My Programming Process by Example: Implementing Base64

Paul R. Potts

January 2020

These days the programming work I do is mostly *embedded* programming, writing code that runs on small microcontrollers which have limited flash memory, RAM, and processing power. In the example below I describe how I implemented Base64 encoding and decoding as part of a bootloader, which had to fit into 16 kibibytes of flash memory, in order to leave as much space as possible free for the application programs.

About Base64

If you aren't familiar with Base64, you can read about Base64 on Wikipedia here. Slightly different versions of Base 64 are commonly used to encode data that is then transmitted in a variety of ways, but they have some key things in common:

> …Base64 is designed to carry data stored in binary formats across channels that only reliably support text content…

and the different versions all

> …represent binary data in an ASCII string format by translating it into a radix-64 representation.

"Radix-64" just means there are 64 possible symbols. "Channels that only reliably support text content" include e-mail messages and Internet URLs. I am using it to encode the binary contents of firmware updates, sent in packets. I'm doing this for several reasons. One reason is that if I limit the set of data bytes that can legally appear in the payload of the packets, I can more easily differentiate the packet delimiters — the "overhead" of the packet — from the data it is carrying.

To understand this, imagine that you are purchasing groceries, and lining them up on the conveyor belt in a checkout line. There are plastic dividers you can place on the belt to help make sure that the cashier knows the difference between the groceries you want to buy, and the groceries the person in front of you and behind you want to buy.

That works great because the grocery store doesn't actually sell those plastic dividers. But what if they did, and you wanted to buy a load of groceries that included some number of those plastic dividers? Things could get confusing fast, because it wouldn't be possible to reliably distinguish the order dividers from the order content.

There are various rules you could put in place to try to keep things working in an orderly way. You might make a rule that everyone *has* to buy 32 grocery items per order, and if they want to buy fewer than 32 items, they have to add candy bars to their order until they have 32. You might make a rule that each order had to start with a slip of paper indicating the number of items in the order, which would work as long as the store didn't sell slips of paper with numbers on them. You could come up with schemes to "encode" the plastic dividers — for example, two plastic dividers in a row means you want to buy one of them, while one divider means your order is finished. But each and every scheme adds various amounts of complexity and overhead to the store's grocery checkout process, and you'd better make sure that the checkout clerks (the decoders) *and* the customers (the encoders) know the rules.

I chose Base64 because it was one of the simplest ways to solve the problem. The set of 64 characters I use to encode my groceries doesn't include any of the plastic dividers — this set of characters is completely distinct from the set that indicates the start and end of the packet, or grocery order, so there is no need to "escape" any of the characters in the packet. There are other safeguards in my serial protocol to help keep things as reliable as possible, but I am not going to talk about those today. And one of the nice things about Base64 is that

> Base64 encoding converts three octets [bytes] into four encoded characters.

so there is always a nice three-to-four ratio, making it relatively easy to calculate the size of the encoded data from the plaintext, and vice-versa. The only complications, and they aren't all that complicated, arise when you want to encode fewer than 3 bytes, or decode fewer than 4 bytes.

Encoding Examples

Let's say we want to encode:

```
>The quick brown fox jumps over the lazy dog's back.
```

Base64 gives us this:

```
>VGhlIHF1aWNrIGJyb3duIGZveCBqdW1wcyBvdmVyIHRoZSBsYXp5IGRvZydzIGJhY2su
```

This doesn't really make the original string more compatible with e-mail or anything like that, so it may not be clear what advantage this has given us. But it becomes more obvious if we That doesn't really help with a string like this; we could have put the original string directly in an e-mail message. But let's say we want to encode this series of bytes (in hexadecimal):

```
0x00 0x01 0x02 0x03 0x80 0x81 0x82 0x83
0xF0 0xF1 0xF2 0xF3 0xFD 0xFE 0xFF
```

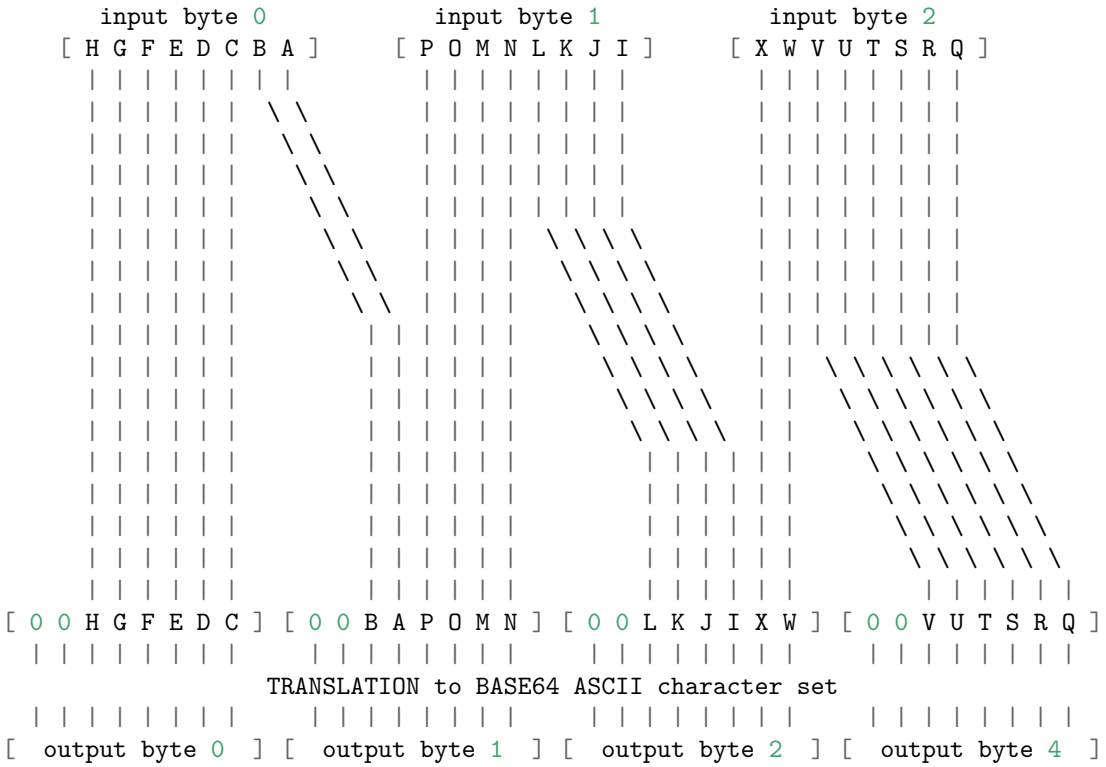
Those bytes can't go as-is into an e-mail message. But encoding the bytes with Base64 gives us this:

```
AAECA4CBgoPw8fLz/f7/
```

And that text *can*.

How Encoding Happens

The actual Base64 standard I'm using is here, if you want to read more about it; I use the "URL and Filename safe" version of the Base64 alphabet, which does not use slashes. But before I continue, let's take a look at a diagram that shows how 24 bits from three plaintext input bytes get shuffled into four output bytes and then these output bytes get translated into the Base64 character set:



In pseudo-code, we have something like:

```
output byte 0 =
    bits 7..2 of input byte 0 (H, G, F, E D, and C)
        shifted right by 2
```

```
output byte 1 =
    bits 1..0 of input byte 0 (A and B)
        shifted left by 4, and
    bits 7..4 of input byte 1 (P, O, M, and N)
        shifted right by 4

output byte 2 =
    bits 3..0 of input byte 1 (L, K, J, and I)
        shifted left by 2, and
    bits 7..6 of input byte 2 (X and W)
        shifted right by 6

output byte 3 =
    bits 5..0 of input byte 2 (V, U, T, S, R, and Q)
```

The values of these 4 byte are in the range 0..63 and so the high 2 bits are always zero. These values are then translated into our Base64 ASCII character set.

Illegal Encoded Data Lengths

Note that to decode the Base64-encoded data, we really just work our way through the diagram from the bottom up, instead of the top down.

This illustrates something interesting about the Base64 encoding. Correctly encoded data *can't* have a length of 1, or 5, or 9, etc. because when we have one input byte, we get two output bytes. Two input bytes will turn into three output bytes, and three input bytes will turn into four output bytes. *No number of input bytes will produce one output byte, or any multiple of four output bytes.*

This means that figuring out the *length* of the plaintext from the length of the encoded text is a bit complicated, because there are "illegal" values.

To handle our legal values, we could just write:

```
plaintext_length = encoded_length * 3 / 4
```

But the "illegal" values of 1, 5, etc. will produce zero. To filter out these illegal values, we can check if:

```
((encoded_length % 4) == 1)
```

which should *never* be the case in correctly encoded Base64 data.

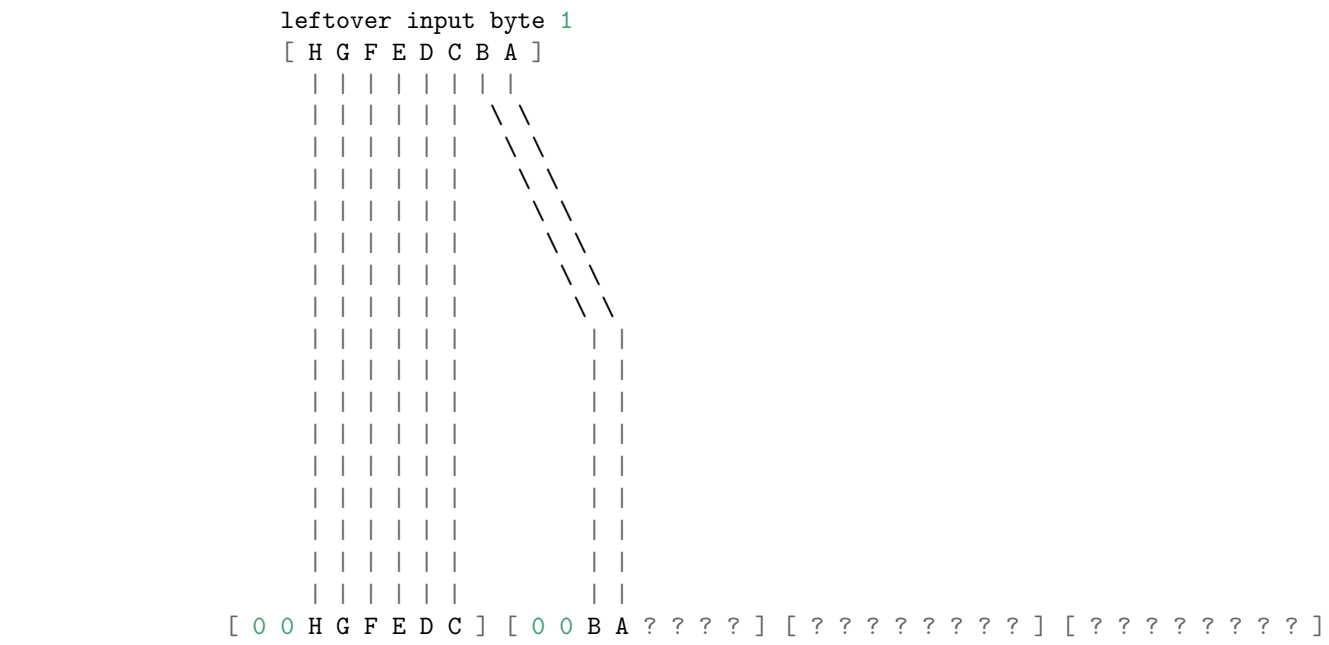
That uses the % operator, which is known as *modulo*. If you aren't familiar with modular arithmetic, you can read up on it here:

> In modular arithmetic, numbers "wrap around" upon reaching a given fixed quantity, which is known as the modulus (which would be 12 in the case of hours on a clock, or 60 in the case of minutes or seconds on a clock).

The % operator in C can be better thought of as the remainder operator, since it doesn't behave like a true modulo operation when the left hand side is negative. And because different versions of the C standard make different guarantees about the modulo operator when applied to negative numbers, I generally avoid using it on signed values.

If we are encoding, and our input data stream contains a multiple of 3 bytes, then the encoding behavior will be covered by the three-in, four-out diagram above. All the bits are accounted for. If we are decoding, and our input data stream contains a multiple of 4 bytes, we're just doing that process backwards, and again all the bits are accounted for. But my encoding and decoding functions have to be able to handle any legitimate number of data bytes, so we'd better make sure we know what happens to leftover bytes when we're *encoding* a *partial* group of three bytes, or *decoding* a *partial* group of four bytes.

One input byte will yield a group of four output bytes where *only the contents of the first byte are fully specified* before translation into the Base64 character set. So we need to come up with something reasonable to do with the unspecified bits. We will stipulate that if we write a byte, we will *always write the two high bits as zero*, since we we only support 6-bit values (0 through 63), but what about the low bits of a partially-filled byte, and what about the bytes in the output we don't write?



In our pseudocode:

```
output byte 0 =
    bits 7..2 of input byte 0 (H, G, F, E D, and C)
```

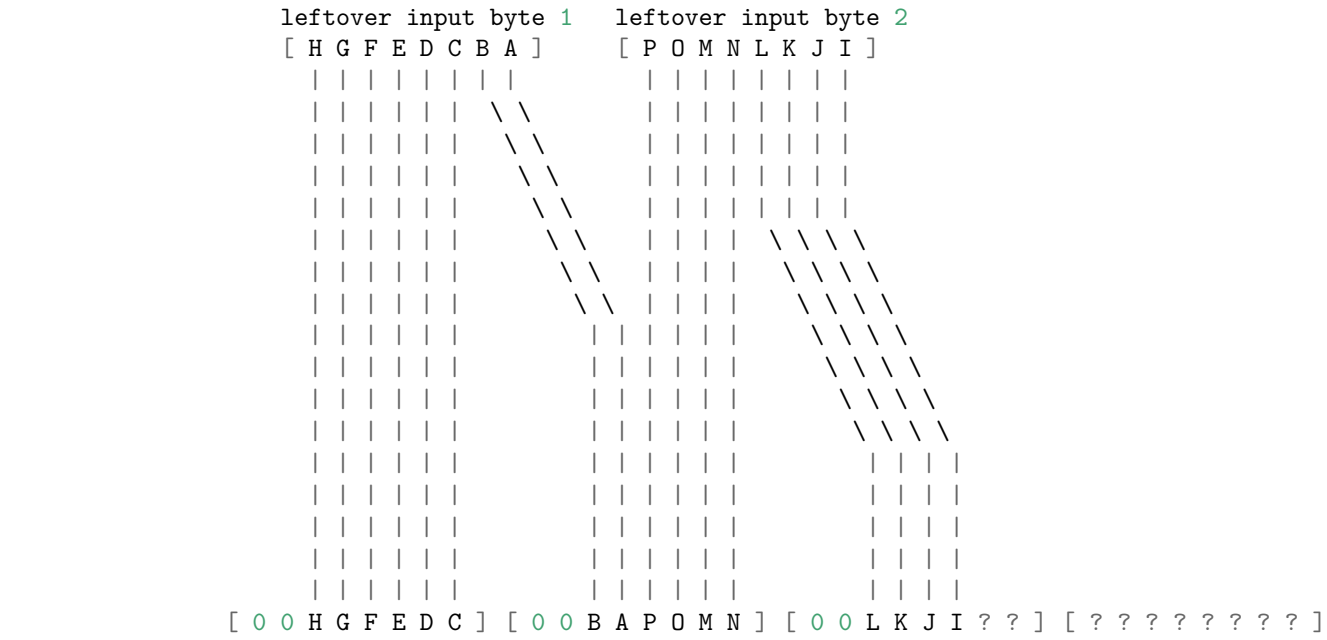
```
        shifted right by 2

output byte 1 =
    bits 1..0 of input byte 0 (A and B)
        shifted left by 4, and
    ?

output byte 2 =
    ?

output byte 3 =
    ?
```

Two leftover input bytes will yield a group of four output bytes where only the contents of the first byte and the second byte are fully specified:



In pseudo-code, we have something like:

```
output byte 0 =
    bits 7..2 of input byte 0 (H, G, F, E D, and C)
        shifted right by 2

output byte 1 =
    bits 1..0 of input byte 0 (A and B)
        shifted left by 4, and
    bits 7..4 of input byte 1 (P, O, M, and N)
        shifted right by 4
```

```
output byte 2 =
    bits 3..0 of input byte 1 (L, K, J, and I)
        shifted left by 2, and
    ?

output byte 3 =
    ?
```

We have similar situations going the other way (decoding the Base64-encoded data), in cases where we are decoding two or three bytes instead of four, but I'll let you draw the diagrams for those cases.

Unpredictable things like those question marks, in code, are *bad*. Sometimes things like this can result in security holes, making it possible to get access to leftover data that comes from reused memory. So, we should come up with rules to make everything well-defined. We can say that bits that are part of a *partly-defined* output byte will be set to zero, but *fully undefined* output bytes — the ones that contain all question marks in the above diagrams — won't be written at all, so our code should never accidentally stuff leftover data into them. In other words, our encoder and decoder function won't write bytes when it doesn't have any data to stuff into them. That seems simple enough. This fixes some of the question marks in the pseudo-code. In the case of one input byte, we can modify our pseudo-code for output byte 1:

```
output byte 1 =
    bits 1..0 of input byte 0 (A and B)
        shifted left by 4, and
    bits 7, 6, 3, 2, 1, and 0 = 0
```

In the case of two input bytes, we can modify our pseudo-code for output byte 2:

```
output byte 2 =
    bits 3..0 of input byte 1 (L, K, J, and I)
        shifted left by 2, and
    bits 7, 6, 1, and 0 = 0
```

This fixes the questions about what goes into the partially-specified *output* bytes, and as I mentioned above, the bytes we don't have anything to write into won't be written at all. This also means that our simple implementation never *pads* the output with extra bytes.

Padding

The Base64 RFC defines a scheme for padding. The missing bytes from a final incomplete group of four ASCII characters are filled with a suitable 65th ASCII character, the equals sign. You might see either one or two of those padding characters on the last line of a Base64-encoded block of ASCII data, in (for example) an e-mail message. Because a partial group of four will always have at

least two characters in it, we will only ever see one or two padding characters, not three.

Padding is not strictly required in most cases, since we can determine the number of output characters in the last group by just counting characters. But some applications use it, and it isn't hard to handle padding bytes, since they can be distinguished from the other 64 legal ASCII characters. So for maximum compatibility, let's make our decoding function handle padding bytes if they are present, even though our encoding function doesn't use them. This is an application of Postel's Law: "Be conservative in what you send; be liberal in what you accept."

The FreeBSD Implementation

After studying the problem and drawing some diagrams to make sure I fully understood the algorithms needed, I looked around to see if there was a good open-source implementation of Base64 encoding that I could borrow. There are plenty of implementations out there, and some of them have licenses that would allow me to borrow them. But all of them seemed overly complicated for what I wanted to do. For example, here's the encoding function from FreeBSD:

```
unsigned char * base64_encode(const unsigned char *src, size_t len,
                    size_t *out_len)
{
    unsigned char *out, *pos;
    const unsigned char *end, *in;
    size_t olen;
    int line_len;

    olen = len * 4 / 3 + 4; /* 3-byte blocks to 4-byte */
    olen += olen / 72; /* line feeds */
    olen++; /* nul termination */
    if (olen < len)
        return NULL; /* integer overflow */
    out = os_malloc(olen);
    if (out == NULL)
        return NULL;

    end = src + len;
    in = src;
    pos = out;
    line_len = 0;
    while (end - in >= 3) {
        *pos++ = base64_table[in[0] >> 2];
        *pos++ = base64_table[((in[0] & 0x03) << 4) | (in[1] >> 4)];
        *pos++ = base64_table[((in[1] & 0x0f) << 2) | (in[2] >> 6)];
        *pos++ = base64_table[in[2] & 0x3f];
```

```
            in += 3;
            line_len += 4;
            if (line_len >= 72) {
                *pos++ = '\n';
                line_len = 0;
            }
        }

        if (end - in) {
            *pos++ = base64_table[in[0] >> 2];
            if (end - in == 1) {
                *pos++ = base64_table[(in[0] & 0x03) << 4];
                *pos++ = '=';
            } else {
                *pos++ = base64_table[((in[0] & 0x03) << 4) |
                    (in[1] >> 4)];
                *pos++ = base64_table[(in[1] & 0x0f) << 2];
            }
            *pos++ = '=';
            line_len += 4;
        }

        if (line_len)
            *pos++ = '\n';

        *pos = '\0';
        if (out_len)
            *out_len = pos - out;
        return out;
    }
```

It is not too difficult to figure out what this function is doing. First there are a few lines that do error checking and memory allocation:

```
olen = len * 4 / 3 + 4; /* 3-byte blocks to 4-byte */
olen += olen / 72; /* line feeds */
olen++; /* nul termination */
if (olen < len)
    return NULL; /* integer overflow */
out = os_malloc(olen);
if (out == NULL)
    return NULL;
```

This code precalculates the output length, then adds the number of extra characters needed to break the output into 72-character lines, then adds one more for C string termination. Then it compares the calculated output length against the input length; if the calculated length, which should be larger, is

smaller, then it concludes that the output length calculation has overflowed and wrapped around, and so is nonsense. This test did does not look entirely safe to me at first glance, though; I wondered if it might be possible to select an input length close to 2^31 which would result in a wrapped-around **olen** which is greater than or equal to **len**. But thinking about it a little more, I concluded that it is probably correct; it's rare these days to find an actual bug in the BSD code base, as it has been looked over very thoroughly by lots of smart people over the years.

Then the code allocates an output buffer, and checks to see if the allocation failed.

That's a lot of things going on in one function. I don't actually want to break my output data into lines, since I'm not formatting them for an e-mail message. I'm not working with C string functions, so I don't want the NUL termination. And I'm writing this to run on a small embedded system, so I don't actually want to allocate memory, either. So this function is doing much more than I want a Base64 encoding function to do. And there is something odd about the length calculation in the first line:

```
olen = len * 4 / 3 + 4;
```

It's not right. Well, it's not right for *my* use case.

Let's say I wanted to write a function to get the encoded size from the plaintext size. That function has the prototype something like this:

```
int Base_64_Get_Encoded_Size( int plaintext_size );
```

In everyday math, the ratio is 4:3, so you might think that you can simply write an expression like this:

```
encoded_length = plaintext_length * 4 / 3
```

But that doesn't actually do what we want. We want an integer result, and integer division in C truncates the result. So a little thought and experimentation in Excel shows that we want to round up before dividing. The formula in Excel looks like this:

```
FLOOR((plaintext_length * 4 + 2) / 3, 1)
```

In C, we don't need to include the **FLOOR**, so that expression looks like

```
(plaintext_size * 4 + 2) / 3
```

Why the discrepancy between this and the **len * 4 / 3 + 4** expression in the FreeBSD code above? Well, it turns out that their calculation assumes that we are going to insert the padding characters I mentioned above. There's no option *not* to insert it. I don't want the overhead of padding all the time. If I borrowed this code, I would want to change that. This is an example of why, if we are considering borrowing code from other sources, it is critical to understand exactly what each line does.

The list of things I would have to change to meet my needs is starting to get pretty long. But let's look at how the function actually works, just to make sure we understand it. The next group of lines of code looks like this:

```
end = src + len;
in = src;
pos = out;
line_len = 0;
while (end - in >= 3) {
    *pos++ = base64_table[in[0] >> 2];
    *pos++ = base64_table[((in[0] & 0x03) << 4) | (in[1] >> 4)];
    *pos++ = base64_table[((in[1] & 0x0f) << 2) | (in[2] >> 6)];
    *pos++ = base64_table[in[2] & 0x3f];
    in += 3;
    line_len += 4;
    if (line_len >= 72) {
        *pos++ = '\n';
        line_len = 0;
    }
}
```

I'm not sure why they are using separate **in** and **pos** values; those **src** and **out** parameters are copies of the data passed by the calling function. They aren't declared **const**, so they can be updated; there's no need to create these local variables and copy in the parameters. I'm not quite sure why this function uses them. It may have to do with memory safety, which is not something I usually need to worry about when writing code for my small chips. Or it may just be a matter of style.

In any case, let's move on to the **while** loop. What it does is pretty simple. It just tries to write as many four-character output groups as possible. It does this as long as there are at least three input bytes left. We wouldn't need or want that conditional logic to break lines of text, but the rest is not too complicated.

After we've run out of full four-character output groups we can write, the next group of lines does "mopping up," to handle any remaining work:

```
if (end - in) {
    *pos++ = base64_table[in[0] >> 2];
    if (end - in == 1) {
        *pos++ = base64_table[(in[0] & 0x03) << 4];
        *pos++ = '=';
    } else {
        *pos++ = base64_table[((in[0] & 0x03) << 4) |
            (in[1] >> 4)];
        *pos++ = base64_table[(in[1] & 0x0f) << 2];
    }
    *pos++ = '=';
```

```
    line_len += 4;
}

if (line_len)
    *pos++ = '\n';
```

If there are any leftover input bytes at all, the code writes the bits from the first input byte to the first output byte. This will only keep six bits from the input byte and because the type is unsigned, zero bits will be shifted in (this is a "logical shift"), so the output byte is fully defined. Then it handles several cases:

- If there is exactly one leftover input byte, the code writes the remaining bits from the first input byte into the second output byte (remember our diagram above: in each group, the bits from the first input byte get split across the first two output bytes), and then adds a padding character to the output. Bits in the second output byte that don't come from the single input byte will also be written as zero, because of the way the value is calculated: it keeps two bits from the first input byte and again, because the type is unsigned, zero bits will be shifted in (this is also a "logical shift").
- If there is more than one leftover input byte (and the only remaining option here is that there are two leftover input bytes), the code writes the remaining bits from the first input byte combined with bits from the second input byte into the second output byte, and then writes the rest of the bits from the second input byte into the third output byte. The bits in the three output bytes that don't come from either the first or second input bytes will wind up as zero.

After handling these two cases, the code adds a padding character, so we always end up with either two or one padding character to pad out the group of four output bytes.

Then, if the code has written any characters, it inserts another line break, so that the last line of output data will always have a line break at the end, even if it is shorter than 72 characters.

Then, there are a few more lines of code:

```
*pos = '\0';
if (out_len)
    *out_len = pos - out;
return out;
```

These lines terminate the output string, store the number of bytes written, if any were written, and return the pointer to allocated memory.

So, there were quite a few things in that code I wanted to change. First, I didn't want to do any allocation! In embedded code for small microcontrollers, I usually try to write my code such that all buffers are allocated at startup and never

dynamically, at runtime, for two reasons — first, I don't want to include the overhead, both in flash storage space and in runtime, of the standard library code that manages the heap, and second, there's no good way to handle an error condition in an embedded device in which a memory allocation operation failed. It's best to make it so it can't fail! And second, I didn't want to add any line breaks, or any padding characters. Adding line breaks would mean extra storage, and I have to watch every byte. But worse, they would complicate the simple size calculations. And they would also mean I'd have to either filter them out before passing data to my decoding function, or complicate the decoding function by adding in logic to skip them, which means more code.

It was starting to look like it would be almost as much work to modify this code as to write something new. So I decided to write something new.

Writing My Own Implementation

My first step was to write the code to turn six-bit values into their corresponding ASCII characters. I took that on first because it was very simple. In the BSD code, they use an array to do table lookup. I took that approach. My embedded toolchain will place this in flash memory, which saves valuable RAM:

```
static const uint8_t to_base_64_a[ENCODING_TABLE_SIZE] = {
    'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
    'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',
    'Q', 'R', 'S', 'T', 'U', 'V', 'W' ,'X',
    'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f',
    'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n',
    'o', 'p', 'q', 'r', 's', 't', 'u', 'v',
    'w', 'x', 'y', 'z', '0', '1', '2', '3',
    '4', '5', '6', '7', '8', '9', '-', '_'  };
```

Our **ENCODING_TABLE_SIZE** is 64 bytes. A zero byte becomes the character 'A,' a 1 byte becomes 'B,' etc.

While I was at it, I wanted to write code to let me do the reverse lookup, translating ASCII characters from our 64-character Base64 character set back to bytes in the range 0..63. But the reverse is not so simple, because our ASCII characters are not *contiguous*. The dash is 45 in ASCII, the numerals are 48 to 57, the uppercase letters are 65 to 90, the underscore is 95, and the lowercase letters are 97 to 122. So I either have to use a much larger lookup table, or use code like so:

```
    if ((byte_in >= 'A') && (byte_in <= 'Z'))
    {
        /*
            Values 0..25
        */
        byte_out = byte_in - 'A';
```

```
    }
    else if ((byte_in >= 'a') && (byte_in <= 'z'))
    {
        /*
            Values 26..51
        */
        byte_out = byte_in - 'a' + 26;
    }
    else ...
```

etc., with three more special cases. That means there is a decision to make — do I want to use more *code*, or more *data*? I decided to write it both ways, so that I could test both options, to see which one used less precious flash memory.

Before I even started to write the encoding function, I wrote a test function, so that I could call the test function and use it to verify that all the test cases worked right. Here is part of that test function:

```
uint8_t source_bytes_1[1] = { 0xA5 };
uint8_t source_bytes_2[2] = { 0xA5, 0xC3 };
uint8_t source_bytes_3[3] = { 0xA5, 0xC3, 0x3C };
uint8_t source_bytes_4[4] = { 0xA5, 0xC3, 0x3C, 0x5A };

uint8_t dest_bytes_0 [1] = { 0xFF };
uint8_t dest_bytes_1 [3] = { 0xFF, 0xFF, 0xFF };
uint8_t dest_bytes_2 [4] = { 0xFF, 0xFF, 0xFF, 0xFF };
uint8_t dest_bytes_3 [5] = { 0xFF, 0xFF, 0xFF, 0xFF, 0xFF };
uint8_t dest_bytes_4 [7] = { 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF };

uint8_t out_byte_count;

Encode_Base_64(source_bytes_1, dest_bytes_0, 0, &out_byte_count);
Encode_Base_64(source_bytes_1, dest_bytes_1, 1, &out_byte_count);
Encode_Base_64(source_bytes_2, dest_bytes_2, 2, &out_byte_count);
Encode_Base_64(source_bytes_3, dest_bytes_3, 3, &out_byte_count);
Encode_Base_64(source_bytes_4, dest_bytes_4, 4, &out_byte_count);
```

This line:

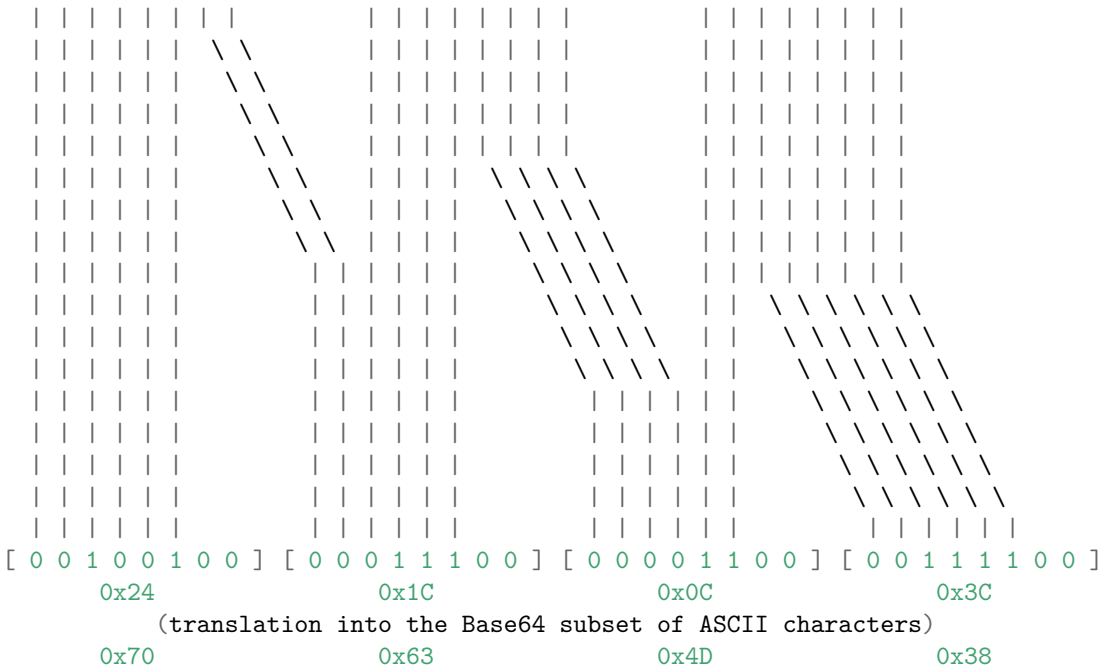
```
Encode_Base_64(source_bytes_3, dest_bytes_3, 3, &out_byte_count);
```

sends the address of our **source_bytes_3** array,

```
{ 0xA5, 0xC3, 0x3C }
```

to our (still hypothetical) **Base_64_Encode** function. Here's what I want to happen:

```
       0xA5                    0xC3                    0x3C
[ 1 0 1 0 0 1 0 1 ]     [ 1 1 0 0 0 0 1 1 ]     [ 0 0 1 1 1 1 0 0 ]
```



When we generate those four binary byte values, we want to convert them into their corresponding ASCII character values. You can look them up in the lookup table above, or you can just take my word for it that the resulting values are 0x70, 0x63, 0x4D, and 0x38.

Note that there's a "sentinel" byte containing 0xFF immediately after the fourth byte in our output buffer, so that I can verify it isn't changed. I do this to reassure myself that the code didn't write more bytes to the output buffer than it was supposed to.

My test function has code that checks that the output buffer always looks exactly as expected; it checks each byte, including the "sentinel" bytes at the end, to make sure they weren't inadvertently overwritten. It's not really worth showing you *all* that code, because it is quite redundant, but here's the line that checks to see if what we wanted to happen actually happened:

```
if ((dest_bytes_3[0] != 0x70) || (dest_bytes_3[1] != 0x63) ||
    (dest_bytes_3[2] != 0x4D) || (dest_bytes_3[3] != 0x38) ||
    (dest_bytes_3[4] != 0xFF)                                 )
{
    return 0;
}
```

Writing the Encoding Function

I worked through the operations on paper, figuring out what combination of bitwise **and** operations, to extract bits, shift operations, and bitwise **or** operations to combine bits back together, were needed to shuffle the bits as shown in the diagram. After working that out, I was ready to ttry writing my own encoding function. Here's what I came up with:

```
void Base_64_Encode(uint8_t * in_p, uint8_t * out_p, uint8_t in_byte_count,
                    uint8_t * out_byte_count_p)
{
    uint8_t in_byte_idx = 0;
    uint8_t out_byte_count = 0;

    while (in_byte_idx < in_byte_count)
    {
        uint8_t remaining_bytes = in_byte_count - in_byte_idx;

        if (1 == remaining_bytes)
        {
            out_p[0] = base64_a[  in_p[0]          >> 2                ];
            out_p[1] = base64_a[ (in_p[0] & 0x03 ) << 4                ];
            out_byte_count += 2;
            break;
        }
        else if ( 2 == remaining_bytes )
        {
            out_p[0] = base64_a[  in_p[0]         >> 2                 ];
            out_p[1] = base64_a[((in_p[0] & 0x03) << 4) | (in_p[1] >> 4)];
            out_p[2] = base64_a[((in_p[1] & 0x0F) << 2)                ];
            out_byte_count += 3;
            break;
        }
        else
        {
            out_p[0] = base64_a[  in_p[0]         >> 2                 ];
            out_p[1] = base64_a[((in_p[0] & 0x03) << 4) | (in_p[1] >> 4)];
            out_p[2] = base64_a[((in_p[1] & 0x0F) << 2) | (in_p[2] >> 6)];
            out_p[3] = base64_a[  in_p[2] & 0x3F                       ];
            in_byte_idx += 3;
            out_byte_count += 4;
            out_p += 4;
        }

    }
```

```
    *out_byte_count_p = out_byte_count;
}
```

Improving the Encoding Function

For my first attempt, I tried to make the function as simple as possible. And so, it worked correctly on the first try. But I wasn't really happy with it. It only supported the use of the lookup table for encoding, and it just seemed like it had too many lines of code, and in particular, too many repeated, or nearly exactly repeated, expressions. There are six **&** operators and eleven shift operators. It just looked like I ought to be able to make it smaller and simpler.

So I experimented with different ways to shrink it down. I made some minor changes, but they didn't really result in a lot of improvement.

So I let that version percolate in my brain over a weekend, and then came back to it with a different approach.

First, I used a macro, **TO_BASE_64**, that allowed me to easily change the strategy I was using to generate the ASCII characters.

And second, I knew that I could simplify the code using a **switch**, implementing it in a way that looked something like Duff's Device, using fall-through so that I executed additional chunks of code only when necessary.

The key to making that work was to use a **do** loop with a switch inside it, and reverse the order: handle 3 or more input bytes first, then 2, then 1, then none; and also, reverse the order of writes into the output data within each case.

And then, finally, I found I could also do a neat trick with **remaining_in_bytes**, a sort of "lookahead" operation to optionally "or" in the bits I needed from the *next* input byte, which meant that if there isn't a next input byte, the bits are zeroes. This "lookahead" allowed me to conditionally add in those bits if we had "fallen through" from the previous case, without setting and checking a separate flag variable.

Of course, I didn't get there all at once. I tried a few experiments and revised the code several times before finally writing the main loop like this:

```
do
    {
        remaining_in_bytes = num_in_bytes - in_byte_idx;

        switch( remaining_in_bytes )
        {
            default:
                out_p[3] = TO_BASE_64(     in_p[in_byte_idx + 2] & 0x3F             );
                FALLTHROUGH

            case 2:
```

```
                out_p[2] = TO_BASE_64(((in_p[in_byte_idx + 1] & 0x0F ) << 2 ) |
                    ((remaining_in_bytes > 2) ? (in_p[in_byte_idx + 2] >> 6) : 0 ) );
                FALLTHROUGH

            case 1:
                out_p[1] = TO_BASE_64(((in_p[in_byte_idx    ] & 0x03 ) << 4 ) |
                    ((remaining_in_bytes > 1) ? (in_p[in_byte_idx + 1] >> 4) : 0 ) );
                out_p[0] = TO_BASE_64(  in_p[in_byte_idx    ]          >> 2     );
                in_byte_idx += 3;
                out_p += 4;
                FALLTHROUGH

            case 0:
                break;
        }

    } while (remaining_in_bytes > 3);

    *num_bytes_written_p = encoded_size;
}
```

I am pretty happy with that, because it seems to me that it expresses what is happening in the diagram above with less redundancy, and does as little extra work as possible. There are now six **&** operators and five shift operators, although some folks might complain that the question-mark operator is confusing. I don't mind it, though, and use it frequently, because it is really just a way of writing an **if** expression that returns a value, and that's very useful.

FALLTHROUGH, by the way, is a macro. I want the compiler to not warn me about the fact that control "falls through" between the different cases. Automatic falll-through is a common source of errors in C programs. Most of the time, you **do** want the code to stop after executing one of the cases, and not continue executing into the next case. But this time I wanted it to fall through. So **FALLTHROUGH** becomes "whatever it takes to shut the compiler warning up, if it can be shut up." It also serves as an indication to the person reading the code that I intended to do it that way.

That isn't the whole function. It has "pre-flight" error checking. You give it the output buffer size, and it will confirm that it can do what you've asked, or return an error. If it succeeds, it tells you how many bytes it wrote.

If I want to pad the data, I can use another function I wrote which calls this function, and then writes the padding bytes, after first checking that it has enough additional room for them. That way, I only pay for the overhead of adding the padding bytes if I want padding.

Writing the Decoding Function

Something very nice happened after writing that function. It allowed me to easily turn it "inside-out" to write the corresponding decode function, which clearly looks like the reverse algorithm. The core of it is very elegant — even simpler-looking than the core of the encoding function. It doesn't have those conditional expressions. Here's the core logic:

```
do
{
    remaining_out_bytes = plaintext_size - out_byte_idx;

    switch( remaining_out_bytes )
    {
        default:
            out_p[out_byte_idx + 2] =
                (uint8_t)( (FROM_BASE_64( in_p[2])         << 6) |
                           (FROM_BASE_64( in_p[3]) & 0x3F      )   );
            FALLTHROUGH

        case 2:
            out_p[out_byte_idx + 1] =
                (uint8_t)( (FROM_BASE_64( in_p[1])         << 4) |
                          ((FROM_BASE_64( in_p[2]) & 0x3C) >> 2)   );
            FALLTHROUGH

        case 1:
            out_p[out_byte_idx    ] =
                (uint8_t)( (FROM_BASE_64( in_p[0])         << 2) |
                          ((FROM_BASE_64( in_p[1]) & 0x30) >> 4)   );
            out_byte_idx += 3;
            in_p += 4;
            FALLTHROUGH

        case 0:
            break;
    }

} while (remaining_out_bytes > 3);
```

I like that.

That seemed pretty good, so I left the code alone for a few weeks, and worked on other things.

Writing Code for People to Read

But it was nagging me a bit.

In his landmark book *Structure and Interpretation of Computer Programs*, Harold Abelson wrote:

> Programs must be written for people to read, and only incidentally for machines to execute.

And I have always tried to write code that embodies that aphorism. I try to make simple things look simple, and complicated things also look simple. The encoding algorithm isn't *really* very complicated, as these things go, so I think it is reasonable to wonder if expressions like

```
out_p[1] = TO_BASE_64(((in_p[in_byte_idx] & 0x03) << 4) |
    ((remaining_in_bytes > 1) ? (in_p[in_byte_idx + 1] >> 4) : 0 ) );
```

are really the best way to express it.

For one thing, although I like to use whitespace to line up the parts of multi-line expressions, in order to make the grouping and structure visible, those conditional expressions make it ugly and hide some of that nice alignment of parallel parts of the expressions. It helps if we unfold the long lines, as shown in this fragment:

```
out_p[1] = TO_BASE_64(((in_p[in_byte_idx] & 0x03) << 4) | ((remaining_in_bytes > 1) ? (in_p
out_p[0] = TO_BASE_64(  in_p[in_byte_idx]          >> 2
```

If you're reading this as a PDF file, the lines above probably march right off the edge of the page (Sorry about that; I'm trying to make a point.) I've seen some textooks that include pages of code in landscape mode. That works but it makes them quite awkward to read, and in any case I don't know how to do that when using Markdown as a source format, although it probably can be done.

Monitors are much bigger than they were in the seventies, eighties, and nineties, and so I don't think we should feel like we need to stick to the width of old punch cards or green-screen terminals. These days I use a maximum line length of 150 characters when I'm writing C code, and a limit of 120 characters seems to be common in my industry now. But if I want to make a readable printed copy of code that is up to 150 characters wide, to review offline, without making the typeface very tiny and hard for my middle-aged eyes to read, I have to either print in landscape mode, which makes for fewer lines of code per page and wasted paper, or print on 11x17 (tabloid-size) pages. And I don't have a printer at home that will handle tabloid-size pages; they are expensive and large.

I guess one approach is just to say "I'm never going to print my code on paper," and "PDF doesn't matter," but I'm not satisfied with those answers.

But all that aside, is the wider code *clearer*? That's the real goal.

Expressing Conditionals

The conditional part, especially, is a little tricky. I like using the question mark operator, since it's the only way in C to create conditional code that returns a

value – something commonplace now in other languages – but it is easy to screw up. For example, if one is looking at an expression like this:

```
result = bits | ((condition) ? (true expression) : (false expression));
```

One might be tempted to remove some of the parentheses, because they appear to be useless:

```
result = bits | (condition ) ? (true expression) : (false expression);
```

But this would be a mistake. The C language operator precedence will come to bite you, and you'll get this behavior:

```
result = (bits | (condition)) ? (true expression) : (false expression);
```

(Ask me how I know that!)

Should I give up the question mark operator? Some coding standards recommend doing just that, but it would take away what I think is a very useful idiom.

My Personal Coding Style

Another aside: my personal C coding style, developed over decades, includes a lot of parentheses that some C programmers would sneer at. I just still find it easier and clearer to include extra sets of parentheses that are technically redundant, because they express exactly what I want the code to do, allowing me to focus on the algorithm as I'm writing the code rather than the arcane rules of C operator precedence. I don't do this in languages such as Haskell, where the operator precedence rules were a bit better thought-out from the start. Even one of the creators of C has admitted that it would have been better if some of the rules had been different:

> In retrospect it would have been better to go ahead and change the precedence of & to higher than ==, but it seemed safer just to split & and && without moving & past an existing operator. (After all, we had several hundred kilobytes of source code, and maybe 3 installations….)

But it is far too late now; C is what it is, and since what it is is often confusing, I use parentheses.

I try not to make it my highest priority, unless I see a demonstrated need to optimize the source code, but there's also a question of efficiency. We may be doing the conditional checks inside the assigments, instead of outside them, but they are still there. In the source code, they appear multiple times, when processing even a case that should be the simplest. The compiler may be able to do very clever things and optimize some of this away, but I have not profiled this code or looked at the resulting assembly language to check if that is happening. And, of course, optimization is specific to the platform, compiler, and compiler settings.

Looping BSD-Style

What if I just bit the bullet and did the looping the way the BSD encoding funtion did it, trying to first encode as many full groups as possible, then handle any leftover input bytes? Well, I still have all my testing apparatus in place, and everything is in version control, so it's very easy to try out a code change. I'm not really *happy* to throw out my "look how clever I am!" version, but it may be for the best. So here's the "look how simple I can be!" version instead, with more comments to explain things:

```
while ((num_in_bytes - in_byte_idx) >= 3)
{
    /*
        Produce all four output bytes following the
        three-to-four pattern.
    */
    out_p[0] = TO_BASE_64(  in_p[in_byte_idx    ]          >> 2    );
    out_p[1] = TO_BASE_64(((in_p[in_byte_idx    ] & 0x03) << 4) |
                          ( in_p[in_byte_idx + 1]          >> 4)   );
    out_p[2] = TO_BASE_64(((in_p[in_byte_idx + 1] & 0x0F) << 2) |
                          ( in_p[in_byte_idx + 2]          >> 6)   );
    out_p[3] = TO_BASE_64(  in_p[in_byte_idx + 2] & 0x3F           );
    in_byte_idx += 3;
    out_p += 4;
}

if ((num_in_bytes - in_byte_idx) > 0)
{
    /*
        There is least one remaining input byte: produce the first
        output byte.
    */
    out_p[0] = TO_BASE_64(  in_p[in_byte_idx    ]          >> 2    );

    if (1 == (num_in_bytes - in_byte_idx))
    {
        /*
            There is exactly one more input byte. Produce the
            second output byte from bits in the first input byte.
        */
        out_p[1] = TO_BASE_64(((in_p[in_byte_idx    ] & 0x03) << 4)   );
    }
    else
    {
        /*
            There are exactly two more input bytes. Produce the
```

```
            second output byte with bits from both the first and
            second input byte, and the third output byte from the
            remaining bits in the second input byte.
        */
        out_p[1] = TO_BASE_64(((in_p[in_byte_idx    ] & 0x03) << 4) |
                              (in_p[in_byte_idx + 1]         >> 4)   );
        out_p[2] = TO_BASE_64(((in_p[in_byte_idx + 1] & 0x0F) << 2)   );
    }
}
```

OK. All my test "scaffolding" was still in place, and the above code passed all the tests. So now I just have to ask myself: is this better in any signficant way than my first attempt?

```
while (in_byte_idx < in_byte_count)
{
    uint8_t remaining_bytes = in_byte_count - in_byte_idx;

    if (1 == remaining_bytes)
    {
        out_p[0] = base64_a[  in_p[0]          >> 2                ];
        out_p[1] = base64_a[ (in_p[0] & 0x03 ) << 4                ];
        out_byte_count += 2;
        break;
    }
    else if ( 2 == remaining_bytes )
    {
        out_p[0] = base64_a[  in_p[0]         >> 2                 ];
        out_p[1] = base64_a[((in_p[0] & 0x03) << 4) | (in_p[1] >> 4)];
        out_p[2] = base64_a[((in_p[1] & 0x0F) << 2)                ];
        out_byte_count += 3;
        break;
    }
    else
    {
        out_p[0] = base64_a[  in_p[0]         >> 2                 ];
        out_p[1] = base64_a[((in_p[0] & 0x03) << 4) | (in_p[1] >> 4)];
        out_p[2] = base64_a[((in_p[1] & 0x0F) << 2) | (in_p[2] >> 6)];
        out_p[3] = base64_a[  in_p[2] & 0x3F                       ];
        in_byte_idx += 3;
        out_byte_count += 4;
        out_p += 4;
    }

}
```

Thinking it over, I concluded that it is a bit better. I like the way it uses a simple

while loop to handle what, when working with a variety of different data lengths, will probably be the most commonly executed case first: producing full chunks of 4 output bytes. There is less logic happening in that loop, at the expense of a bit more logic in the "mopping up" branches. But we will only actually execute *one* of the "mopping-up" branches per call to the encoding function. And my comments seem to explain it reasonably well. So I'm going to keep it.

So, I wound up using some of the structure of the BSD code after all. I thought about using other ways to structure the "leftover" logic, but despite my experimentation, I really couldn't think of a way that was better than the way the BSD code implements that core algorithm.

Debrief

So what was the point of all this effort?

Well, for one thing, I now understand the algorithm very well. And I have convinced myself that while there are some more clever ways to implement it, I can't think of one that is obviously *better*. And I now have a thoroughly-tested implementation of Base64 encoding and decoding functions, and a suite of test code. And my versions do exactly what I want them to do, and no more. I find that time spent *practicing* implementation of algorithms is never actually wasted.

My version also generate more detailed errors than the BSD code. For example, I have a separate error code that my decoding function returns if the length of the encoded data is not sensible, as per my earlier discussion of impossible output data lengths.

I also think my decoding function is a big improvement over the BSD version, at least for my purposes. In that function, the switch with fall-through logic is more elegant than it was in the encoding function. So I'm keeping that one the way it is.

Land of Revision

And so, that's an example of how I think and how I write code. In real life, it's much messier than this. I committed twelve different versions of my C file to the version control system, and eight different versions of the header file. I covered a number of sketchbook pages with scribbled notes. I went down some dead ends, and backed stuff out. I ran code under the debugger many times.

My process is unavoidably messy like this because it relies on *revision*. Traditionally, *that isn't't how the discipline of software engineering expected code to be written*. But it's the only way that has ever really worked for me, and I think an awful lot of programmers will, if they are honest, admit that they like to work this way, too; they don't *actually* get the best results by thinking about the problem, writing the one and only true version, maybe giving it a quick test (or maybe not!), and then walking away from it.

Wow, that took me a long time to write. And I wound up revising my code, which I thought was finished, several more times while I wrote it. In particular, I improved my comments, which I thought I was completely done with. But fortunately because I had all that test scaffolding, it was very easy to test my changes. And so the process of *explaining* my code resulted in improvements. Which is what usually happens, if one is humble enough to keep realizing that there is room for improvement, and allow the process to do its magic.
