chips

MDPI

Article

Memory Optimisation on AVR Microcontrollers for IoT Devices' Minimalistic Displays

Antoine Bossard

Graduate School of Science, Kanagawa University, Tsuchiya 2946, Hiratsuka 259-1293, Japan; abossard@kanagawa-u.ac.jp

Abstract: The minimalistic hardware of most Internet of things (IoT) devices and sensors, especially those based on microcontrollers (MCU), imposes severe limitations on the memory capacity and interfacing capabilities of the device. Nevertheless, many applications prescribe not only textual but also graphical display features as output interface. Due to the aforementioned limitations, the storage of graphical data is however highly problematic and existing solutions have even resorted to requiring external storage (e.g., a microSD card) for that purpose. In this paper, we present, evaluate and discuss two solutions that enable loading fullscreen, optimal 18-bit colour image data directly from the MCU, that is, without having to rely on additional hardware. Importantly, these solutions retain a very low footprint to suit the microcontroller architecture; the AVR architecture has been selected given its popularity. The obtained results show the feasibility and practicability of the proposal: in the worst case, 21 Kbytes of memory are required, in other words approximately 33% of the flash memory of a 32-Kbyte MCU remain available.

Keywords: IoT; sensors; microcontroller; interface; display; storage; output

Citation: Bossard, A. Memory Optimisation on AVR Microcontrollers for IoT Devices' Minimalistic Displays. *Chips* **2022**, *1*, 2–13. https://doi.org/10.3390/chips1010002

Academic Editor: Pak Kwong Chan

Received: 9 March 2022
Accepted: 15 April 2022
Published: 21 April 2022

1. Introduction

Internet of things (IoT) devices and sensors are in general small units with minimal interfacing. For example, a temperature sensor can both report its measurements via Internet for remote monitoring, and show on a display panel the current temperature in the room. Humidity measurement is another example. In recent years, smart agriculture has relied on such technology: adding the remote feature to sensing devices in, say, a crop factory is obviously attractive [1]. Weather forecast is yet another popular usage: the inferred meteorological condition is displayed as a graphical icon (sun, cloud, rain, etc.) on a small screen. These are only a few of the numerous application examples of output interfacing for IoT devices and sensors.

Because of power consumption (battery) and cost issues, the hardware of such devices is minimalistic [2]. This is especially the case when they are based on microcontrollers: very few memory is available, and interfacing is reduced to a minimum. Yet, as explained above, some applications can prescribe a display interface, both textual and graphical.

Displays for such minimalistic devices are often either a text-only liquid crystal display (LCD) panel that is only capable of displaying hard-coded characters (e.g., displays based on the Hitachi driver HD44780 [3]), very small thin-film transistor-LCD (TFT-LCD) panels (e.g., 18-bit colour TFT-LCD panels based on the Sitronix driver ST7735R [4], like the Adafruit 0.96 in, 1.44 in and 1.8 in ones) or electronic paper (e.g., WaveShare devices, like [5]). Text-only displays with hard-coded characters do not face memory issues as only character codes, like ASCII codes, need to be transferred to the peripheral. Regarding electronic paper, because they are in most cases monochrome or near-monochrome (e.g., dual colour) displays and because only small size devices are considered here, fullscreen pixel data can easily fit into the microcontroller memory [5]. Therefore, we focus hereinafter on 18-bit colour displays as they exacerbate the memory limitation issue. Precisely, we have selected

the larger 1.8 in TFT-LCD version since it requires the most memory compared to other, smaller sized TFT-LCD panels. (A smaller size here means a lower display resolution.)

It is not possible to load a fullscreen, optimal 18-bit colour image as is from the microcontroller unit (MCU) into the memory: there are too many pixels, so they cannot all fit in the program memory (details are given in the next section). Existing workarounds involve extra hardware, for instance a microSD card and the corresponding card reader, and thus have a non negligible impact on the system cost. For example, a small TFT-LCD panel mounted on a such a board with a card reader (but not including a microSD card) costs about twice as much ($19.95 for the Adafruit 1.8 in TFT-LCD as of March 2022 [6]) as the exact same TFT-LCD panel on its own ($9.95 as of March 2022 [7]). Even if part of this difference can be explained by the soldering work needed, these numbers clearly show the cost impact that this memory issue has on hardware.

So, our objective is to optimise the extremely limited storage capacity of such minimalistic devices. To this end, we describe, then evaluate, in this paper two very low footprint solutions for AVR microcontrollers to address this severe memory limitation issue in order to enable loading of fullscreen, optimal 18-bit colour image data directly from the MCU. These two techniques, colour indexing and run-length encoding (RLE), are combined in an attempt to further reduce memory usage. Although these two solutions have already been applied to a variety of computing scenarios, such as accessibility improvement for visually impaired users and information hiding (steganography) for colour indexing and three-dimensional display with electro holography for RLE [8–11], it is here interesting to concretely describe, practically measure and quantitatively discuss their implementations and performance thereof in the case of such specific and limited hardware.

The rest of this paper is organised as follows. Additional details on the selected hardware are given in Section 2. Then, the proposal is described in Section 3 before being both qualitatively and quantitatively evaluated. The obtained results are presented in Section 4. These results are discussed and compared to those of related works in Section 5. Finally, this paper is concluded in Section 6.

2. Preliminaries

When making hardware choices, it is recalled that the adoption (popularity) of a chip is key as it directly impacts, or even conditions further developments. Effectively, a widely adopted hardware architecture enjoys multiple software libraries and peripherals. In other words, technical specifications such as the clock frequency and the amount of available memory of a chip are generally imposed by the selected ecosystem. This is why it is relevant to consider the 8-bit AVR architecture and precisely the ATmega328P chip: it equips the extremely popular microcontroller board Uno of the Arduino ecosystem, ranked no. 1 by Amazon.com in the Robotics (Industrial & Scientific) category (https://www.amazon.com/gp/bestsellers/industrial/8498884011/, (last accessed 8 March 2022)) as of March 2022, and which "has been used in thousands of different projects and applications" to quote the official guide introduction (https://www.arduino.cc/en/Guide/Introduction, (last accessed 8 March 2022)). Besides, the price of 8-bit AVR chips is very competitive: for example, the chip used in this research is sold for $2.70 by the supplier (Microchip Technology, part no. ATMEGA328P-PU) and a very similar chip (same core size and speed, same program memory size, comparable connectivity and peripherals) by another manufacturer, National Semiconductor (manufacturer part no. COP8CDR9IMT7), for $3.06 by the supplier (Rochester Electronics) at the time of writing. (Both chips are still available for purchase as of March 2022).

In addition, small size TFT-LCD panels are well adapted to minimalistic devices, such as sensors, as they are light, thin (e.g., 2.4 mm [7]), provide high colour display—even better: an 18-bit colour depth is frequent—while limiting the required memory size, albeit with a resolution that is greatly superior to, for instance, LED matrix panels. Furthermore, they do not suffer from the lengthy display update and poor colour support of electronic papers [5]. And obviously, TFT-LCD panels have a more flexible usage than text-only displays: they can display whatever is needed, not only characters. Finally, their cost is, as mentioned previously,

generally lower than, or on a par with the aforementioned alternatives: for example, $9.95 for a 1.8 in 18-bit colour TFT-LCD (Adafruit part no. 618 [7]) versus $17.50 for a 2.13 in monochrome flexible electronic paper (Adafruit part no. 4243), $15.95 for a bicolour LED matrix (Adafruit part no. 902), $9.95 for a monochrome 2-row 16-character text-only display (Adafruit part no. 181). (All the prices have been taken from the same vendor for fair comparison; they are as of March 2022. The price of the raw display panel was taken, when available.)

The microcontrollers that are based on the AVR instruction set architecture (ISA) have proven to be very popular for sensors and Internet of things devices. A reduced power consumption and ease of use are two important properties which can explain this trend. The AVR ISA abides by the reduced instruction set computer (RISC) architecture principles: it consists in few, simple instructions. The memory architecture of such microcontrollers is also reduced to a minimum: for example, the popular ATmega328P chip which equips many Arduino boards (Uno, Nano, etc.), only features 32 Kbytes of program memory in flash and 2048 bytes for the data memory space in SRAM. (1024 bytes of EEPROM non-volatile memory are also provided [12]).

In this research, in order to reduce the footprint of the proposed system and the memory usage in general as much as possible, we directly rely on assembly instructions. According to the AVR ISA, data in the program memory space can be read with the LPM instruction, while the data memory space in SRAM can be accessed with the LD instruction [13]. In this research, we rely on both of these memory spaces and thus on both of these two instructions.

These hardware characteristics demand comparable, simple I/O interfacing which accommodates itself to the extremely limited memory capacities of the MCU. The selected 1.8 in TFT-LCD panel features an 18-bit colour depth, which is common for such low-specification TFT-LCDs. In other words, the colour of one single pixel is expressed with eight bits, and there are in total 2^{18} = 262,144 available colours. The resolution of the display is 128 × 160 pixels, with thus a total of 20,480 pixels. Hence, this display has a pixel density of

$$\frac{\sqrt{128^2 + 160^2}}{1.8} \approx 113.83$$

pixels per inch. The 18-bit colour mode expressing each red, green, blue (RGB) channel with one byte, an uncompressed fullscreen image requires 3 × 20,480 = 61,440 bytes, which is, as explained, problematic considering the amount of available memory. We control this display device with the serial peripheral interface (SPI) from an ATmega328P microcontroller and in accordance with the display driver (Sitronix ST7735R [4]).

3. Methodology

The two compression techniques and, importantly, their very low-footprint implementations are described in this section. We naturally focus here on the processing done by the microcontroller; preprocessing, such as image data preparation, is only briefly presented as not directly related to our research subject.

3.1. Colour Indexing with a Palette

3.1.1. Approach Description

As explained previously, it is not possible to store fullscreen 18-bit colour image data as is directly into the program memory space (flash memory) of the MCU. So, in a first attempt at reducing the size required by the pixels, we rely on the colour indexing method: the set of colours present in the image data is reduced to a set of at most 256 distinct colours and each image pixel is mapped to the index of one palette entry. This can be done for instance with a conventional palette generation method such as that of GIMP, based on histograms [14]. And, to visually improve the resulting image, we adjust pixels with Floyd-Steinberg dithering [15].

Because both the palette and the indexed image data need to be stored into the program memory space of the MCU, we limit the maximum number of palette entries to 256. Each palette entry consists in three bytes, one per RGB channel, with the value x of each RGB channel being first converted from the range $[0, 255]$ into $[0, 63]$ with the function

$f(x) = r(63x/255)$, $r()$ rounding to the nearest integer, and second left-shifted by two bits, so as to match the native 18-bit colour format of the TFT-LCD panel [4]. As a result, storing the palette itself takes at most $3 \times 256 = 768$ bytes.

Moreover, thanks to this palette limitation, each pixel of the image data can be represented with one single byte—it is recalled that palette indexing is 0-base. Therefore, storing the image data after indexation requires exactly $128 \times 160 = 20{,}480$ bytes for the selected TFT-LCD panel. In total, palette and image data thus require at most $768 + 20{,}480 = 21{,}248$ bytes of the program memory space. Of course, colour indexing is a lossy compression method.

3.1.2. Two Loading Techniques

Now that the image representation issue has been discussed, we describe in this section how to concretely load the compressed image data so that it is displayed on the TFT-LCD panel. We present two techniques: the first one relies solely on the program memory (flash) for the loading process, whereas the second one relies on both the program memory and the data memory in SRAM for increased performance (see below). We mention in this section only the code that is directly relevant to the image loading issue. In other words, initialization of peripherals and devices (e.g., SPI) is omitted.

After being formatted in accordance with the assembler syntax, the palette and indexed image data are included into the assembly source file with the `.include` directive; both are assigned a label (`palette` and `image`, respectively) for subsequent address manipulations. The main idea of the image loading algorithm is here simply to iterate each of all the palette indices that make the image data, looking up the corresponding RGB channel values inside the palette. Details are given in Listing 1.

Listing 1. Image loading solely done with the program memory space.

```
ldi r20, lo8(palette)
ldi r21, hi8(palette)
ldi r26, lo8(image)
ldi r27, hi8(image)
ldi r24, lo8(20480) ; image size
ldi r25, hi8(20480)
clr r0

data_send:
movw r30, r26
lpm r17, Z ; current pixel's index

; index address in the palette
movw r30, r20
add r30, r17
adc r31, r0
add r30, r17
adc r31, r0
add r30, r17
adc r31, r0

lpm r16, Z+ ; red channel value
rcall transmit ; SPI transmission
lpm r16, Z+ ; green channel value
rcall transmit
lpm r16, Z ; blue channel value
rcall transmit

adiw r26, 1 ; next pixel

sbiw r24, 1 ; decrement counter
brne data_send
```

Because loading directly from the program memory in flash takes one more cycle than loading from the data memory space in SRAM (the LPM instruction requires 3 cycles against only 2 for the LD instruction), copying the entire palette to the data memory space in SRAM before doing lookups is a possible optimisation since a palette entry is likely to be accessed

more than once: the probability that a same index is used multiple times inside the image data is high. So, the loading code can be refined as shown in Listing 2; it is key to notice therein that the transfer of the red, green, blue channel values to the display are now done with the LD instruction instead of the LPM instruction as done in Listing 1. Besides, this optimisation requires the number of palette entries to be known; it is stored as prefix to the palette.

Listing 2. Image loading done with both the program memory space (flash) and the data memory space (SRAM).

```
ldi r26, lo8(SRAM) ; data space addr.
ldi r27, hi8(SRAM)
ldi r30, lo8(palette)
ldi r31, hi8(palette)
lpm r1, Z+

mov r24, r1 ; palette size calcul.
clr r25
adiw r24, 1
clr r0
add r24, r1
adc r25, r0
add r24, r1
adc r25, r0

copy: ; one channel at a time
lpm r16, Z+ ; load from flash
st X+, r16 ; copy into SRAM
sbiw r24, 1 ; decrement counter
brne copy

ldi r30, lo8(image)
ldi r31, hi8(image)
ldi r24, lo8(20480)
ldi r25, hi8(20480)
ldi r26, lo8(SRAM)
ldi r27, hi8(SRAM)
clr r0

data_send:
lpm r17, Z+ ; current pixel's index

movw r28, r26 ; index address calcul.
add r28, r17
adc r29, r0
add r28, r17
adc r29, r0
add r28, r17
adc r29, r0

ld r16, Y+ ; red channel value
rcall transmit ; SPI transmission
ld r16, Y+ ; green channel value
rcall transmit
ld r16, Y ; blue channel value
rcall transmit

sbiw r24, 1 ; decrement counter
brne data_send
```

3.2. Run-Length Encoding

Compression with colour indexing as presented and implemented earlier is a first step to reduce the size of the image data. As a second, complementary step, we rely on run-length encoding (RLE), which is this time a lossless compression method (refer, for instance, to the T.45 recommendation [16] for a sample application). The principle of this compression method is to represent a sequence of consecutive same values as a $(count, value)$ pair with *count* the number of times *value* appears in the sequence. Hence,

the size of the compression result greatly varies depending on the structure of the data: the more the consecutive same values, the smaller the result size.

Considering the severe memory restriction faced as mentioned previously, not only needs the size of the image data to be appropriately reduced, but the size of the machine code required to process such data also needs to be minimal in order to leave enough program memory available for the rest of the microcontroller program. Hence, when selecting a compression algorithm, its ease of implementation and simplicity in general are critical. Besides, although not as critical as the memory issue, the processing performance of the chip has to be taken into account: decompression of image data is expected to be fast in order to retain practicability. For these reasons, RLE compression is a meaningful solution, and this is why it is frequently used in such minimalistic environments (e.g., sensors) [17]. Finally, even if an RLE compression technique is suboptimal when dealing with images including many colours and thus few consecutive pixels of the same colour, the fact that we combine this compression method together with colour indexing with a palette, thus effectively reducing the number of colours throughout the image data and consequently increasing the probability of consecutive pixels of a same colour, induces the relevance of our approach even in this particular situation.

So, we rely on this principle and adapt it to the MCU architecture so as to optimise the results. Just as we restricted the number of palette entries to 256 in order to be able to express image data with single byte palette indices, we limit the length of a sequence of consecutive same pixel values (i.e., palette indices) to 255, that is, $1 \leq \mathit{count} \leq 255$, so as to use one single byte to iterate. Therefore, if there is a sequence of more than 255 consecutive same values, several $(\mathit{count}, \mathit{value})$ pairs are generated. Precisely, a sequence of x consecutive same values v induces $\lfloor x/255 \rfloor$ pairs of the form $(255, v)$ followed by one pair of the form (c, v) where $c = x \bmod 255$, if $c \neq 0$. The total number of such pairs is needed to process the image data, and thus added as prefix to them.

Not only can this second compression method drastically reduce the program memory required, it can also accordingly reduce the number of program memory read operations (LPM instruction), which is, we recall, a heavier operation (3 cycles) than usual ones (1 cycle). This is important for parallelisation with pipelining as provided by the MCU. Precisely, instead of executing n ($n \leq 255$) program memory read operations with the LPM instruction, we execute one read operation to retrieve *count* and another one for *value*, here a palette index.

For the sake of conciseness, the assembly source of this second method is omitted. Of course, the two loading methods described previously and detailed in Listings 1 and 2 are applicable. The main difference when implementing the RLE compression method we have just described is that image data consist of count–index pairs, so, for each such pair, *count* and *index* are read from the program memory, *index* is looked up in the palette as before and the corresponding RGB channel values are repeatedly transmitted *count* times.

4. Results

4.1. Experimental Conditions

The experimental setup is shown in Figure 1: on the left, the ATmega328P microcontroller is mounted on an Arduino Uno board, and on the right, the TFT-LCD panel is inserted into a breadboard for wiring. No additional memory is used.

Three sample images have been selected so that each of them illustrates a precise rendering scenario as detailed below:

Image 1—a text logo image, that is, featuring notably antialiasing, few colours and large monochromatic areas;

Image 2—a colour spectrum image, that is, featuring notably many colours and no large monochromatic areas;

Image 3—a “photograph” image, that is, featuring notably many colours, but with possible same or near-same colour areas.

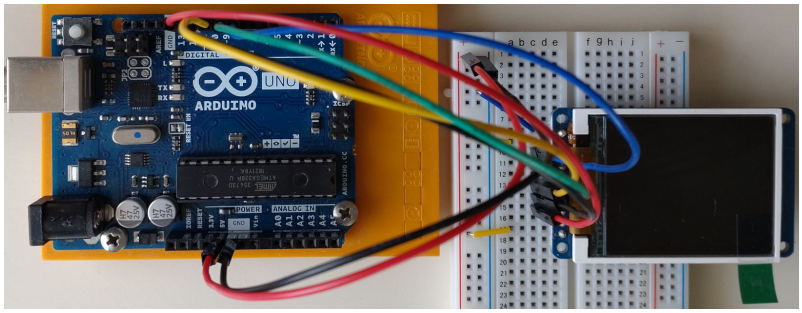

Figure 1. A photograph of the experimental setup, showing the ATmega328P microcontroller mounted on an Arduino Uno board on the left and the TFT-LCD on the right.

These original images have then been preprocessed: precisely, they have been converted with the colour indexing method described in Section 3, from which the corresponding palette has been calculated.

Finally, the source code described in Section 3 has been assembled by the `avr-as` assembler of the GNU as assembler collection. The obtained machine code has been linked with `avr-ld` of GNU `ld` and translated to the Intel HEX file format with `avr-objcopy` of GNU `objcopy`. These utilities are all from the GNU Binutils collection. The HEX file has then been conventionally transferred to the microcontroller with the `avrdude` utility (see, for instance, ref. [18] for additional details). The size of the generated machine code has been calculated with `avr-size` of GNU `size`, also from the GNU Binutils collection. This utility reports the size in bytes of each memory section (`.text`, `.data` and `.bss`); the amount of required program memory is thus given by the sum of the two `.text` and `.data` memory sections. The version of GNU Binutils is 2.26.20160125.

Renderings on the TFT-LCD panel have been photographed in daytime, indoor conditions, without direct exposure to sunlight.

4.2. Qualitative Results

As first results, we start by showing the actual renderings obtained on the TFT-LCD panel. As explained previously, we have selected three sample images which each matches one particular rendering scenario. For each of these three sample images, the actual rendering on the TFT-LCD panel has been photographed; it appears below the original sample image in Figure 2.

4.3. Quantitative Results: Machine Code Size

We have measured the size (in bytes) of the machine code produced in the following four cases, which match the two methods described in Section 3:

Palette/Flash—colour indexing (palette) compression with image loading from the program memory space in flash only;

Palette/Flash-SRAM—colour indexing (palette) compression with image loading from both the program memory space in flash and the data memory space in SRAM;

Palette-RLE/Flash—colour indexing (palette) compression combined with the proposed run-length encoding, with image loading from the program memory space in flash only;

Palette-RLE/Flash-SRAM—colour indexing (palette) compression combined with the proposed run-length encoding, with image loading from both the program memory space in flash and the data memory space in SRAM;

The results corresponding to these four scenarios are given in Figure 3 and detailed in Table 1.

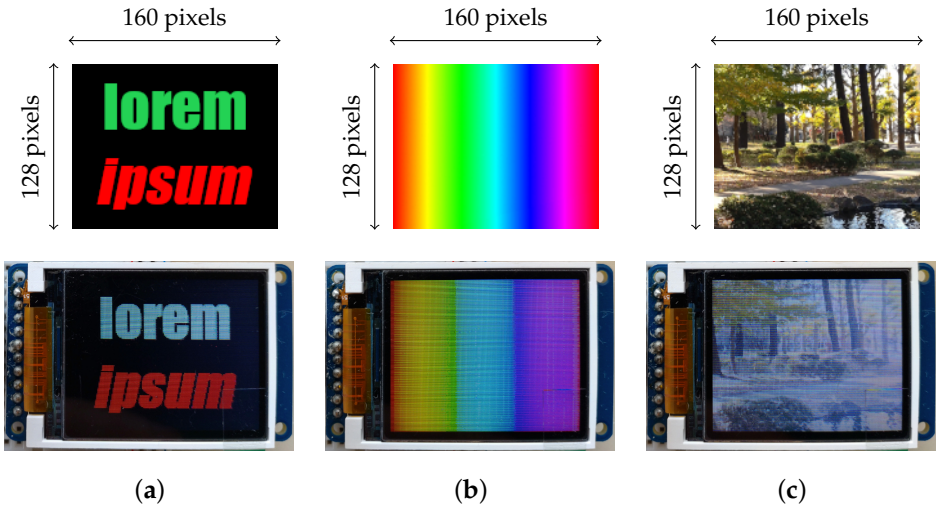

Figure 2. The rendering of three sample images which have been each selected to illustrate a precise rendering scenario: (**a**) a text logo image, that is, with antialiasing, few colours, large monochromatic areas; (**b**) a colour spectrum image, that is with many colours, no large monochromatic areas; (**c**) a "photograph" image, that is, with many colours, but with possible same or near-same colour areas. The rendering result is given below each original image.

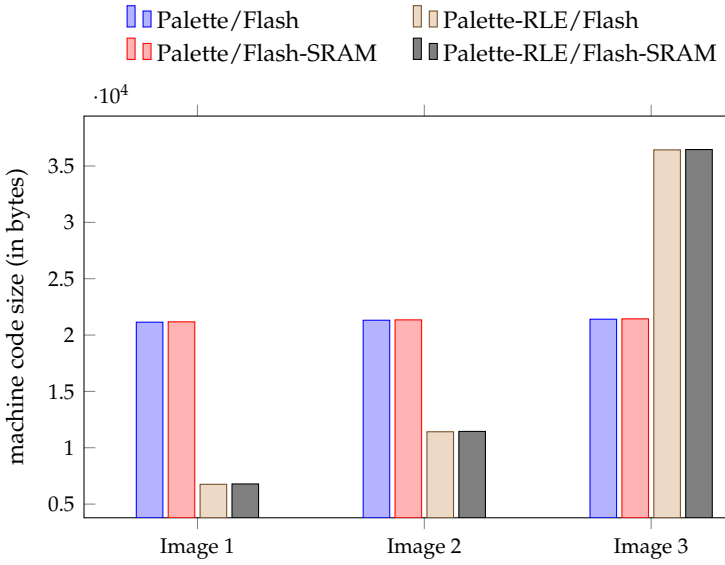

Figure 3. Size (in bytes) of the machine code produced for each of the three sample images in the four described scenarios.

Table 1. The detailed results of the size (in bytes) of the machine code produced for each of the three sample images in the four described scenarios.

	Palette/Flash	**Palette/Flash-SRAM**	**Palette-RLE/Flash**	**Palette-RLE/Flash-SRAM**
Image 1	21,142	21,174	6758	6788
Image 2	21,316	21,348	11,420	11,450
Image 3	21,406	21,438	36,432	36,462

4.4. Quantitative Results: Loading Time

We next measure the time required to load the fullscreen image from the microcontroller, data being transferred as explained to the TFT-LCD panel. The required time is expressed in the number of cycles needed to execute the corresponding program; for

reference, the clock frequency of the ATmega328P microcontroller is at most 20 Mhz [12] (external clock). It should be noted that, for the sake of clarity, we only count the instructions that load the image, not the instructions for the TFT-LCD panel setup since they are the same for all programs, and we do not count the cycles induced by the RCALL instruction since, once again, they are exactly the same for all programs. Finally, we consider a palette of maximum size, that is, with 256 entries, and a fullscreen image of $128 \times 160 = 20{,}480$ pixels so as to obtain an upper bound on the required loading time.

Because the proposed run-length encoding induces image data of varying sizes, the calculated number of cycles depends on n the number of count–index pairs in the compression result of the image file. The detailed results of the loading time, that is, clock cycles, for each of the four described scenarios are given in Table 2 and illustrated in Figure 4.

Table 2. The detailed results of the loading time (in clock cycles) for each of the four described scenarios. n is the number of count–index pairs in the compression result of the image file.

Palette/Flash	Palette/Flash-SRAM	Palette-RLE/Flash	Palette-RLE/Flash-SRAM
532,486	411,925	$11 + 1564n$	$2329 + 1558n$

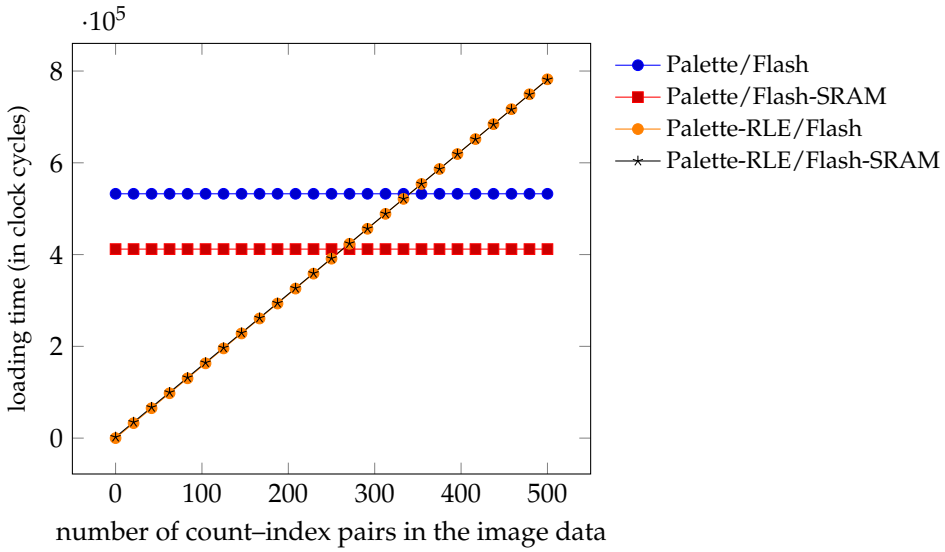

Figure 4. The loading time in clock cycles for each of the four described scenarios. The number of count–index pairs in the image data depends on the image file.

5. Discussion

5.1. General Discussion of the Obtained Results

First, we discuss the obtained qualitative results (renderings): the images were all instantly loaded and rendered successfully on the TFT-LCD panel. Besides, there are no apparent graphical artefacts or glitches other than the inevitable minor ones induced by colour indexing compression. Image 1 induces a colour palette of 223 entries (text anti-aliasing produces many colours) and Images 2 and 3 a full colour palette of 256 entries. The quality of the renderings, for instance the visible horizontal lines, matches the hardware specifications of the TFT-LCD panel used for this experiment and are thus unrelated to the proposed methodology. The reduction of the number of colours is most visible with the rendering of the colour spectrum of Image 2 (Figure 2b); it is characteristic of colour spectra since they consist of colour gradients.

While the run-length encoding compression method has been applied to binary (monochrome) images [19], we have selected this algorithm for its performance, simplicity and lossless properties. It has also enabled us to increase parallelization by reducing

the number of multi-cycles instructions (LPM, precisely), which is discussed in [20] but again for binary images.

The feasibility of the proposal has been successfully shown given that image data are successfully loaded and displayed. The practicability of the proposal has also been shown: program memory remains available, it is not fully used. Of course, this depends on the loaded image, but in the worst case (palette only, no RLE compression) $1 + 256 \times 3$ bytes are required for the palette, $20{,}480 \times 1$ bytes for the image data, plus the machine code that corresponds to the loading code—here we take the loading method that is based on the program memory & SRAM combination since we are considering the worst case—for a grand total of 21,438 bytes. Hence, since 32 Kbytes of program memory are available, approximately 33% remain available for other instructions.

Without compression, $20{,}480 \times 3 = 61{,}440$ bytes are required to represent a fullscreen image in this 18-bit colour mode, which is thus impossible to store directly into the microcontroller as explained. The results show that the proposal is able to significantly reduce the size required to represent such an image.

Depending on the image data, RLE compression can adversely affect the required memory size (i.e., a larger size than the original image data), and that even if RLE is applied to image data that have been compressed beforehand with colour indexing as in our proposal. This is precisely the case for Image 3 as shown in Figure 3; the size of the produced machine code exceeds the available amount of program memory and cannot thus be uploaded to the ATmega328P microcontroller. The worst case of such a scenario is when the image has no two consecutive pixels of the same colour.

Regarding the loading time results, they exhibit an optimal time complexity, that is, a time complexity which is linear in the size of the input data (i.e., the image data). It is recalled that these results are an upper bound on the loading time: this time is reduced for images that induce smaller colour palettes.

5.2. Comparison with Related Works

Finally, we conclude this discussion section by further showing the contribution of the proposal by comparing it to related works. First, with the Adafruit GFX Library by Phillip Burgess [21], it is not possible to load such images, only monochrome bitmaps (e.g., in the XBM format)—see the `drawBitmap` function and its variants. (Coloured bitmap files could be loaded with this library from an SD card.)

Second, we consider the RLEbitmap library by Michael Hotchin [22]. To start, it should be noted that due to its low-level memory management calls (e.g., `pgm_read_byte_far`), the RLEbitmap library is simply not usable for a low-memory microcontroller such as the ATmega328P—errors are produced when attempting to compile the image loading program. It is instead required to select a chip such as the ATmega2560, and this is yet another severe limitation compared to our proposal. Nonetheless, we specify an ATmega2560 chip to the library to further investigate.

We reuse the exact same three sample images (Images 1, 2 and 3 of Section 4) for fair comparison. These images are saved in the BMP format (uncompressed format) as required by RLEbitmap and converted to a bit map in C++ code (a byte array, precisely) by the utility provided with RLEbitmap (`bmper`). Then, the corresponding machine code for the MCU which simply displays the image on the TFT-LCD has been conventionally built with the Arduino development environment with ATmega2560 as build target. It should be noted that, as specified by RLEbitmap, the Adafruit GFX Library and Adafruit ST7735 and ST7789 Library are loaded to generate the machine code.

First, it is essential to note that the RLEBitmap library generates significantly larger machine code than our method: +132% in the case of Image 1 and +30% in the case of Image 2, approximately. The required size of the program memory space is calculated just as it was for the results of Section 4, with `avr-size`. Second, the RLEBitmap library simply fails at producing any machine code in the case of Image 3 due to the large size of this image. This library is thus again superseded by our proposal. The obtained results

with respect to the machine code size, that is, the required amount of program memory, are summarised and compared with our proposal in Figure 5.

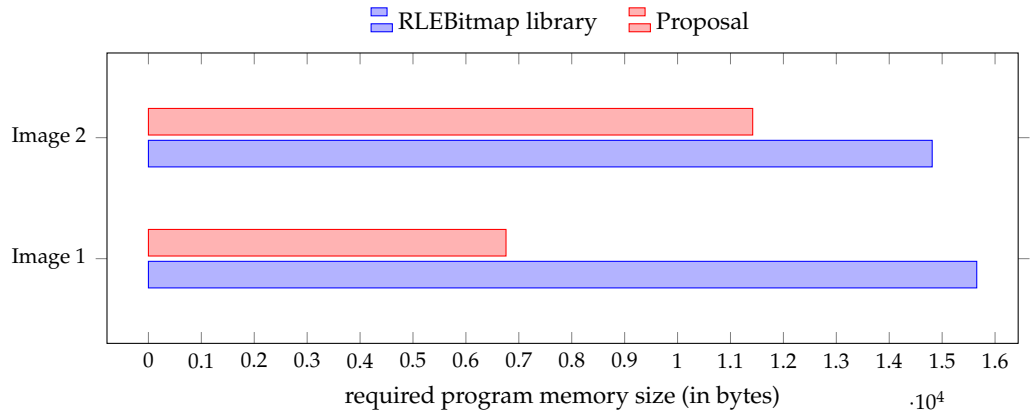

Figure 5. The obtained results with respect to the machine code size: our proposal clearly outperforms the RLEBitmap library. In the case of Image 3, the RLEBitmap library simply fails at generating machine code.

6. Conclusions

The minimalistic hardware of most IoT devices and sensors, especially those based on microcontrollers, induces severe restrictions on their storage capacity and interfacing capabilities. Nonetheless, there exist many applications that demand not only textual but also graphical display features. The storage of fullscreen data is thus highly problematic and existing solutions have even resorted to requiring external storage (e.g., a microSD card) for that purpose. In this paper, we have described two very low-footprint solutions to enable this scenario without having to rely on additional hardware. These two solutions can be combined for improved results. The proposal has been qualitatively and quantitatively evaluated, and the obtained results have been discussed. Simply stated, we have shown that what was previously impossible is now not only feasible but also practical, and can pave the way for significant cost reductions.

Regarding future works, conducting experiments with additional sample images to further generalise the obtained results is one possibility. Although it might be interesting to consider other compression algorithms, one merit of our approach is its simplicity, necessary for microcontroller architectures, and short implementation, necessary given tight storage limitations. In any case, investigating integration means of the proposal into existing applications is a meaningful objective. Finally, although devices based on the ARM architecture include generally larger memory spaces, some like the Arduino MKR Zero and the Raspberry Pi Pico still face severe memory restrictions. Considering the ARM architecture instead of AVR is thus yet another possible future research direction.

Funding: This research was partly supported by a Grant-in-Aid for Scientific Research (C) of the Japan Society for the Promotion of Science under grant no. 19K11887.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: The author wishes to thank the reviewers for their insightful comments and suggestions which helped improve the paper. In addition, the author is sincerely grateful towards Leo Nagamatsu (Kanagawa University, Japan) and Takeyuki Nagao (Chiba University of Commerce, Japan) for their insightful advices on microcontrollers.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Ijaz, F.; Siddiqui, A.A.; Im, B.K.; Lee, C. Remote management and control system for LED based plant factory using ZigBee and Internet. In Proceedings of the 14th International Conference on Advanced Communication Technology (ICACT), PyeongChang, Korea, 19–22 February 2012; pp. 942–946.
2. Bossard, A. Autonomous on-chip debugging for sensors based on AVR microcontrollers. *J. Sens. Technol.* **2021**, *11*, 19–38. http://doi.org/10.4236/jst.2021.112002. [CrossRef]
3. Hitachi. *HD44780U (LCD-II)—Dot Matrix Liquid Crystal Display Controller/Driver*; ADE-207-272(Z), '99.9, Rev. 0.0; Hitachi: Tokyo, Japan, 1998.
4. Sitronix Technology. *ST7735R—262K Color Single-Chip TFT Controller/Driver*; Version 1.4; Sitronix Technology: Jhubei, Taiwan, 2010.
5. Waveshare Electronics. *Specification—2.9″ e-Paper (B)*; Revision 2.0; Waveshare Electronics: Shenzhen, China, 2017.
6. Adafruit Industries. 1.8″ Color TFT LCD display with MicroSD Card Breakout—ST7735R. Available online: https://www.adafruit.com/product/358 (accessed on 18 April 2022).
7. Adafruit Industries. 1.8″ SPI TFT Display, 160x128 18-bit Color—ST7735R Driver. Available online: https://www.adafruit.com/product/618 (accessed on 18 April 2022).
8. Tigwell, G.W.; Flatla, D.R.; Archibald, N.D. ACE: A colour palette design tool for balancing aesthetics and accessibility. *ACM Trans. Access. Comput.* **2017**, *9*, 1–32. http://doi.org/10.1145/3014588. [CrossRef]
9. Brisbane, G.; Safavi-Naini, R.; Ogunbona, P. High-capacity steganography using a shared colour palette. *IEE Proc.—Vision Image Signal Process.* **2005**, *152*, 787–792. http://doi.org/10.1049/ip-vis:20045047. [CrossRef]
10. Xiang, L.; Wang, X.; Yang, C.; Liu, P. A novel linguistic steganography based on synonym run-length encoding. *IEICE Trans. Inf. Syst.* **2017**, *E100-D*, 313–322. http://doi.org/10.1587/transinf.2016EDP7358. [CrossRef]
11. Nishitsuji, T.; Shimobaba, T.; Kakue, T.; Ito, T. Fast calculation of computer-generated hologram using run-length encoding based recurrence relation. *Opt. Express* **2015**, *23*, 9852–9857. http://doi.org/10.1364/OE.23.009852. [CrossRef] [PubMed]
12. Microchip Technology. *ATmega48A/PA/88A/PA/168A/PA/328/P megaAVR Data Sheet*; DS40002061A; Microchip Technology: Chandler, AZ, USA, 2018; ISBN 978-1-5224-3502-0.
13. Microchip Technology. *AVR Instruction Set Manual*; DS40002198A; Microchip Technology: Chandler, AZ, USA, 2020; ISBN 978-1-5224-5882-1.
14. Arvo, J. *Graphics Gems II*; Graphics Gems—IBM; Elsevier Science: Amsterdam, The Netherlands, 1991.
15. Floyd, R.W.; Steinberg, L. An adaptive algorithm for spatial grey scale. *Proc. Soc. Inf. Disp.* **1976**, *17*, 75–77.
16. International Telecommunication Union. *Series T: Terminals for Telematic Services, Run-Length Colour Encoding*; Article no. E 18362; International Telecommunication Union: Geneva, Switzerland, 2000.
17. Saidani, A.; Xiang, J.; Mansouri, D. A new lossless compression scheme for WSNs using RLE algorithm. In Proceedings of the 20th Asia-Pacific Network Operations and Management Symposium (APNOMS), Matsue, Japan, 18–20 September 2019; pp. 1–6. http://doi.org/10.23919/APNOMS.2019.8893093. [CrossRef]
18. Bossard, A. *Understanding Microcontrollers—A Gentle Introduction to an AVR Architecture*; Ohmsha: Tokyo, Japan, 2021.
19. Qin, Y.; Wang, Z.; Wang, H.; Gong, Q. Binary image encryption in a joint transform correlator scheme by aid of run-length encoding and QR code. *Opt. Laser Technol.* **2018**, *103*, 93–98. http://doi.org/10.1016/j.optlastec.2018.01.018. [CrossRef]
20. Trein, J.; Schwarzbacher, A.; Hoppe, B.; Noffz, K.H. A hardware implementation of a run length encoding compression algorithm with parallel inputs. In Proceedings of the IET Irish Signals and Systems Conference (ISSC), Galway, Ireland, 18–19 June 2008; pp. 337–342. http://doi.org/10.1049/cp:20080685. [CrossRef]
21. Burgess, P. *Adafruit GFX Graphics Library*; Adafruit Industries: New York, NY, USA, 2021.
22. Hotchin, M. RLEBitmap—Run Length Encoded Bitmaps for the Arduino. Available online: https://github.com/MHotchin/RLEBitmap (accessed on 18 April 2022).