[hexString.Length / 2];
  for (int i = 0; i < hexString.Length; i += 2)
  {
    array[i/2] = ByteFromTwoChars(hexString[i], hexString[i + 1]);
  }
  return array;
}

private static byte ByteFromTwoChars(char p, char p_2)
{
  byte ret;
  if (p <= '9' && p >= '0')
  {
    ret = (byte) ((p - '0') << 4);
  }
  else if (p <= 'f' && p >= 'a')
  {
    ret = (byte) ((p - 'a' + 10) << 4);
```

```
  }
  else if (p <= 'F' && p >= 'A')
  {
    ret = (byte) ((p - 'A' + 10) << 4);
  } else throw new ArgumentException("Char is not a hex digit: " + p

  if (p_2 <= '9' && p_2 >= '0')
  {
    ret |= (byte) ((p_2 - '0'));
  }
  else if (p_2 <= 'f' && p_2 >= 'a')
  {
    ret |= (byte) ((p_2 - 'a' + 10));
  }
  else if (p_2 <= 'F' && p_2 >= 'A')
  {
    ret |= (byte) ((p_2 - 'A' + 10));
  } else throw new ArgumentException("Char is not a hex digit: " + p

  return ret;
}
```

I tried some stuff with `unsafe` and moving the (clearly redundant) character-to-nibble `if` sequence to another method, but this was the fastest it got.

(I concede that this answers half the question. I felt that the string->byte[] conversion was underrepresented, while the byte[]->string angle seems to be well covered. Thus, this answer.)

edited Jun 21 '17 at 23:10

community wiki
3 revs, 2 users 89%
Ben Mosher

1 For the followers of Knuth: I did this because I need to parse a few thousand hex strings every few minutes or so, so it's important that it be as fast as possible (in the inner loop, as it were). Tomalak's solution is not

notably slower if many such parses are not occurring. – Ben Mosher May 22 '12 at 17:01

13

Complement to answer by @CodesInChaos (reversed method)

```
public static byte[] HexToByteUsingByteManipulation(string s)
{
    byte[] bytes = new byte[s.Length / 2];
    for (int i = 0; i < bytes.Length; i++)
    {
        int hi = s[i*2] - 65;
        hi = hi + 10 + ((hi >> 31) & 7);

        int lo = s[i*2 + 1] - 65;
        lo = lo + 10 + ((lo >> 31) & 7) & 0x0f;

        bytes[i] = (byte) (lo | hi << 4);
    }
    return bytes;
}
```

Explanation:

`& 0x0f` is to support also lower case letters

`hi = hi + 10 + ((hi >> 31) & 7);` is the same as:

```
hi = ch-65 + 10 + (((ch-65) >> 31) & 7);
```

For '0'..'9' it is the same as `hi = ch - 65 + 10 + 7;` which is `hi = ch - 48` (this is because of `0xffffffff & 7`).

For 'A'..'F' it is `hi = ch - 65 + 10;` (this is because of `0x00000000 & 7`).

For 'a'..'f' we have to big numbers so we must subtract 32 from default version by making some bits `0` by using `& 0x0f`.

65 is code for `'A'`

48 is code for `'0'`

7 is the number of letters between `'9'` and `'A'` in the ASCII table (`...456789:;<=>?@ABCD...`).

edited Jun 21 '17 at 23:07

community wiki
3 revs, 2 users 93%
CoperNick

1

Here's my shot at it. I've created a pair of extension classes to extend string and byte. On the large file test, the performance is comparable to Byte Manipulation 2.

The code below for ToHexString is an optimized implementation of the lookup and shift algorithm. It is almost identical to the one by Behrooz, but it turns out using a `foreach` to iterate and a counter is faster than an explicitly indexing `for` .

It comes in 2nd place behind Byte Manipulation 2 on my machine and is very readable code. The following test results are also of interest:

ToHexStringCharArrayWithCharArrayLookup: 41,589.69 average ticks (over 1000 runs), 1.5X
ToHexStringCharArrayWithStringLookup: 50,764.06 average ticks (over 1000 runs), 1.2X
ToHexStringStringBuilderWithCharArrayLookup: 62,812.87 average ticks (over 1000 runs), 1.0X

Based on the above results it seems safe to conclude that:

1. The penalties for indexing into a string to perform the lookup vs. a char array are significant in the large file test.
2. The penalties for using a StringBuilder of known capacity vs. a char array of known size to create the string are even more significant.

Here's the code:

```
using System;

namespace ConversionExtensions
{
    public static class ByteArrayExtensions
    {
        private readonly static char[] digits = new char[] { '0', '1
'5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };

        public static string ToHexString(this byte[] bytes)
        {
            char[] hex = new char[bytes.Length * 2];
            int index = 0;

            foreach (byte b in bytes)
            {
                hex[index++] = digits[b >> 4];
                hex[index++] = digits[b & 0x0F];
            }

            return new string(hex);
        }
    }
}


using System;
using System.IO;

namespace ConversionExtensions
{
    public static class StringExtensions
    {
        public static byte[] ToBytes(this string hexString)
        {
            if (!string.IsNullOrEmpty(hexString) && hexString.Length
```

```
            {
                throw new FormatException("Hexadecimal string must n
contain an even number of digits to be valid.");
            }

            hexString = hexString.ToUpperInvariant();
            byte[] data = new byte[hexString.Length / 2];

            for (int index = 0; index < hexString.Length; index += 2
            {
                int highDigitValue = hexString[index] <= '9' ? hexSt
hexString[index] - 'A' + 10;
                int lowDigitValue = hexString[index + 1] <= '9' ? he
'0' : hexString[index + 1] - 'A' + 10;

                if (highDigitValue < 0 || lowDigitValue < 0 || highD
lowDigitValue > 15)
                {
                    throw new FormatException("An invalid digit was
hexadecimal digits are 0-9 and A-F.");
                }
                else
                {
                    byte value = (byte)((highDigitValue << 4) | (lowl
                    data[index / 2] = value;
                }
            }

            return data;
        }
    }
}
```

Below are the test results that I got when I put my code in @patridge's testing project on my machine. I also added a test for converting to a byte array from hexadecimal. The test runs that exercised my code are ByteArrayToHexViaOptimizedLookupAndShift and HexToByteArrayViaByteManipulation. The HexToByteArrayViaConvertToByte was taken from XXXX. The HexToByteArrayViaSoapHexBinary is the one from @Mykroft's answer.

> Intel Pentium III Xeon processor
>
> ```
> Cores: 4 <br/>
> Current Clock Speed: 1576 <br/>
> Max Clock Speed: 3092 <br/>
> ```
>
> Converting array of bytes into hexadecimal string representation
>
> ByteArrayToHexViaByteManipulation2: 39,366.64 average ticks (over 1000 runs), 22.4X
>
> ByteArrayToHexViaOptimizedLookupAndShift: 41,588.64 average ticks (over 1000 runs), 21.2X
>
> ByteArrayToHexViaLookup: 55,509.56 average ticks (over 1000 runs), 15.9X
>
> ByteArrayToHexViaByteManipulation: 65,349.12 average ticks (over 1000 runs), 13.5X
>
> ByteArrayToHexViaLookupAndShift: 86,926.87 average ticks (over 1000 runs), 10.2X
>
> ByteArrayToHexStringViaBitConverter: 139,353.73 average ticks (over 1000 runs),6.3X
>
> ByteArrayToHexViaSoapHexBinary: 314,598.77 average ticks (over 1000 runs), 2.8X
>
> ByteArrayToHexStringViaStringBuilderForEachByteToString: 344,264.63 average ticks (over 1000 runs), 2.6X
>
> ByteArrayToHexStringViaStringBuilderAggregateByteToString: 382,623.44 average ticks (over 1000 runs), 2.3X
>
> ByteArrayToHexStringViaStringBuilderForEachAppendFormat: 818,111.95 average ticks (over 1000 runs), 1.1X

> ByteArrayToHexStringViaStringConcatArrayConvertAll:
> 839,244.84 average ticks (over 1000 runs), 1.1X
>
> ByteArrayToHexStringViaStringBuilderAggregateAppendFormat:
> 867,303.98 average ticks (over 1000 runs), 1.0X
>
> ByteArrayToHexStringViaStringJoinArrayConvertAll: 882,710.28 average ticks (over 1000 runs), 1.0X

edited Jun 21 '17 at 23:05

community wiki
16 revs, 3 users 81%
JamieSee

-1

If you want to get the "4x speed increase" reported by wcoenen, then if it's not obvious: replace `hex.Substring(i, 2)` with `hex[i]+hex[i+1]`

You could also take it a step further and get rid of the `i+=2` by using `i++` in both places.

edited Jun 21 '17 at 23:02

community wiki
3 revs, 3 users 40%
Olipro

I just encountered the very same problem today, and I came across this code:

53

```
private static string ByteArrayToHex(byte[] barray)
{
    char[] c = new char[barray.Length * 2];
    byte b;
    for (int i = 0; i < barray.Length; ++i)
    {
        b = ((byte)(barray[i] >> 4));
        c[i * 2] = (char)(b > 9 ? b + 0x37 : b + 0x30);
        b = ((byte)(barray[i] & 0xF));
        c[i * 2 + 1] = (char)(b > 9 ? b + 0x37 : b + 0x30);
    }
    return new string(c);
}
```

Source: Forum post *byte[] Array to Hex String* (see the post by PZahra). I modified the code a little to remove the 0x prefix.

I did some performance testing to the code and it was almost eight times faster than using BitConverter.ToString() (the fastest according to patridge's post).

edited Jun 21 '17 at 22:58

community wiki
2 revs, 2 users 85%
Waleed Eissa

7 Only answers half the question. – Sly Gryphon Jun 28 '11 at 6:50

1 The accepted answer provides 2 excellent HexToByteArray methods, which represent the other half of the question. Waleed's solution answers the running question of how to do this without creating a huge number of strings in the process. – Brendten Eickstaedt Oct 10 '12 at 16:08

If you want more flexibility than `BitConverter`, but don't want those

89

clunky 1990s-style explicit loops, then you can do:

```
String.Join(String.Empty, Array.ConvertAll(bytes, x => x.ToString("X
```

Or, if you're using .NET 4.0:

```
String.Concat(Array.ConvertAll(bytes, x => x.ToString("X2")));
```

(The latter from a comment on the original post.)

edited Jun 21 '17 at 22:56

community wiki
5 revs, 4 users 55%
Will Dean

20 Even shorter: String.Concat(Array.ConvertAll(bytes, x => x.ToString("X2")) – Nestor Nov 25 '09 at 15:04

14 Even shorter: String.Concat(bytes.Select(b => b.ToString("X2"))) *[.NET4]* – Allon Guralnek Jun 16 '11 at 6:39

14 Only answers half the question. – Sly Gryphon Jun 28 '11 at 6:50

1 Why does the second one need .Net 4? String.Concat is in .Net 2.0. – Polyfun Oct 17 '14 at 11:42

1 those "90's style" loops are generally faster, but by a negligible enough amount that it wont matter in most contexts. Still worth mentioning though – Austin_Anderson Oct 24 '17 at 19:47

Extension methods (disclaimer: completely untested code, BTW...):

3

```
public static class ByteExtensions
{
    public static string ToHexString(this byte[] ba)
    {
        StringBuilder hex = new StringBuilder(ba.Length * 2);

        foreach (byte b in ba)
        {
            hex.AppendFormat("{0:x2}", b);
        }
        return hex.ToString();
    }
}
```

etc.. Use either of Tomalak's three solutions (with the last one being an extension method on a string).

edited Jun 21 '17 at 22:51

community wiki
3 revs, 3 users 71%
Pure.Krome

0

```
static string ByteArrayToHexViaLookupPerByte2(byte[] bytes)
{
        var result3 = new uint[bytes.Length];
        for (int i = 0; i < bytes.Length; i++)
                result3[i] = _Lookup32[bytes[i]];
        var handle = GCHandle.Alloc(result3, GCHandleType.Pinned);
        try
        {
                var result = Marshal.PtrToStringUni(handle.AddrOfPin
bytes.Length * 2);
                return result;
        }
        finally
        {
```

```
            handle.Free();
        }
    }
```

This functions in my tests is always the second entry after the unsafe implementation.

Unfortunately, the test bench is not so reliable... if you run it multiple times the list got shuffled so much that who knows after the unsafe which is really the fastest! It doesn't take into a account pre-warming, jit compilation time, and GC performance hits. I would like to have rewritten it to have more information, but I didn't had really the time for it.

answered May 2 '17 at 11:12

community wiki
Tommaso Ercole

1

Another way is by using `stackalloc` to reduce GC memory pressure:

```
static string ByteToHexBitFiddle(byte[] bytes)
{
        var c = stackalloc char[bytes.Length * 2 + 1];
        int b;
        for (int i = 0; i < bytes.Length; ++i)
        {
            b = bytes[i] >> 4;
            c[i * 2] = (char)(55 + b + (((b - 10) >> 31) & -7));
            b = bytes[i] & 0xF;
            c[i * 2 + 1] = (char)(55 + b + (((b - 10) >> 31) & -7));
        }
        c[bytes.Length * 2 ] = '\0';
        return new string(c);
}
```

answered May 20 '16 at 16:23

community wiki
Kel

133

When writing crypto code it's common to avoid data dependent branches and table lookups to ensure the runtime doesn't depend on the data, since data dependent timing can lead to side-channel attacks.

It's also pretty fast.

```
static string ByteToHexBitFiddle(byte[] bytes)
{
    char[] c = new char[bytes.Length * 2];
    int b;
    for (int i = 0; i < bytes.Length; i++) {
        b = bytes[i] >> 4;
        c[i * 2] = (char)(55 + b + (((b-10)>>31)&-7));
        b = bytes[i] & 0xF;
        c[i * 2 + 1] = (char)(55 + b + (((b-10)>>31)&-7));
    }
    return new string(c);
}
```

Ph'nglui mglw'nafh Cthulhu R'lyeh wgah'nagl fhtagn

> Abandon all hope, ye who enter here

An explanation of the weird bit fiddling:

1. `bytes[i] >> 4` extracts the high nibble of a byte
   `bytes[i] & 0xF` extracts the low nibble of a byte

2. `b - 10` is `< 0` for values `b < 10`, which will become a decimal digit is `>= 0` for values `b > 10`, which will become a letter from `A` to `F`.
3. Using `i >> 31` on a signed 32 bit integer extracts the sign, thanks to sign extension. It will be `-1` for `i < 0` and `0` for `i >= 0`.
4. Combining 2) and 3), shows that `(b-10)>>31` will be `0` for letters and `-1` for digits.
5. Looking at the case for letters, the last summand becomes `0`, and `b` is in the range 10 to 15. We want to map it to `A` (65) to `F` (70), which implies adding 55 (`'A'-10`).
6. Looking at the case for digits, we want to adapt the last summand so it maps `b` from the range 0 to 9 to the range `0` (48) to `9` (57). This means it needs to become -7 (`'0' - 55`). Now we could just multiply with 7. But since -1 is represented by all bits being 1, we can instead use `& -7` since `(0 & -7) == 0` and `(-1 & -7) == -7`.

Some further considerations:

- I didn't use a second loop variable to index into `c`, since measurement shows that calculating it from `i` is cheaper.
- Using exactly `i < bytes.Length` as upper bound of the loop allows the JITter to eliminate bounds checks on `bytes[i]`, so I chose that variant.
- Making `b` an int allows unnecessary conversions from and to byte.

edited Mar 24 '16 at 14:27

community wiki
6 revs, 3 users 90%
CodesInChaos

7 And `hex string` to `byte[] array` ? – AaA Jan 18 '13 at 7:56

13 +1 for properly citing your source after invoking that bit of black magic. All hail Cthulhu. – Edward Aug 2 '13 at 20:41

4 What about string to byte[]? – Syaiful Nizam Yahya Nov 6 '13 at 10:14

9 Nice! For those who need lowercase output, the expression obviously changes to `87 + b + (((b-10)>>31)&-39)` – eXavier Jan 6 '14 at 17:36

1 @AaA You said " `byte[] array` ", which literally means an array of byte arrays, or `byte[][]` . I was just poking fun. – CoolOppo Jun 10 '15 at 3:09

4

Inverse function for Waleed Eissa code (Hex String To Byte Array):

```
    public static byte[] HexToBytes(this string hexString)
    {
        byte[] b = new byte[hexString.Length / 2];
        char c;
        for (int i = 0; i < hexString.Length / 2; i++)
        {
            c = hexString[i * 2];
            b[i] = (byte)((c < 0x40 ? c - 0x30 : (c < 0x47 ? c - 0x3
4);
            c = hexString[i * 2 + 1];
            b[i] += (byte)(c < 0x40 ? c - 0x30 : (c < 0x47 ? c - 0x3
        }

        return b;
    }
```

Waleed Eissa function with lower case support:

```
    public static string BytesToHex(this byte[] barray, bool toLower
    {
        byte addByte = 0x37;
        if (toLowerCase) addByte = 0x57;
```

```
        char[] c = new char[barray.Length * 2];
        byte b;
        for (int i = 0; i < barray.Length; ++i)
        {
            b = ((byte)(barray[i] >> 4));
            c[i * 2] = (char)(b > 9 ? b + addByte : b + 0x30);
            b = ((byte)(barray[i] & 0xF));
            c[i * 2 + 1] = (char)(b > 9 ? b + addByte : b + 0x30);
        }

        return new string(c);
    }
```

answered Dec 17 '15 at 11:15

community wiki
Geograph

0

The following expands the excellent answer here by allowing native lower case option as well, and also handles null or empty input and makes this an extension method.

```
    /// <summary>
    /// Converts the byte array to a hex string very fast. Excellent
    /// with code lightly adapted from 'community wiki' here:
https://stackoverflow.com/a/14333437/264031
    /// (the function was originally named: ByteToHexBitFiddle). Now
lowerCase option
    /// to be input and allows null or empty inputs (null returns nu
empty).
    /// </summary>
    public static string ToHexString(this byte[] bytes, bool lowerCa
    {
        if (bytes == null)
            return null;
        else if (bytes.Length == 0)
            return "";
```

```
        char[] c = new char[bytes.Length * 2];

        int b;
        int xAddToAlpha = lowerCase ? 87 : 55;
        int xAddToDigit = lowerCase ? -39 : -7;

        for (int i = 0; i < bytes.Length; i++) {

            b = bytes[i] >> 4;
            c[i * 2] = (char)(xAddToAlpha + b + (((b - 10) >> 31) &

            b = bytes[i] & 0xF;
            c[i * 2 + 1] = (char)(xAddToAlpha + b + (((b - 10) >> 31
        }

        string val = new string(c);
        return val;
    }

    public static string ToHexString(this IEnumerable<byte> bytes, b
false)
    {
        if (bytes == null)
            return null;
        byte[] arr = bytes.ToArray();
        return arr.ToHexString(lowerCase);
    }
```

edited May 23 '17 at 12:03

community wiki
2 revs
Nicholas Petersen

Two mashups which folds the two nibble operations into one.

Probably pretty efficient version:

1

```
public static string ByteArrayToString2(byte[] ba)
{
    char[] c = new char[ba.Length * 2];
    for( int i = 0; i < ba.Length * 2; ++i)
    {
        byte b = (byte)((ba[i>>1] >> 4*((i&1)^1)) & 0xF);
        c[i] = (char)(55 + b + (((b-10)>>31)&-7));
    }
    return new string( c );
}
```

Decadent linq-with-bit-hacking version:

```
public static string ByteArrayToString(byte[] ba)
{
    return string.Concat( ba.SelectMany( b => new int[] { b >> 4, b &
=> (char)(55 + b + (((b-10)>>31)&-7))) );
}
```

And reverse:

```
public static byte[] HexStringToByteArray( string s )
{
    byte[] ab = new byte[s.Length>>1];
    for( int i = 0; i < s.Length; i++ )
    {
        int b = s[i];
        b = (b - '0') + ((('9' - b)>>31)&-7);
        ab[i>>1] |= (byte)(b << 4*((i&1)^1));
    }
    return ab;
}
```

edited Jul 16 '14 at 10:53

community wiki
3 revs
JJJ

1 HexStringToByteArray("09") returns 0x02 which is bad – CoperNick Jul 29 '13 at 10:26

59

Another lookup table based approach. This one uses only one lookup table for each byte, instead of a lookup table per nibble.

```
private static readonly uint[] _lookup32 = CreateLookup32();

private static uint[] CreateLookup32()
{
    var result = new uint[256];
    for (int i = 0; i < 256; i++)
    {
        string s=i.ToString("X2");
        result[i] = ((uint)s[0]) + ((uint)s[1] << 16);
    }
    return result;
}

private static string ByteArrayToHexViaLookup32(byte[] bytes)
{
    var lookup32 = _lookup32;
    var result = new char[bytes.Length * 2];
    for (int i = 0; i < bytes.Length; i++)
    {
        var val = lookup32[bytes[i]];
        result[2*i] = (char)val;
        result[2*i + 1] = (char) (val >> 16);
    }
    return new string(result);
}
```

I also tested variants of this using `ushort`, `struct{char X1, X2}`, `struct{byte X1, X2}` in the lookup table.

Depending on the compilation target (x86, X64) those either had the approximately same performance or were slightly slower than this variant.

And for even higher performance, its `unsafe` sibling:

```
private static readonly uint[] _lookup32Unsafe = CreateLookup32Unsaf
private static readonly uint* _lookup32UnsafeP =
(uint*)GCHandle.Alloc(_lookup32Unsafe,GCHandleType.Pinned).AddrOfPin

private static uint[] CreateLookup32Unsafe()
{
    var result = new uint[256];
    for (int i = 0; i < 256; i++)
    {
        string s=i.ToString("X2");
        if(BitConverter.IsLittleEndian)
            result[i] = ((uint)s[0]) + ((uint)s[1] << 16);
        else
            result[i] = ((uint)s[1]) + ((uint)s[0] << 16);
    }
    return result;
}

public static string ByteArrayToHexViaLookup32Unsafe(byte[] bytes)
{
    var lookupP = _lookup32UnsafeP;
    var result = new char[bytes.Length * 2];
    fixed(byte* bytesP = bytes)
    fixed (char* resultP = result)
    {
        uint* resultP2 = (uint*)resultP;
        for (int i = 0; i < bytes.Length; i++)
        {
            resultP2[i] = lookupP[bytesP[i]];
        }
    }
    return new string(result);
}
```

Or if you consider it acceptable to write into the string directly:

```
public static string ByteArrayToHexViaLookup32UnsafeDirect(byte[] by
{
    var lookupP = _lookup32UnsafeP;
    var result = new string((char)0, bytes.Length * 2);
    fixed (byte* bytesP = bytes)
    fixed (char* resultP = result)
    {
```

```
            uint* resultP2 = (uint*)resultP;
            for (int i = 0; i < bytes.Length; i++)
            {
                resultP2[i] = lookupP[bytesP[i]];
            }
        }
        return result;
    }
```

edited Jun 21 '14 at 17:00

community wiki
2 revs
CodesInChaos

3 This just answer half of the question... How about from hex string to bytes? – Narvalex Mar 8 '17 at 17:28

2

I'll enter this bit fiddling competition as I have an answer that also uses bit-fiddling to *decode* hexadecimals. Note that using character arrays may be even faster as calling `StringBuilder` methods will take time as well.

```
public static String ToHex (byte[] data)
{
    int dataLength = data.Length;
    // pre-create the stringbuilder using the length of the data * 2
    StringBuilder sb = new StringBuilder (dataLength * 2);
    for (int i = 0; i < dataLength; i++) {
        int b = data [i];

        // check using calculation over bits to see if first tuple is
        // isLetter is zero if it is a digit, 1 if it is a letter
        int isLetter = (b >> 7) & ((b >> 6) | (b >> 5)) & 1;
```

```
            // calculate the code using a multiplication to make up the d
            // a digit character and an alphanumerical character
            int code = '0' + ((b >> 4) & 0xF) + isLetter * ('A' - '9' - 1
            // now append the result, after casting the code point to a c
            sb.Append ((Char)code);

            // do the same with the lower (less significant) tuple
            isLetter = (b >> 3) & ((b >> 2) | (b >> 1)) & 1;
            code = '0' + (b & 0xF) + isLetter * ('A' - '9' - 1);
            sb.Append ((Char)code);
        }
        return sb.ToString ();
    }

    public static byte[] FromHex (String hex)
    {

        // pre-create the array
        int resultLength = hex.Length / 2;
        byte[] result = new byte[resultLength];
        // set validity = 0 (0 = valid, anything else is not valid)
        int validity = 0;
        int c, isLetter, value, validDigitStruct, validDigit, validLette
    validLetter;
        for (int i = 0, hexOffset = 0; i < resultLength; i++, hexOffset
            c = hex [hexOffset];

            // check using calculation over bits to see if first char is
            // isLetter is zero if it is a digit, 1 if it is a letter (up
            isLetter = (c >> 6) & 1;

            // calculate the tuple value using a multiplication to make d
    between
            // a digit character and an alphanumerical character
            // minus 1 for the fact that the letters are not zero based
            value = ((c & 0xF) + isLetter * (-1 + 10)) << 4;

            // check validity of all the other bits
            validity |= c >> 7; // changed to >>, maybe not OK, use UInt

            validDigitStruct = (c & 0x30) ^ 0x30;
            validDigit = ((c & 0x8) >> 3) * (c & 0x6);
            validity |= (isLetter ^ 1) * (validDigitStruct | validDigit)

            validLetterStruct = c & 0x18;
            validLetter = (((c - 1) & 0x4) >> 2) * ((c - 1) & 0x2);
            validity |= isLetter * (validLetterStruct | validLetter);
```

```
            // do the same with the lower (less significant) tuple
            c = hex [hexOffset + 1];
            isLetter = (c >> 6) & 1;
            value ^= (c & 0xF) + isLetter * (-1 + 10);
            result [i] = (byte)value;

            // check validity of all the other bits
            validity |= c >> 7; // changed to >>, maybe not OK, use UInt

            validDigitStruct = (c & 0x30) ^ 0x30;
            validDigit = ((c & 0x8) >> 3) * (c & 0x6);
            validity |= (isLetter ^ 1) * (validDigitStruct | validDigit)

            validLetterStruct = c & 0x18;
            validLetter = (((c - 1) & 0x4) >> 2) * ((c - 1) & 0x2);
            validity |= isLetter * (validLetterStruct | validLetter);
        }

        if (validity != 0) {
            throw new ArgumentException ("Hexadecimal encoding incorrect
        }

        return result;
    }
```

Converted from Java code.

answered Jan 20 '14 at 23:38

community wiki
Maarten Bodewes

5

Safe versions:

```
public static class HexHelper
{
    [System.Diagnostics.Contracts.Pure]
```

```
    public static string ToHex(this byte[] value)
    {
        if (value == null)
            throw new ArgumentNullException("value");

        const string hexAlphabet = @"0123456789ABCDEF";

        var chars = new char[checked(value.Length * 2)];
        unchecked
        {
            for (int i = 0; i < value.Length; i++)
            {
                chars[i * 2] = hexAlphabet[value[i] >> 4];
                chars[i * 2 + 1] = hexAlphabet[value[i] & 0xF];
            }
        }
        return new string(chars);
    }

    [System.Diagnostics.Contracts.Pure]
    public static byte[] FromHex(this string value)
    {
        if (value == null)
            throw new ArgumentNullException("value");
        if (value.Length % 2 != 0)
            throw new ArgumentException("Hexadecimal value length mu
"value");

        unchecked
        {
            byte[] result = new byte[value.Length / 2];
            for (int i = 0; i < result.Length; i++)
            {
                // 0(48) - 9(57) -> 0 - 9
                // A(65) - F(70) -> 10 - 15
                int b = value[i * 2]; // High 4 bits.
                int val = ((b - '0') + ((('9' - b) >> 31) & -7)) << 
                b = value[i * 2 + 1]; // Low 4 bits.
                val += (b - '0') + ((('9' - b) >> 31) & -7);
                result[i] = checked((byte)val);
            }
            return result;
        }
    }
}
```

Unsafe versions For those who prefer performance and do not afraid of unsafeness. About 35% faster ToHex and 10% faster FromHex.

```
public static class HexUnsafeHelper
{
    [System.Diagnostics.Contracts.Pure]
    public static unsafe string ToHex(this byte[] value)
    {
        if (value == null)
            throw new ArgumentNullException("value");

        const string alphabet = @"0123456789ABCDEF";

        string result = new string(' ', checked(value.Length * 2));
        fixed (char* alphabetPtr = alphabet)
        fixed (char* resultPtr = result)
        {
            char* ptr = resultPtr;
            unchecked
            {
                for (int i = 0; i < value.Length; i++)
                {
                    *ptr++ = *(alphabetPtr + (value[i] >> 4));
                    *ptr++ = *(alphabetPtr + (value[i] & 0xF));
                }
            }
        }
        return result;
    }

    [System.Diagnostics.Contracts.Pure]
    public static unsafe byte[] FromHex(this string value)
    {
        if (value == null)
            throw new ArgumentNullException("value");
        if (value.Length % 2 != 0)
            throw new ArgumentException("Hexadecimal value length mu
"value");

        unchecked
        {
            byte[] result = new byte[value.Length / 2];
            fixed (char* valuePtr = value)
            {
                char* valPtr = valuePtr;
```

```
                for (int i = 0; i < result.Length; i++)
                {
                    // 0(48) - 9(57) -> 0 - 9
                    // A(65) - F(70) -> 10 - 15
                    int b = *valPtr++; // High 4 bits.
                    int val = ((b - '0') + ((('9' - b) >> 31) & -7))
                    b = *valPtr++; // Low 4 bits.
                    val += (b - '0') + ((('9' - b) >> 31) & -7);
                    result[i] = checked((byte)val);
                }
            }
            return result;
        }
    }
}
```

BTW For benchmark testing initializing alphabet every time convert function called is wrong, alphabet must be const (for string) or static readonly (for char[]). Then alphabet-based conversion of byte[] to string becomes as fast as byte manipulation versions.

And of course test must be compiled in Release (with optimization) and with debug option "Suppress JIT optimization" turned off (same for "Enable Just My Code" if code must be debuggable).

edited Dec 20 '13 at 6:33

community wiki
3 revs
Maratius

1

Not optimized for speed, but more LINQy than most answers (.NET 4.0):

```
<Extension()>
Public Function FromHexToByteArray(hex As String) As Byte()
```

```
    hex = If(hex, String.Empty)
    If hex.Length Mod 2 = 1 Then hex = "0" & hex
    Return Enumerable.Range(0, hex.Length \ 2).Select(Function(i)
Convert.ToByte(hex.Substring(i * 2, 2), 16)).ToArray
End Function

<Extension()>
Public Function ToHexString(bytes As IEnumerable(Of Byte)) As String
    Return String.Concat(bytes.Select(Function(b) b.ToString("X2")))
End Function
```

answered Aug 30 '13 at 23:53

community wiki
MCattle

2

This version of ByteArrayToHexViaByteManipulation could be faster.

From my reports:

- ByteArrayToHexViaByteManipulation3: 1,68 average ticks (over 1000 runs), 17,5X
- ByteArrayToHexViaByteManipulation2: 1,73 average ticks (over 1000 runs), 16,9X
- ByteArrayToHexViaByteManipulation: 2,90 average ticks (over 1000 runs), 10,1X
- ByteArrayToHexViaLookupAndShift: 3,22 average ticks (over 1000 runs), 9,1X
- ...

```
static private readonly char[] hexAlphabet = new char[]
    {'0','1','2','3','4','5','6','7','8','9','A','B','C','D','E'
static string ByteArrayToHexViaByteManipulation3(byte[] bytes)
{
```

```
        char[] c = new char[bytes.Length * 2];
        byte b;
        for (int i = 0; i < bytes.Length; i++)
        {
            b = ((byte)(bytes[i] >> 4));
            c[i * 2] = hexAlphabet[b];
            b = ((byte)(bytes[i] & 0xF));
            c[i * 2 + 1] = hexAlphabet[b];
        }
        return new string(c);
    }
```

And I think this one is an optimization:

```
static private readonly char[] hexAlphabet = new char[]
    {'0','1','2','3','4','5','6','7','8','9','A','B','C','D','E'
static string ByteArrayToHexViaByteManipulation4(byte[] bytes)
{
    char[] c = new char[bytes.Length * 2];
    for (int i = 0, ptr = 0; i < bytes.Length; i++, ptr += 2)
    {
        byte b = bytes[i];
        c[ptr] = hexAlphabet[b >> 4];
        c[ptr + 1] = hexAlphabet[b & 0xF];
    }
    return new string(c);
}
```

edited Aug 23 '13 at 7:41

community wiki
JoseH

1 2 next

protected by Sheridan Feb 6 '15 at 10:03

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?