



BarcodeXpress

# Refactoring Legacy Code for Speed in Barcode Xpress



**AUTHOR**

John Reynolds

Principal Software  
Engineer



---

# Introduction



For a recent hackathon here at Accusoft, I had the idea of going through some ancient legacy code for Barcode Xpress and rewriting some particularly dense and confusing code into a more understandable form. In the process, both speed and accuracy were greatly improved. Let's give a little background on the project itself.

Barcode Xpress is an SDK that detects and reads barcodes on images, whether they be older-style 1D barcodes (eg. UPC, Code128, etc) or newer 2D barcodes (eg. DataMatrix, QR Codes, etc). It's been around for a long time; I've been a developer at Accusoft for almost 13 years now and it was still a mature product when I joined. Much of the oldest code, that which handles 1D code, was developed by people who long ago retired and was written in an older style of C programming, the C89 standard. Similar to much old legacy code, there were functions in the thousands of lines of code, with copy/pasted content spread throughout. The code worked great though, so for a long time, we hadn't looked too deep into it because, after all, "if it ain't broke, don't fix it."

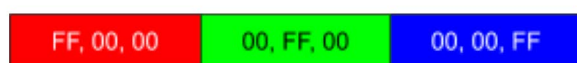
Hackathons make for a golden opportunity to experiment, free of consequence, so I lept at the chance to try and make the code more readable. The changes were initially just to use more smaller, encapsulated functions, locally scoped variables, and other things we typically associate with modern programming paradigms, designed to reduce developer cognitive load. In the process though, some observations about the functionality of the code allowed me to make some pretty big speed improvements.

I'll go through the process of how I changed the code, first by going over how image data is laid out in memory, how the pixels are accessed, and how the existing code was modified to achieve both readability and performance improvements.



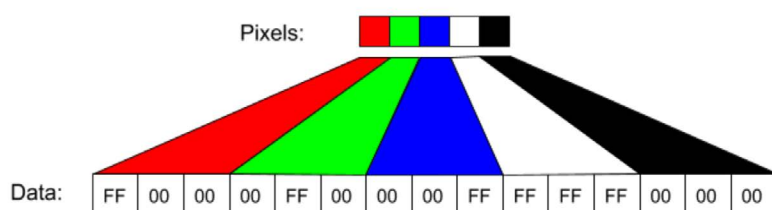
# Image Data in Memory

Normal color images seen everywhere on the net are typically represented in memory as either 24 bpp (bits-per-pixel) or sometimes 32 bpp data buffers. This corresponds to RGB (red, green, blue) data in the case of 24 bpp images and RGBA (red, green, blue, alpha) for 32 bpp images. Let's consider 24 bpp RGB images. Each color pixel on the image will contain an 8-bit value representing the redness of the pixel, 8 bits for green and 8 bits for blue.



**Fig 1**  
Three pixels – red (255, 0, 0), green (0, 255, 0), and blue (0, 0, 255)

In memory, the buffer for this image will be a sequence of bytes concatenating the pixels together row-wise. Using hexadecimal notation, an example image row might look like this:



**Fig 2**  
The memory representation of five 24-bpp color pixels

*Note: In some bitmap memory formats, rows are always padded to be equally divisible by 4 bytes. This has already been incorporated into the `bytes_per_row` variable ahead, but it is something to keep in mind when doing work on images in memory. If you're looking at an image in memory and your results show it severely skewed, vertically, then it's likely you have a stride vs. width mismatch.*

If I want to find the color information of a pixel with an (x, y) coordinate, I might perform the following calculation:

```
const byte *pixel_data = (const byte*)
image.data + y*image.bytes_per_row + x*3;
const byte red = pixel_data[0];
const byte green = pixel_data[1];
const byte blue = pixel_data[2];
```

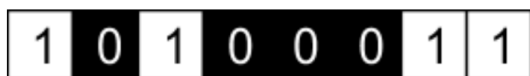
This is pretty simple, because the pixel information is nicely partitioned into byte sized bunches of information and only direct indexing is needed.

When analyzing a 1D barcode though, we need to measure the transitions between white and black columns of pixels — we don't need all of this color information. While we do have some grayscale processing in other modules, for this code in particular, we only need black and white color information. Thus, we use 1bpp images internally. Let's look at how these are organized.

# Black and White Images

Because there is only 1 bit of information for each pixel, this means we can store 8 pixels of information in every byte of memory. This is great, and very compact, but it makes reading the information an exercise in bit shifts and masking.

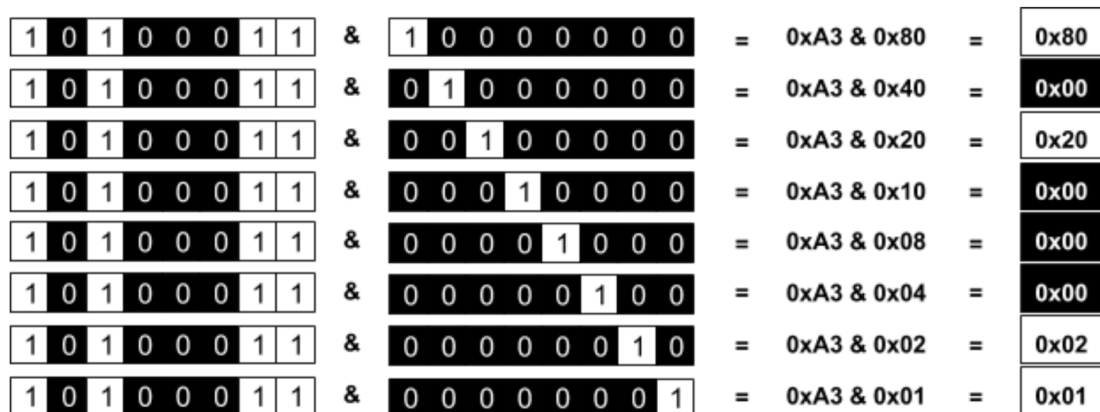
If we were to store the pattern of WBWBWW in a single byte of memory, this would be represented by the binary data 10100011. In hexadecimal, this is the value 0xA3.



**Fig 3**

*Eight pixels fit inside one byte of information.*

To access each pixel, we use something called masking, using the bitwise AND operation, to look at a particular bit of the byte. For example, to look at each of the bits of the value 0xA3, we could do the following:



**Fig 4**

*How we might check each of the bit locations of our eight pixels in the byte.*

As you can see, we get positive values when that pixel is white, and a zero value when the pixel is black. Some code to determine if a pixel at the coordinate (x,y) is black or not, we might write some code that looks similar to the following, simplified:

```
const byte *pixels_data = (const byte*)image.data + y*image.
bytes_per_row + x/8;

// calculate the mask, the first pixel is 0x80, the second
0x40, etc.
const byte mask = 1 << (7 - (x % 8));

const bool is_pixel_black = (*pixels_data & mask) == 0;
```

Thus, for each pixel, you need the coordinates, the image bitmap buffer, and then can calculate the mask.

00101000  
00101000  
00101000



## Existing Code

At the most-reduced level, some of the functionality that I was simplifying can be boiled down to the following statement: Given a particular scanning direction, count the length of black / white runs in a direction.



**Fig 5**

*Example lines in a 1D barcode*

Because repeatedly calculating the mask and data pointer for every coordinate is cumbersome, if we do it every time, we can do some shortcuts. For example, we can keep a running tally of our mask as we progress across the image. On an ultra-simplified level, the code looked like this:

```
void find_traces(const Bitmap &image, const int direction) {
    // Initialize the running information
    int x = 0, y = 0;
    const byte *pixel_data = (const byte*)image.data;
    byte mask = 0x80;

    // Keep scanning until everything's done
    while (is_more_to_scan()) {
        const bool is_cur_pixel_black = (*pixel_data & mask) == 0;

        // Process this current pixel
        // ...

        switch(direction) {
            // Increment by one pixel in the horizontal direction
            case DIRECTION_HORIZONTAL:
                x++;
                mask >>= 1;
                // If we reach zero, we've transitioned byte boundaries
                // so reset the mask
                if (mask == 0) {
                    mask = 0x80;
                    pixel_data++;
                }

                // handle wrap-around if we've reached the row border
                // ...
                break;

            // Increment by one pixel in the vertical direction
            case DIRECTION_VERTICAL:
                y++;
                pixel_data += image.bytes_per_row;

                // handle wrap-around if we've reached the column edge
                // ...
                break;

            // Diagonal cases
            // ...
        }
    }
}
```

Phew, that's a lot of code that deals with the `pixel_data`, `mask`, `coordinates`, etc. What's shown above is also a drastic simplification — in reality, there were hundreds of lines just handling wrapping behavior, transferring mask data in and out of functions, etc. I needed a way to clean this up.

# DirectionalBuffer

The operations we're doing on the image at this state are effectively the following: initialize buffer, location, and mask, query if we're at a black pixel, increment in a direction, and wrap if we reach the border. We can pull in all of this code into a simple class such that we don't have to worry about keeping track of each variable in tandem.

Here's how a first draft of that class might look:

```
class DirectionalBuffer {
    const Bitmap &image;
    byte *pixel_data;
    Point location;
    byte mask;

public:
    DirectionalBuffer(const Bitmap &_image, const Point &point)
        : image(_image)
    {
        this->location = point;
        this->pixel_data = (byte*)image.data + point.y * image.bytes_per_row + point.x / 8;
        this->mask = 1 << (7 - (point.x % 8));
        // alternatively: 1 << (7 - (point.x & 7))
    }

    void increment(const int direction) {
        switch(direction) {
            case DIRECTION_HORIZONTAL:
                this->location.x++;
                this->mask >>= 1;
                if (this->mask == 0) {
                    this->mask = 0x80;
                    this->pixel_data++;
                }
                break;

            case DIRECTION_VERTICAL:
                this->location.y++;
                this->pixel_data += this->image.bytes_per_row;
                break;

            // further cases omitted for brevity
        }
    }

    bool is_pixel_black() {
        return (*(this->pixel_data) & this->mask) == 0;
    }
};
```



This is great! Now our loop code simplifies greatly and we have fewer things to keep track of:

```
void find_traces(const Bitmap &image, const int direction) {
    DirectionalBuffer directional_buffer(image, { 0, 0 });

    // Keep scanning until everything's done
    while (is_more_to_scan()) {
        bool is_cur_pixel_black = directional_buffer.is_pixel_black();

        // Process this current pixel
        // ...

        // move in the desired direction
        directional_buffer.increment(direction);
    }
}
```

As you can see, this cleans up the code pretty nicely. Now we can finally get to performance improvements, now that we've made the code more modular.

# Templating Out the Direction

One of the things I noticed when profiling this code was that there were huge sections of code that were skipped over due to the different code handling directions, and that in hot code loops, we kept doing the switch statements on values that never changed. Granted, the CPUs branch predictor would quickly determine where flow goes, but I wanted to minimize the amount of branching code in the function, so I decided to take a drastic step and pull the direction parameter out into a function template.

**An example of how the code might look with this, is as follows:**

```
template <int direction>
void find_traces(const Bitmap &image) {
    DirectionalBuffer directional_buffer(image, { 0, 0 });

    // Keep scanning until everything's done
    while (is_more_to_scan()) {
        const bool is_cur_pixel_black = db.is_pixel_black();

        // Process this current pixel
        // ...

        // move in the desired direction
        directional_buffer.increment<direction>();
    }
}

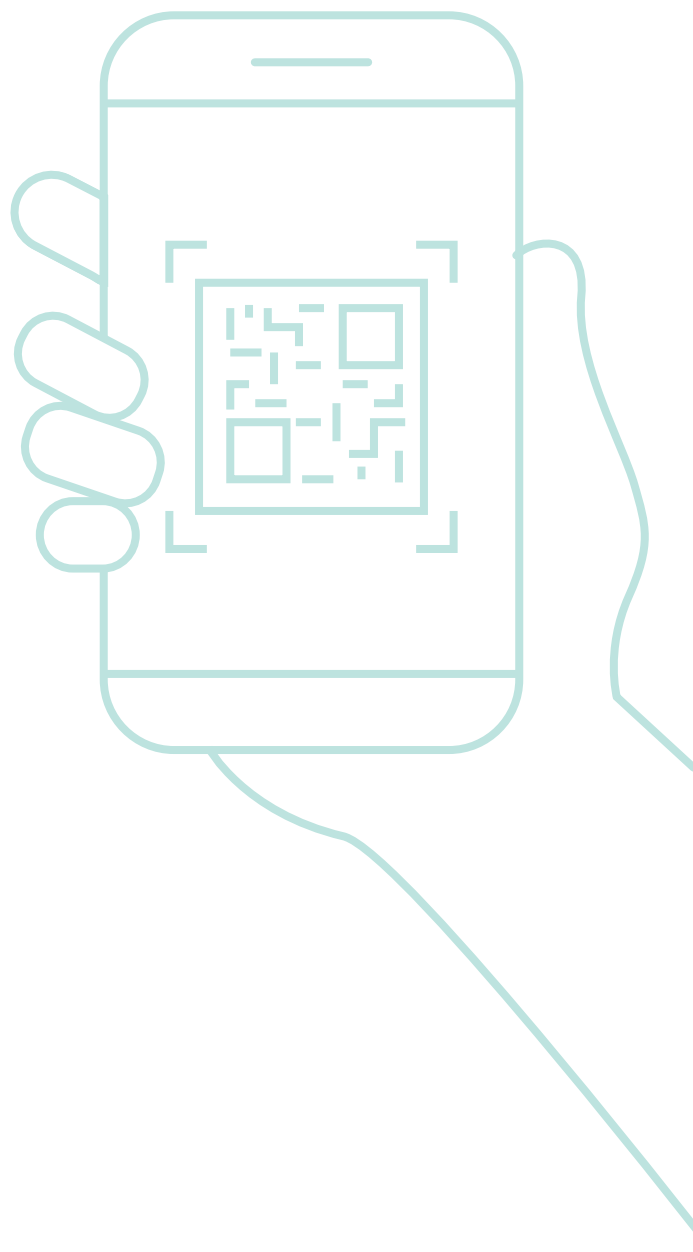
template void find_traces<DIRECTION_HORIZONTAL>(const Bitmap &image);
template void find_traces<DIRECTION_VERTICAL>(const Bitmap &image);
// ... similar for diagonal directions
```

**... and inside our DirectionalBuffer class:**

```
template <int direction>
void DirectionalBuffer::increment() {
    switch(direction) {
        case DIRECTION_HORIZONTAL:
            this->location.x++;
            this->mask >>= 1;
            if (this->mask == 0) {
                this->mask = 0x80;
                this->pixel_data++;
            }
            break;

        case DIRECTION_VERTICAL:
            this->location.y++;
            this->pixel_data += this->image.bytes_per_row;
            break;

        // further cases omitted for brevity
    }
}
```



This is a limited form of compile-time function execution, where variables are turned into compile-time constants. What does this give us? Well, any time we have large blocks of code that are dependent upon the direction value, the compiler will treat the direction as if it was a constant, and only the machine code for the particular direction we're working under will be generated.

For example, if we look at the compiled output for the increment function, we find the following generated assembly code for the two functions (code will vary depending upon compiler, source code, etc):

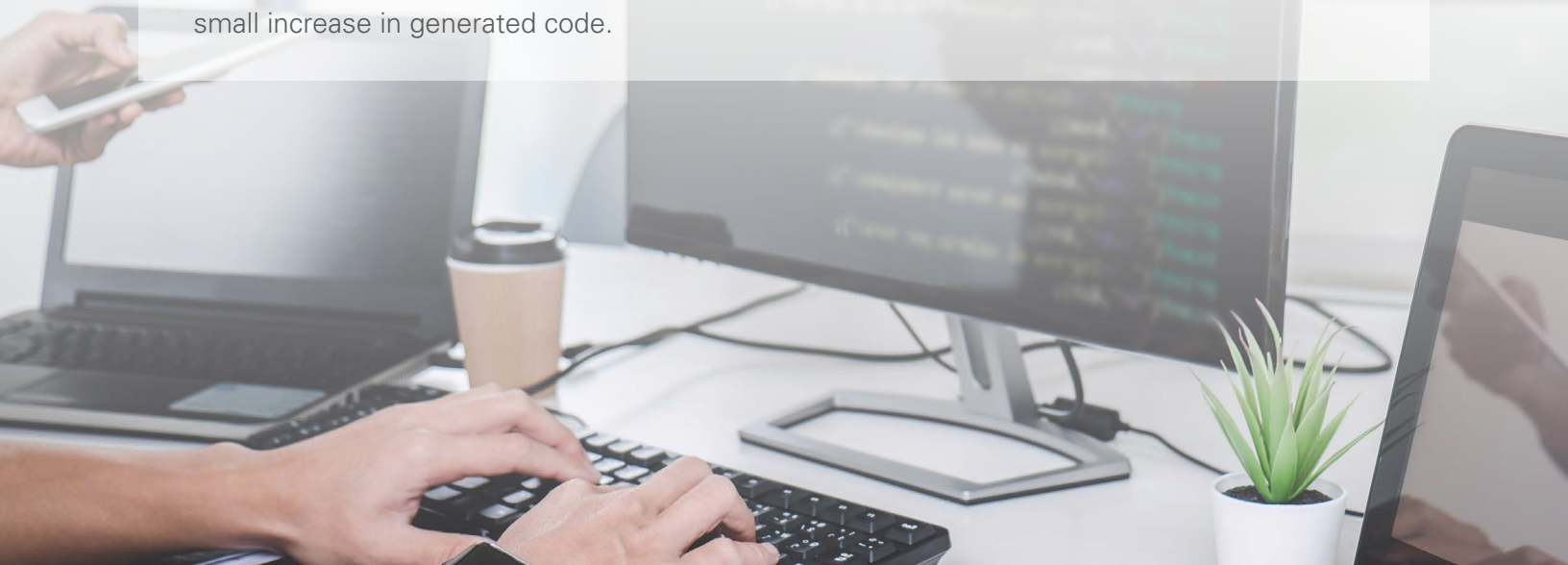
```

; void DirectionalBuffer::increment<DIRECTION_HORIZONTAL>()
  add    dword ptr [rdi + 16], 1    ; this->location.x++;
  mov    al, byte ptr [rdi + 24]   ;
  shr    al                        ;
  mov    byte ptr [rdi + 24], al   ; this->mask >>= 1;
  je     .LBB3_1                   ;
  ret                                     ; if (this->mask != 0) return;
.LBB3_1:
  mov    byte ptr [rdi + 24], -128 ; this->mask = 0x80
  add    qword ptr [rdi + 8], 1     ; this->pixel_data++;
  ret

; void DirectionalBuffer::increment<DIRECTION_VERTICAL>()
  add    dword ptr [rdi + 20], 1    ; this->location.y++;
  mov    rax, qword ptr [rdi]       ;
  movsxd rax, dword ptr [rax + 8]   ; this->pixel_data +=
  add    qword ptr [rdi + 8], rax   ; image.bytes_per_row;
  ret


```

As you can see, this generates smaller machine code for each function itself, which can help the inliner by having more room to work with. In fact, in practice, for the hot loop, the entirety of the DirectionalBuffer class was optimized and inlined into the parent function. The downside for this method is that the functions are generated multiple times in the executed, one for each direction. In this case, the benefits gained from inlined execution far outweighed the (relatively) small increase in generated code.









I did some profiling with skipping by 32 pixels by checking for the values `0xFFFFFFFF` or `0x00000000`, but this was surprisingly slower in aggregate than the single-byte skipping. Always profile your code changes! Additionally, if you really wanted to get deep into the weeds of micro-optimizations, additional bit-twiddling code or processor instructions could also be used to really crank up the speed.



---

## Results

Our 1D barcode processing code had always been really quick to analyze a full image, but this change improved our analysis times from up to 5% to ~60%, depending upon the image. In doing so, I also greatly simplified the code that scans our black and white traces, so development in this area of code is more possible in the future.

While in general, refactoring is something best done with a delicate touch, sometimes it can help expose previously hidden areas of potential performance wins.



---

[www.accusoft.com](http://www.accusoft.com)

