Family Photos vs 256 Kb RAM

2023-06-01

> *Discussion on HackerNews and Lobsters.*

A while ago I was thinking about making something nice for my wife as a present for the Valentine's Day. My eye caught a rather cool-looking 7-color eInk display with Raspberry Pico W on board - Pimoroni Inky Frame. From the first glance it looked quite polished with ready-to-use SDKs in Python and C++. I thought it would be interesting to make a dynamic family photo frame out of it, something that my wife could have on her table, with ability to switch photos from time to time with a press of a button.

I bought it on the same day. Oh boy, I did not know what I was getting myself into...

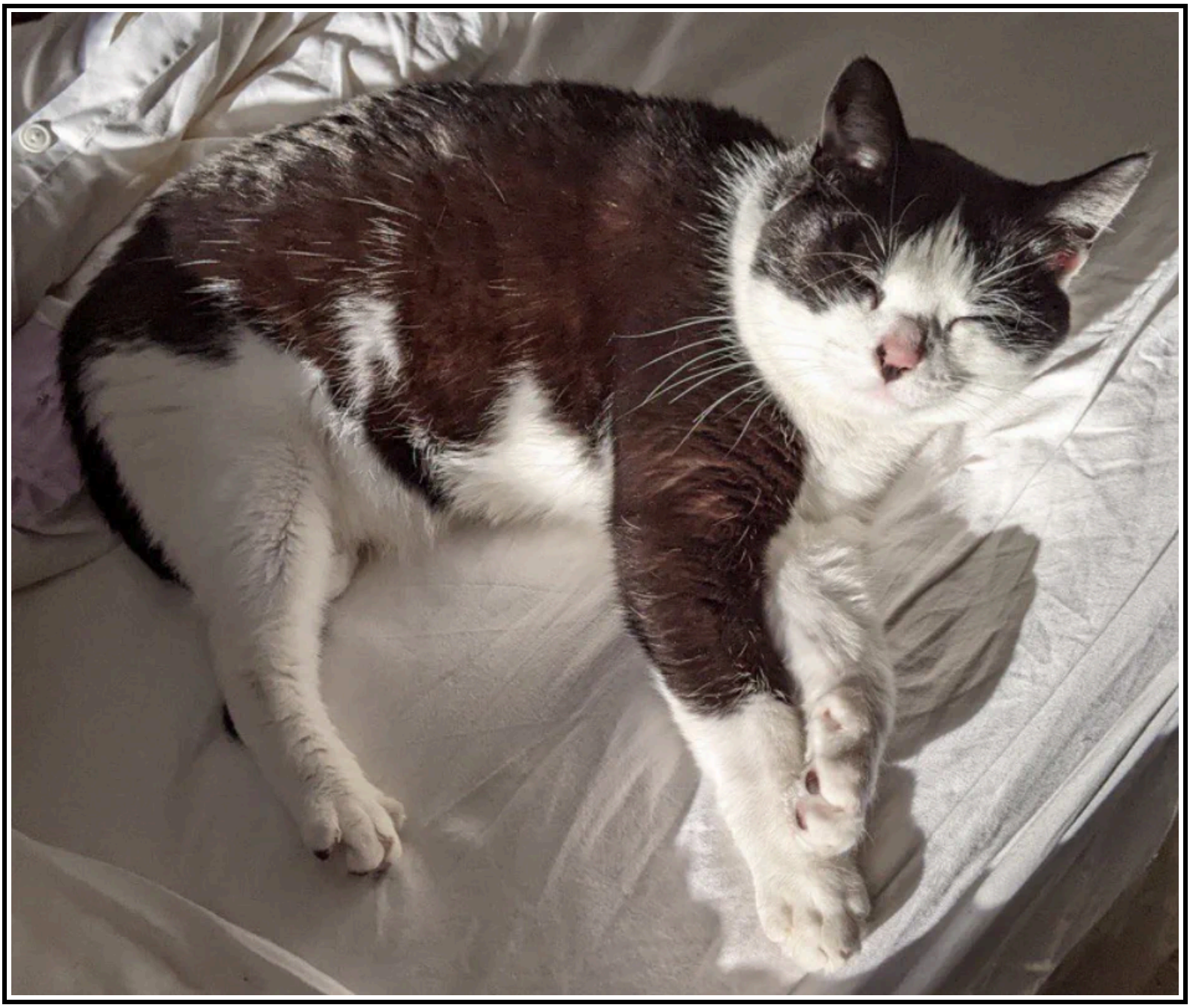
Disclaimer: This is not a promotional post and there are no affiliate links in this article. I actually hate this device quite a bit! Read along to find out why.

First impressions

> *Label on the packaging of Inky Frame. I think it looks fantastic! I have repurposed my box for other parcels, so this is from another buyer from the Pimoroni's website.*

First hour with the device was actually quite nice. The packaging was on point, the frame itself felt well-built with some pirate-let's-hack-this-together vibe. I liked it! However, for some reason, the battery pack did not work. This was not a big deal for me, because my wife's table has a built-in USB power, which I intended to use with the frame. Pimoroni did send me a replacement battery pack free of charge, but that did not work either. Probably there is some fault in the board itself, I am not sure and again, I did not really care at the time. The way the frame is powered turned out to be extremely important later, but I did not know that yet.

What do you mean a single photo is 3MB?



> *Chonky filesize for a chonky girl. This beauty is called Guinness, she is a lovely creature I had the pleasure to catsit for some time when I lived in Ireland.*

The frame itself includes an SD card slot. My original plan was to load all the photos onto it and read it into the Raspberry Pico W's RAM on demand. Pico then would convert the image into the 7-color pallete and transfer it onto the eInk screen. Pimoroni did a great job with their SDK - it is an absolute beast of a library. It already includes the logic for decoding JPEG images and implements a very decent dithering

from RBG color space into 7 colors of the eInk screen. There is even an example of an image gallery - problem solved, right?

Or so I thought. Example image gallery uses space images from the James Webb Space Telescope. There is an interesting property about these images apart from their beauty - they are mostly black, which allows them to compress pretty nicely. Some of them are only 8Kb, which comfortably fits into the 256Kb of RAM on Raspberry Pico W. Even after cutting for the size of the screen (600 x 448 pixels) some of my family photos were still at least 130Kb of JPEG. Which by all means should fit into the 256Kb of RAM, but it actually does not.

The thing is, the Pimoroni SDK itself does take a considerable portion of RAM. It provides an interface to “draw” on some internal buffer and then flush this buffer onto the screen. You cannot turn this buffer off, probably because writing the stream of bytes into the eInk screen directly is a very nich use case. From my experiments, we have around 130Kb of RAM to spare, which is too close to the photo size. My naive version coded in MicroPython ran of memory every half of the time I turned on the frame.

Hacking a solution close to a deadline

I think it was like 12th of Febraury when I discovered the problem with the image size. I needed a solution, and fast, so I came up with a simplest one I could think of. If a single image does not fit into RAM, let’s cut it into multiple smaller ones and read it one-by-one. The process of adding a photo to the frame turned into:

1. Open GIMP
2. Scale the image so that an interesting part of it fits 600 x 448 crop area
3. Cut the image to 600 x 448 size
4. Save it with the lowest possible quality
5. Run imagemagick incantation `convert input.jpeg -crop 75x56 +repage +adjoin output_%d.jpeg` to cut the image into 64 parts
6. Repeat

I knew this would bite me in the future, but at the time I just rolled with it. The resulting images did not have the size of `130Kb / 64`, unfortunately. The smaller the image, the less context the compression algorithm has, so the resulting parts turned out to be around 18Kb each.

The Raspberry Pico W then read all 64 images from the SD card sequentially (which took at least 10 seconds in total) and put each of them in the corresponding part of the buffer. It worked. It was dog slow, but it worked.

There was one thing I did not fully understand though. For some reason, none of the code examples from Pimoroni contained any loops. I was expecting to see at least one loop in the image gallery example, because you need to check periodically if any of the buttons were pressed and change the image if they were. Again, I did not give much care to it because of the deadline, and just wrote a busy loop checking for button inputs. I did not care for power consumption either because of USB power, so it was good enough. As it would turn out later, I dodged stepping into an enourmous rabbit whole of undocumented behaviour by doing this. I mean, of course I did step into it later down the line, but at least not before the deadline.

Finally, I presented the photo frame backed by a hacky solution to my wife and she loved it! It did bring me a lot of joy to pass by her table each day and see that she changed a photo to a different one.

After roughly a month, the inevitable happened. My wife told me that the frame stopped working.

Designing a proper solution

> *The back of the Inky Frame with Raspberry Pico W soldered onto it - a brutal, but honest solution.*

Instead of debugging the problem and optimizing memory usage of my MicroPython code, I have done the expected and decided to rewrite the whole thing in C++. This

was mainly because I do not have much experience with MicroPython and it is far easier for me to predict memory usage of C++ code. With MicroPython, I was calling `gc.collect()` everywhere in hope it would magically work.

I wanted to solve the problem of images randomly causing out-of-memory situations for good. For this to happen, I needed an image format which would have a stable size and still fit into the memory of Raspberry Pico W. The naive approach of just writing RGB values for all pixels clearly does not work. There are `600 * 480` pixels to describe and each of them requires 3 bytes - one for each of the RGB components. This results in `864Kb`, clearly exceeding Pico's capabilities. However, the eInk screen cannot display full RGB range - it only has 7 colors. What if we encoded our image using only those colors? 7 colors means we need only 3 bits to describe each pixel. In total we have `600 * 480 * 3 / 8` bytes, which is `108Kb`. Even with Pimoroni SDK's memory usage accounted, we would able to fit our image into RAM! A bonus to this solution is that we do not need to do any decoding on the Pico's side. We can just read raw bytes from the file and interpret them as the color indexes.

The obvious problem with this approach is that know we need to do the encoding for it. Cropping and rescaling the image is pretty simple, but to get from RGB into the 7-color space, we also need to implement dithering, which was done by Pimoroni SDK. Luckily, the dithering logic is relatively simple and can be easily extracted from the SDK code. The only thing I could not get right away were precomputed thresholds for dithering. I took a simple approach and just printed them to console from the Raspberry Pico W running a program that only does just that.

With all the necessary data for encoding images, I had two paths in front of me:

1. Write a Python script doing the encoding and call it after cropping/scaling the image in GIMP. This would work, but I will be the bottleneck for my wife to update photos on the frame
2. Create a web UI friendly to a non-technical person, which will do all the cropping, scaling and encoding

I decided to go with route (2), because it allows my wife to select new photos when she wants to, edit them in the UI and then simply copy them to the SD card.

This turned out to be simpler than I thought! All the heavy lifting was done by the cropper.js, which already implements drag-n-drop controls with crop and scale functionality. The only thing left for me to do was the encoding part, which was very simple to copy from C++ in Pimoroni SDK to vanilla Javascript. The resulting user flow is very straightforward: upload an image (to the browser, no data is sent to the server), edit it however you like and download the encoded image. I decided to publish it on my website for simplicity, so feel free to check it out here. Feel free to explore the dithering logic in the main.js file, it is just a few lines of code really doing all the work:

```
const canvas = cropper.getCroppedCanvas({
    width: 600,
    height: 448,
});
const rawPixels = canvas.getContext("2d").getImageData(0, 0, 600, 44
const inkyPixels = new Uint8Array(600 * 448 * 3 / 8);

function setBitByIndex(index, value) {
    const byteIndex = Math.floor(index / 8);
    const bitIndex = index % 8;
    inkyPixels[byteIndex] |= (value << bitIndex);
}

for (let y = 0; y < 448; y++) {
    for (let x = 0; x < 600; x++) {
        const r_index = y * 600 * 4 + x * 4;
        const r = rawPixels[r_index];
        const g = rawPixels[r_index + 1];
        const b = rawPixels[r_index + 2];

        // You do not ask what this is, I do not ask what this is,
        // everybody is happy.
        const cacheKey = ((r & 0xE0) << 1) | ((g & 0xE0) >> 2) | ((b
```

```
            const patternIndex = (x & 0b11) | ((y & 0b11) << 2);
            const color3Bit = candidateCache[cacheKey][dither16Pattern[p

            const startBitIndex = y * 600 * 3 + x * 3;
            setBitByIndex(startBitIndex, color3Bit & 1);
            setBitByIndex(startBitIndex + 1, (color3Bit >> 1) & 1);
            setBitByIndex(startBitIndex + 2, (color3Bit >> 2) & 1);
        }
    }
```

After the image is encoded, I simply read the whole thing from the SD card into `108Kb` buffer on the Raspberry Pico W side and then copy it into the Pimoroni SDK's eInk screen buffer. The whole thing is only a couple of functions, most of which is handling IO errors:

```
#define IMAGE_WIDTH 600
#define IMAGE_HEIGHT 448
#define IMAGE_SIZE ((IMAGE_WIDTH * IMAGE_HEIGHT * 3) / 8)

InkyFrame inky;

uint8_t extract_bit(const uint8_t* bytes, size_t index) {
    size_t byte_index = index / 8;
    size_t bit_index = index % 8;
    return (bytes[byte_index] >> bit_index) & 1;
}

void display_image(const char* filename) {
    FIL file;
    res = f_open(&file, filename, FA_READ);
    if (res != FR_OK) {
        printf("[ERROR] Failed to open image '%s', code: %d\n", file
        return;
    }
```

```
    uint8_t buffer[IMAGE_SIZE];
    size_t read = 0;
    res = f_read(&file, buffer, IMAGE_SIZE, &read);
    if (res != FR_OK) {
        printf("[ERROR] Failed to read contents of image '%s', code:
        return;
    }

    if (read != IMAGE_SIZE) {
        printf("[ERROR] Image '%s' has the wrong size, actual: %d, e
        return;
    }

    for (size_t y = 0; y < IMAGE_HEIGHT; y++) {
        for (size_t x = 0; x < IMAGE_WIDTH; x++) {
            size_t base = y * IMAGE_WIDTH * 3 + x * 3;
            uint8_t color = extract_bit(buffer, base) | (extract_bit
            inky.set_pen(color);
            inky.set_pixel(Point(x, y));
        }
    }

    inky.update(true);
}
```

The only thing left to do was to implement some simple controls for switching to a different photo using buttons on the frame. Clearly, after inventing a custom image format and a whole frontend for it, this would be the easiest part, right? Right?..

Sweet dreams and wake-ups

Remember I told you that there are no loops in the examples to the Pimoroni SDK? Even the image gallery example just exited after displaying one image. This got me curious - wasn't the image gallery supposed to react to the user inputs and, well, change the image? I tried to copy the same approach into my code and it did not

work. As expected, the frame displayed a single image and immediately exited. Thinking that the problem must be on my side, I have tried to remove as much as possible from my code. In the end, it was just blinking with a LED, but it still did not work. Loosing my faith and sanity, I tried to copy the code from the example as is - it still did not work!

After I lurking around Pimoroni forums for some time, my eye caught a curious word mentioned in the context of Inky Frame - RTC. I assumed that RTC here probably means Real-time Clock. I remembered that Pimoroni SDK has a method `inky.sleep(NUMBER_OF_SECONDS)`, which curiously was always called just before the final `return` statement in the `main` function. I lurked around some more and found a phrase “RTC wake-up”. Sudden realization started to crawl in my head.

Real-time Clocks are usually external devices. External devices are powered by battery. No battery - no external device. No RTC - no wake-ups. No wake-ups - no code running. Since my battery pack did not work, the frame was powered by the USB the entire time.

I searched the forum for these specific keywords and, sure enough, there was a thread about it. Every time you ask the frame to `inky.sleep(..)`, Raspberry Pico W is fully powered down. Button presses and RTC on board can cause the Pico W to “wake up”, running the code from the top. No wonder there were no loops in the examples - the frame hardware provided the guarantee to run the code in a loop. But *only* if the battery was provided, which, of course, was not documented anywhere. Actually, none of this was documented anywhere. I literally pieced the whole thing from replies on Pimoroni forums and this is not okay for a product with a retail price of ~100$.

I was not happy to discover this. Running out of options, I implemented the busy loop again. A true Grug Brained Developer move would be to implement the busy loop from the start, but we all have something to learn I guess.

Conclusion

I am pretty happy with how this project turned out. The new version written in C++ is stable and has been working good for over a month now with no issues. My wife can easily add new photos or remove old ones when she wants to. She grew fond of this little thing and it warms my heart.

Regarding the frame itself - I am honestly not sure who this product targets. The RAM of Raspberry Pico W is clearly no fit for modern photography. There is no external RAM chip on the board itself and no interface to stream the image into the chip. The screen is clearly not fit for any interactive purposes, because the update time is 30 seconds and it does not support partial updates. The only two use cases left are calendars (seriously?) and images from James Webb Space Telescope. If you go off the marketed path, then you are completely on your own, with no documentation and little hope that whatever you come up with fits into the device capabilities. One can say that this is “a true hacker spirit”. In my opinion, “a true hacker spirit” is reverse-engineering a chip which was made before you were born. This is just a poorly documented product with a limited potential.

🖤

***

🐝 Help bumblebees

← Fire Chicken Webring →