Base41 Encoding Specification

May 2015

1 The Base 41 encoding

Base 41 encoding is inspired by Ascii85 encoding. You can think of it as "Ascii85 encoding for two-octet (instead of four octet) multiples of data".

1.1 Recap: Ascii85

Ascii85 is an encoding which exploits the fact that $85^5 > 2^{32}$, while $84^5 < 2^{32}$. So, with five characters that take one of 85 possible values, we can encode 4 octets (in modern computers pretty much all bytes are octets). A lot of data has a multiple of 4 octets in size, but, even if it doesn't, you can pad. Thus, this encoding is rather efficient - it has a 25% overhead (a little more if you have to pad).

We can refer to a four-byte data element and its five-char representation as "(Ascii85) words".

The problem of Ascii85 is that it derives its alphabet with a simple ASCII value − 33 formula. So, that means it uses quotes and backslash, which are, in most text formats, including pretty much all the programming languages, used for string representation. Also, Ascii85 has a special rule for character z - it represents a "zero word" (word consisting of four bytes that equal 0). You don't have to encode it like that, but you have to decode it (if you encounter z). So, this is a little special handling that is not nice.

There are other encodings with the base of 85, such as Z85 (ZeroMQ Base85) and RFC 1924, that fix problems of the alphabet by defining it as a table which skips quotes and backslash. This is interesting, but it requires two tables (one for encoding and the other for decoding). On embedded systems, this can be a problem, both in size and in processing time.

Also, in embedded systems, sometimes it's inconvinient to pad data to have a multiple of 4 elements, especially for shorter data.

1.2 Base 41

So, we're like Ascii85, but for two octets in a word. We observe that $41^3 > 2^{16}$ and $40^3 < 2^{16}$. Looking at the ASCII chart, we choose the alphabet as ASCII value − 41. This alphabet does not contain quotes or backslash, so it is "string safe".

If the data has an odd number of elements... well, pad it. It is much less inconvinient to pad by one byte than by "one to three" bytes (and you pad three times less often than for Ascii85). We don't define what you pad with, it's up to the user, but be aware that the other side has to know that data is padded and with what.

It should be obvious that overhead of Base41 encoding is 50% (well, for odd sized data it is a little more, depending on the actual size, because you pad with one more byte).

1.3 Code

... is rather simple, which is not odd, as we designed the encoding with that in mind:

```
void base41_encode(uint8_t const *input, size_t n, char *output)
{
    uint8_t const *p = input;
    char *s = output;

    assert(n % 2 == 0);

    for (p = input; p < input + n; p += 2) {
        int x = *p + 256 * p[1];
        *s++ = (x % 41) + 41;
        x /= 41;
        *s++ = (x % 41) + 41;
        *s++ = x/41 + 41;
    }
    *s = '\0';
}

void base41_decode(char const *input, uint8_t *output)
{
    char const *s = input;
    size_t n = strlen(s);
    uint8_t *p = output;
    int i;

    assert(n % 3 == 0);

    for (i = 0; i < n; i += 3) {
        int x = (*s - 41) + 41 * (s[1] - 41) + 41*41*(s[2] - 41)
            ;
        *p++ = x % 256;
        *p++ = x / 256;
        s += 3;
    }
}
```

This code assumes that user knows how to allocate the needed memory for output. For some more defensive/paranoid code, you may want to pass the size of the output and check or assert it at the start.

In decoding, the `x / 256` may actually be larger than 255 for badly encoded string, but,

we don't care, as we assign to an octet, thus the "excess" will be thrown out. Paranoid code could check this and report an error.

1.4 Bad encoding - letter not in alphabet

The decoding of a string which has characters that are not in the Base41 alphabet is not defined. A more paranoid implementation can report an error, while a trusting implementation (like the one above) could simply "let it be" and just decode something.

But, this is not "undefined behavior", It is "undefined decoding". Behavior is always "decode each three letter 'word' to two octets". We just don't care what octets you decode to if the Base41 string is invalid. But, the decoder can't go formatting the disk, writing to memory it does not own...

1.5 Bad encoding - bad three letter words

The three letters of Base41 that form encoding of a 2-octet datum, can be referred to as a "Base41 word" and can be such that their encoding value actually exceeds 65535.

Such three-letter "words" are, let's say, four letter words. That is, those are invalid.

Decoder is free to report those as errors or decode anything it wants from it. That is, clamping at 65535 is OK, so is "clamping" to 0... Whatever.

So, this too is "undefined decoding", but not "undefined behavior".

1.6 Bad encoding - string with length other than multiple of three

This is totally defined. The decoder will simply ignore the excess chars.

That is, if you send four or five chars, decoder will decode only one Base41 word (the first). If you send six chars, decoder will decode two Base41 words.

So, if n is the length of the Base41 string, decoder will decode n/3 words (or (n/3)∗3 characters), where / is the integer division.

1.7 Just show me the alphabet

If you insist:

```
) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @
A B C D E F G H I J K L M N O P Q
```

That aligns nicely, but, maybe this view is easier to comprehend:

```
) * + , - . / 0 1 2
3 4 5 6 7 8 9 : ; <
= > ? @ A B C D E F
G H I J K L M N O P
Q
```

For those who love tables:

value	symbol	value	symbol	value	symbol	value	symbol
0	)	10	3	20	=	30	G
1	*	11	4	21	>	31	H
2	+	12	5	22	?	32	I
3	,	13	6	23	@	33	J
4	-	14	7	24	A	34	K
5	.	15	8	25	B	35	L

6	/	16	9	26	C	36	M
7	0	17	:	27	D	37	N
8	1	18	;	28	E	38	O
9	2	19	<	29	F	39	P
40	Q						

1.8 Discussion

1.8.1 What's this for?

It is good for encoding binary data in JSON (or similar) strings.

It is pretty fast and uses very little program memory and just a little stack and no static or const memory. This makes it suitable for embedded systems.

It is easy to implement in any language.

It is easy to just plug into your Javascript code. Javascript "modules" are a mess. Javascript doesn't really support them and all implementations thereof are just a bunch of conventions. Yes, nowadays npm and various tools that "build" your Javascript are in wide use, but, it's still a mess, albeit a mess with some vaccum cleaners you can use to help you clean it up from time to time. It is way easier to just put the code you need on the page itself, but that goes bad when there's lots of such code. But, Base41 is just a few lines of Javascript. While we're at it:

```
function base41_decode(input) {
    var rslt = []
    var n = input.length
    for (var i = 0; i < n; i += 3) {
        var x = (input.charCodeAt(i) - 41) + 41 * (input.
            charCodeAt(i+1) - 41) + 1681*(input.charCodeAt(i+2)
            - 41);
        rslt.push(x % 256, Math.floor(x / 256))
    }
    return rslt
}
function base41_encode(input) {
    var rslt = ""
    var n = input.length
    for (var i = 0; i < n; i += 2) {
        var x = input[i] + 256 * input[i+1];
        rslt = rslt.concat(String.fromCharCode((x % 41) + 41));
        x /= 41;
        rslt = rslt.concat(String.fromCharCode((x % 41) + 41),
            String.fromCharCode((x / 41) + 41))
    }
    return rslt
}
```

Again, this is the short, "non-paranoid" version. Add checks as per your liking.

1.8.2 Why not Base64, it is also string safe?

Base64 is rather weird:

- There are two Base64 encodings, one for email, other for URLs
- There are various Base64 variants, differring in alphabet, padding character(s), newline handling...
- A lot of implementations assume it is text that is encoded/decoded and fail if it is not
- It needs tables for encoding and decoding
- Pretty much no data is multiple of 3 in length, except by accident, so most of the time you'll have one or two padding chars at the end. For short data length this may be important.
- Code is weird and complex and tricky, with those 3 byte chunks...
- Since it's tricky, code also isn't very small or fast

1.8.3 This encoding requires division by 41, isn't that slow?

On some processors with the reference implementation, yes. Obviously, on such processors, you may be better of with some other encoding, or you may try some optimised version.

If speed is that important to you, the simplest encoding you can use is a Base16 variation, which is like Hex (Base16), but instead of 0123456789ABCDEF uses the ABCDEFGHIJKLMNOP alphabet (ASCCI value - 65). It has no padding issues, so it's as simple as it gets. The only problem is that it has a 100% overhead.

But, also, look below for tricks for efficient division by 41.

1.8.4 Why start at ASCII 41? And not, say, 48?

Obviously, it is not by accident. We want to escape quotes, but also #, $ and % which are often used for interpolation in strings in various programming languages.

Actually, the first ASCII char that is string safe would be value 40, char (. But, using 41 matches the "base", so we choose that instead.

It is true that ASCII 41 is), so maybe it would be nice to omit it, since we already omitted (. OTOH, since we only use the closing parenthesis, we avoid a problem in some interpretation of strings.

There is an interesting idea of having a "Base48". The nice thing about it is that division by 48 can be done with division by 16 (a shift by 4 bits) followed by division by three, which is a special form of division that can be done more efficiently than division by 41, which is a large prime number. So, on slow processors, especially those that don't have a HW divider, this would likely be faster than division by 41.

But:

- such processors can be better served by Base32 or Base16.
- actually, the trick for efficient division by 3 is to abuse multiplication, and that can be used for division by 41, too. So, unless somebody could employ some non-multiplication trick for division by 3, there is no benefit. This is further analyzed below
- 48^3 is much larger than 2^{16}, so there is a sense of waste
- We can't start the alphabet at ASCII 48, as that will "incorporate" the backslash. We can start at ASCII 42, which would end at ASCII 90 which is Z, but we do like the idea of "starting at the base".

Efficient division by 3 The trick is to multiply by some number that is the product of 3 and some power of two, then shift right by that power. Of course, the best such number is a representation of 1/3 in fixed-length binary.

So, for 32-bit processors (which are common nowadays in embedded world), this is division by 3:

```
(x * 0x5556) >> 16;
```

and the remainder is:

```
x - ((x * 0x5556) >> 16) * 3);
```

Thus the encoding loop of the Base48 would be:

```
for (p = input; p < input + n; p += 2) {
  int x = *p + 256 * p[1];
  int q = ((x>>4) * 0x5556) >> 16;
  *s++ (x - q*48) + 42;
  x = q;
  q = ((x>>4) * 0x5556) >> 16;
  *s++ (x - q*48) + 42;
  x = q;
  q = ((x>>4) * 0x5556) >> 16;
  *s++ q + 42;
}
```

For 16-bit processors, you have to do more tricks, and we don't have a solution present here. But you may try this code on a 16 bit processor, its emulation of 32-bit arithmetic may be faster than its 16 bit division.

Obviously, this is not very smart, as division by 48 can be expressed as (x*0x5556) >> 20, so a smart(er) loop would be:

```
for (p = input; p < input + n; p += 2) {
  int x = *p + 256 * p[1];
  int q = (x * 0x5556) >> 20;
  *s++ (x - q*48) + 42;
  x = q;
  q = (x * 0x5556) >> 20;
  *s++ (x - q*48) + 42;
  *s++ ((x * 0x5556) >> 20) + 42;
}
```

Efficient division by 41 Actually, the same trick can be applied to 41. That is, a close approximation of division by 41 is:

```
(x * 0x63e7) >> 20;
```

and the remainder is:

```
x - ((x * 0x63e7) >> 20) * 41);
```

and the encoding loop is:

```
  for (p = input; p < input + n; p += 2) {
    int x = *p + 256 * p[1];
    int q = (x * 0x63e7) >> 20;
    *s++ = (x - q*41) + 41;
    x = q;
    q = (x * 0x63e7) >> 20;
    *s++ = (x - q*41) + 41;
    *s++ = ((x * 0x63e7) >> 20) + 41;
}
```

So, actually, it's more efficient than "naive" division by 48 and as efficient as "smart" division by 48.

BTW, 1/41 is 0x063e70..., so, since last four bits shown are 0, it has a nice property: we can use the first five hex digits (20 bits), but actually get the precision of six hex digits (24 bits)! That is, since those last four bits are all 0, they don't influence the precision if you're OK with 24 bits precision. This means that it's very hard for an error in division to accumulate such as to make our calculations incorrect even in the *general* case, let alone in our usage.

This division error is much better than for 1/40 and a little better than 1/42. Also, it's by pure chance, 41 was not choosen because of it. But, while investigating this "division optimisation" we found it to be a good thing, which enforces our choice of base 41.

Thus, if you're on 32-bit (or 64-bit) processor, use this loop (unless you have a great processor that has a fast HW divider). For 64 bit processors, some more optimization are possible, but that is beyond the scope of our analysis (which is restricted to "why base 41 and not 48").

1.9 Base 41 alphabet has < and > so that is bad for XML

We don't like XML, so we don't care.

On a more serious note, encoding binary data in XML is not a great idea anyway. If you really want to, use Base64, or some "XML friendly" Ascii85 dialect.

Also, if you use XML, it is likely you're not on a constrained embedded system, so you're fine with added complexity of an encoding "higher" than Base41.

At long last, put the encoded data in a XML attribute (string), like:

```
<some-tag base41="ABC" />
```

1.10 Any interesting examples?

There are a few interesting examples of decoding and encoding.

Decoding:

Binary	String	**comment**
[49,49] (ASCII "11")	"/=0"	Base41 is mathematically correct, as 11 does not equal 0
[78,79] (ASCII "NO")	"0,5"	"no" means "maybe"

Encoding:

String	Binary	**comment**
”861”	[172,54] Latin1 NOT“6”	more proof of mathematical correctness of Base 41, just read it: “861 is not 6”
”:–P”	not allowed (would decode to [204,256])	Base41 is polite, as it doesn’t allow you to stick your tongue out to people