ARL-TN-1021 ● JUNE 2020

DEVCOM ARMY RESEARCH LABORATORY

Simulating High-Speed Objects with LEDs

by Benjamin Linne

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TN-1021 ● JUNE 2020

Simulating High-Speed Objects with LEDs

Benjamin Linne
Weapons and Materials Research Directorate, CCDC Army Research Laboratory

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
June 2020	Technical Note	1 February–30 April 2020

4. TITLE AND SUBTITLE
Simulating High-Speed Objects with LEDs

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Benjamin Linne

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
CCDC Army Research Laboratory
ATTN: FCDD-RLW-PD
Aberdeen Proving Ground, MD 21005

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TN-1021

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
ORCID ID: Benjamin Linne, 0000-0003-0128-8053

14. ABSTRACT
This report presents a method of simulating high-speed objects with LEDs. Several LED circuits are investigated to determine which one can simulate the fastest speed, and the Adafruit DotStars are chosen. To provide the necessary timing for simulating high-speed objects, a novel algorithm is presented on how to produce fast timing signals for the DotStars. This algorithm is not available in the included Adafruit library that comes with the LED strip due to limitations of using the Arduino framework.

15. SUBJECT TERMS
LED, high-speed object, DMA, SPI, NeoPixel, DotStar, Adafruit, STM32, blue pill

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Benjamin Linne
Unclassified	Unclassified	Unclassified	UU	27	19b. TELEPHONE NUMBER (Include area code) (410) 278-6219

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

List of Tables

1. Introduction and Project Overview

The high-speed object simulator is a tool created to aid in high-speed event-based sensor characterization, calibration, and algorithm development. Inspiration for this project came from the need to quickly produce a ground truth, define capabilities such as bandwidth of different sensors, and develop algorithms without needing to capture data from actual high-speed objects. The simulator replaces a moving physical object with a series of LEDs that turn on and off sequentially. Compared to launching a projectile, experiments using the simulator promise to be easier to execute and provide more consistency. The breadboard and protoboard versions of the tool are shown in Figs. 1 and 2. The requirements given for this project to simulate high-speed objects with LEDs sounds fairly simple, but it turned out to be much more complicated than anticipated. The requirements are to use a 1-m-long strip of LEDs and transition the LEDs to simulate a high-speed object traveling across from one end to the other with a speed of at least 2000 m/s. At the time of writing this report, the creation of this device successfully characterized the bandwidth of event based sensors with slight adjustments of brightness and color.

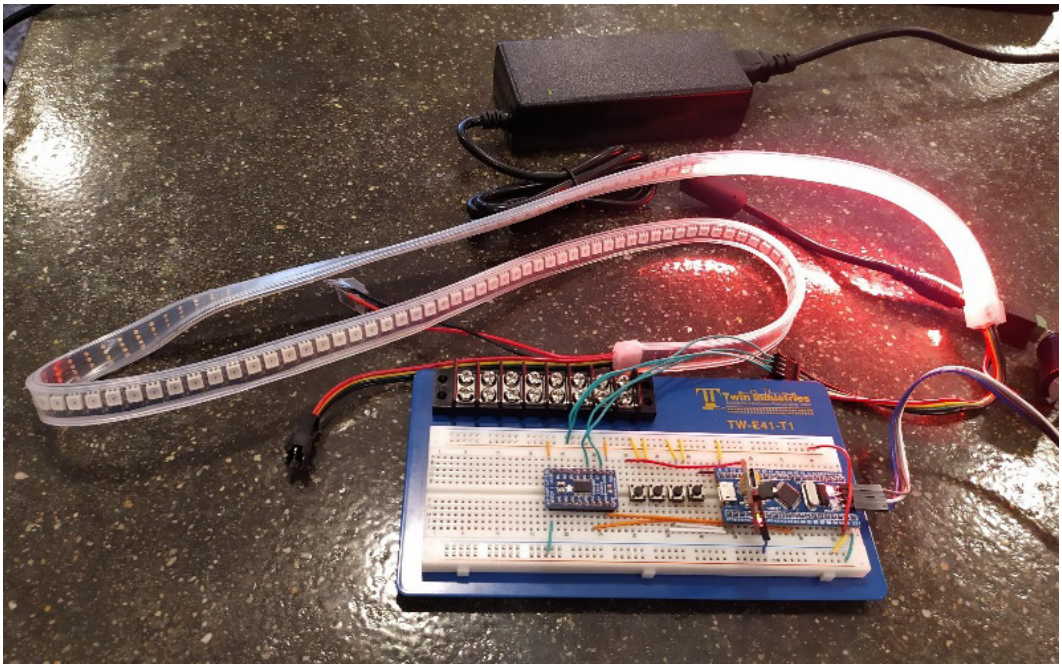

Fig. 1 Breadboard version

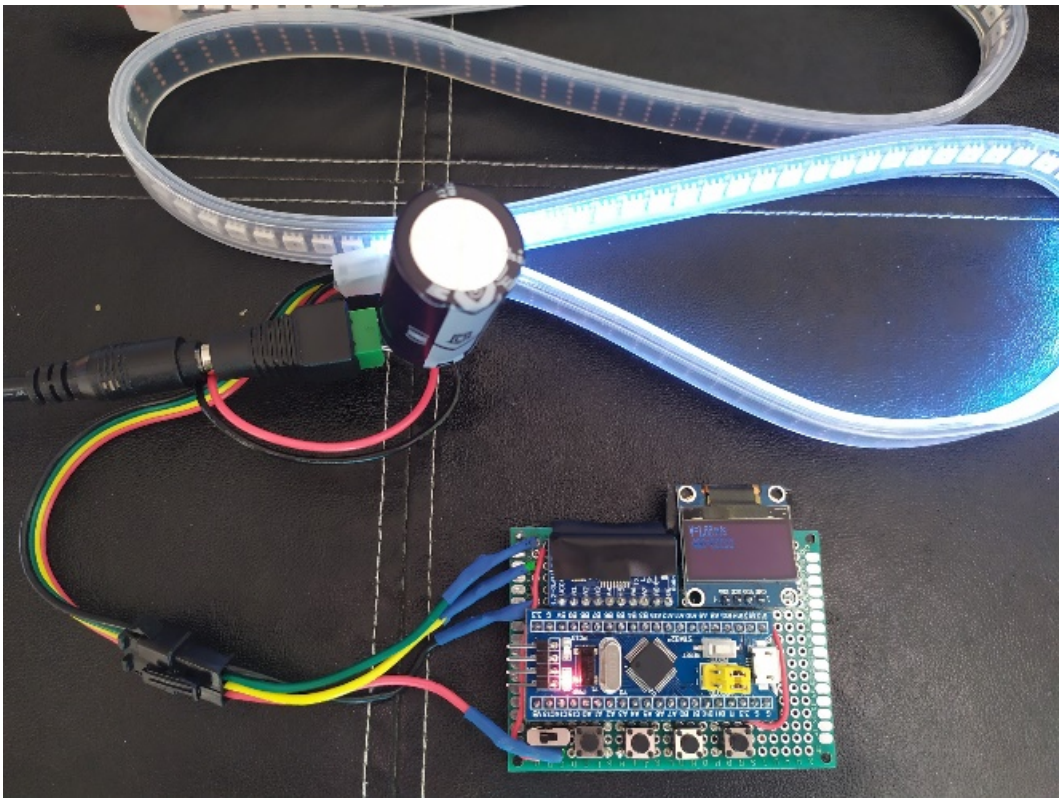

Fig. 2 Protoboard version

2. Hardware Considerations

The first approach was to simply connect numerous LEDs in a row and turn them on sequentially with a microprocessor. While that is possible, it is limited to well under 100 LEDs due to limited microprocessor pins, and it would be fairly time consuming to wire it all up. That is where individually indexable LED strips came into play.

After searching online, several LED strips were found including various lengths, and number of LEDs per length, or LED density. Conveniently, Adafruit Industries (New York, New York) had 1-m-long strips with LED densities up to 144/m. With no requirement given for the density of LEDs, the highest density was selected given that more LEDs per meter will provide a more realistic high-speed object motion.

For communication with the LEDs, two different chips were available: Adafruit's NeoPixel or Adafruit's DotStar LEDs. The NeoPixels are based off the WS2812 and WS2811 LED/drivers and have been around awhile, but require specific timing for data transfer at 800 KHz, which is significantly slower than the DotStars based off the APA102 or SK9822. The DotStars use a two-wire serial peripheral interface

(SPI) instead of the proprietary one-wire protocol. Since the DotStars use SPI, they can run as fast as the microprocessor can output data without signal distortion; also, SPI does not require specific timing, which allows precise timing for the delay between each LED and is crucial to simulate variable speeds.

The microprocessor picked was the STM32 blue pill or stm32f103c8t6, which is similar to the typical Arduino ATmega328p, but it can run at a much-higher clock speed. The blue pill runs at 72 MHz as compared to the 16 MHz of the ATmega328p, which means the blue pill is 4.5× faster at transferring data since both have a top SPI transfer speed of (clock speed)/2. The blue pill also has 64 kB of flash memory and 20 kB of static random access memory (SRAM), which is 2× flash and 10× SRAM of the ATmega328p. Having more flash and SRAM allows more features to be added later on, if desired. Finally, the blue pill paired with a ST-LINK/V2 enables debugging, which includes stepping through code line-by-line and viewing/updating variables/registers live. This makes developing code significantly faster by understanding exactly how the code is being executed.

Appendix A contains the wiring diagram and Bill of Materials (BOM).

3. Algorithm Development

The major downfall to using convenient individually indexable LED strips is the large amount of data required to simply turn them on or off. Each LED module has three colors inside: red, green, and blue; and each color requires 8 bits, along with 8 bits for brightness. This gives a total of 32 bits per LED. Given that 144 LEDs are on a strip, at least 4608 bits are needed to refresh the entire strip once ("refresh" means a simulated high-speed object [LED] moves across one LED). Then to simulate a high-speed object, 144 refreshes are needed, giving a total of $4608 \times 144 = 663{,}552$ bits. Even with the advanced chosen microprocessor, it can only output data on its SPI port up to 36 MHz, which allows a full strip refresh rate of

$$\frac{36\ Mbit/s}{663{,}552\ bits} = 54.3\ Hz \tag{1}$$

or, since the strip is 1 m, 54.3 m/s. That is 40 times slower than needed, so a different approach was taken.

The process described previously updates the entire trip by turning on one LED down the strip while also turning off the previous LED for each refresh. This is because the LEDs are wired in series, and data travels from the first LED to the next once the data for the first one have transferred, and the process repeats. Ideally, all the LED data pins would be connected in parallel and each LED could be

addressed individually, which would make the strip significantly quicker by only requiring two updates (one ON and one OFF) per refresh instead of 144. However, that would require each LED to have a unique ID and would be more difficult to manufacture. Instead, it is cheaper and easier to chain any amount of LEDs together if they are wired in series, since there is nothing unique about each LED.

A better solution is that on each refresh, only send the data required for each LED up to the one turning on. This results in the following sequence of commands: START, ON, END | START, OFF, ON, END | START, OFF, OFF, ON, END | START, OFF, OFF, OFF, ON, END |... This sequence of commands produces Table 1, which shows how many commands are needed to simulate a high-speed object given the number of pixels used.

Table 1 Commands per pixels

Pixels	No. of commands (32 bits each)
1	3
2	3+4 = 7
3	7+5 = 12
4	12+6 = 18

From Table 1, the number of commands are $\frac{5n+n^2}{2}$, where n is the number of pixels. For 144 pixels, only 10,728 commands or 343,296 bits are required. Then the speed increases to

$$\frac{36\ Mbits/s}{343{,}296\ bits} = 104.9\ m/s, \tag{2}$$

which is still significantly slower than required.

The final solution is a bit of a compromise; however, it is the only way to get quick enough speeds. It works by simply sending all ON commands in one refresh. This is different now because the simulated high-speed object increases with time as each new pixel is lit. Another refresh containing all OFF commands can be sent afterward, but for calculating speed, only the first refresh will be considered. The improved speed can be calculated as follows:

$$\frac{36Mbit/s}{(32bits\ per\ LED * 144\ LEDs} = 7813\ m/s. \tag{3}$$

That speed is well above the requirement; however, the actual speed will be slower, but still above the requirement.

The actual top speed is 3846 m/s. This slowdown from the theoretical top speed is due to signal distortion when attempting to run SPI at 36 MHz. When observing

the clock signal generated at this speed, it appears much distorted. Perhaps instead of using a breadboard and using a proper printed circuit board, or using a better level shifter, it would be possible to run at the top speed, but instead the next fastest selectable speed is 18 MHz since the clock can only be divided in increments by two.

To calculate the actual speed also relies on adding extra bytes for the APA102 protocol as described in the blog article.[1] The referenced website explains in detail how the protocol works; however, this section will focus on calculating the actual speed of the LEDs instead of the exact specifications of the protocol. The number of total bytes are: 4 bytes at the start, $144\ LEDs \times 4\frac{bytes}{LED}$, or 576 bytes for the colors, and 9 bytes at the end. Since the 4 bytes at the start do not transition any LEDs, they will not be used in speed calculations. The reason for 9 bytes at the end is because the data between each LED are delayed by half a clock cycle and a lag builds up between the LEDs. The lag does not get clocked out until at least *n*/2 bits are clocked where *n* is the number of LEDs. This means at least $\frac{144\ LEDs}{2} = 72\ bits$ or 9 bytes are needed. That means a total of 576 color bytes and 9 end bytes, or **585 total bytes**, are responsible for transitioning the LEDs. Then, the speed can be calculated as

$$\frac{clkFrequency}{\#\ of\ bits} \text{ or } \frac{18MHz}{585\ bytes * 8\frac{bits}{byte}}. \tag{4}$$

Or, converting the units $\frac{Mb/s}{bits/meter}$ gives meters per second.

This comes out to

$$\frac{18{,}000{,}000}{585*8} = 3846\ m/s. \tag{5}$$

To accomplish the high speeds calculated in the equations, a custom program was needed since the example library that came with the LEDs was not designed to run the LEDs at extreme speeds. The issue with the included library is there are large delays between sending bytes (see Fig. 3), since the CPU is responsible for setting the transmit buffer, and then polling the SPI peripheral until the data are sent to transfer the next byte. The result of the CPU transferring data from memory to the SPI peripheral reduces the theoretical speed by 10×, and the fastest recorded speed by an oscilloscope was 245 m/s.

[1] https://cpldcpu.wordpress.com/2014/11/30/understanding-the-apa102-superled/

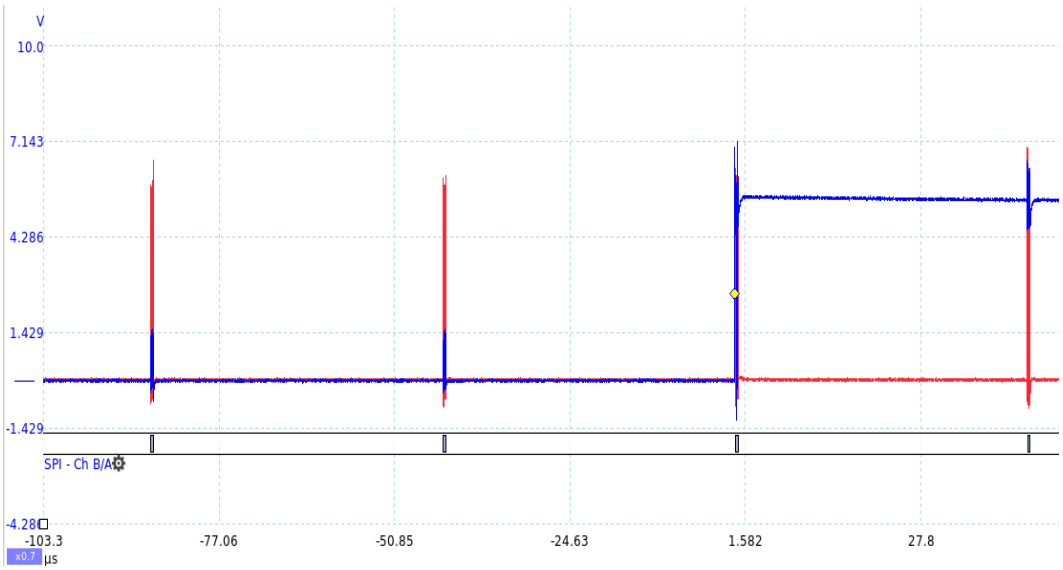

Fig. 3 Large delays between each byte being transferred

At first, development began with the Arduino framework since it was required for the included library code for the LEDs; it was also quick and easy to get started. This is because Arduino allows any microprocessor that operates with it to interface very simply with any peripheral. However, there were several issues using this library. The worst problem was the delay when transferring bytes because it uses the CPU to load data into the SPI peripheral then poll it until it can load more data. This adds significant delays between when bytes are sent. The delays can be seen in the oscilloscope output shown in Fig. 3 where the spikes are from individual bytes being transferred with delays significantly longer than the time to send a single byte. Other issues included not having full control over timers and interrupts, which are needed for custom timing, since the Arduino framework has control over them. To overcome this problem, the libopencm3 framework was used for complete control over timers and interrupts.

The delays shown in Fig. 3 can be eliminated by bypassing the CPU and letting the direct memory access (DMA) peripheral transfer data between memory and the SPI peripheral. This works by enabling the SPI peripheral to trigger DMA immediately once its transmit buffer is empty (shown in Fig. 4). This removes all delay between bytes as shown in the oscilloscope output (Fig. 5). This also confirms the calculated speed it takes to send all the color and end bytes to be 3846 m/s.

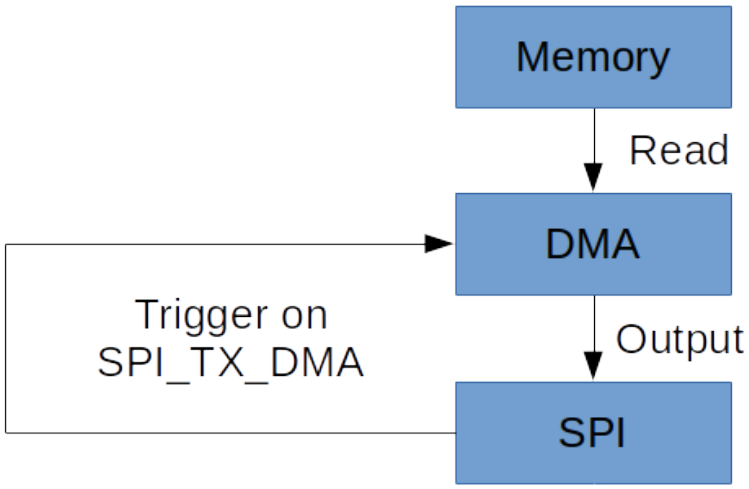

Fig. 4 DMA algorithm

Fig. 5 DMA algorithm oscilloscope output

With the system limited to its top speed, the only use is to test if a camera can detect that speed or not. Creating a more versatile variable speed requires a timer dictating the speed at which bytes are sent, as shown in Fig. 6.

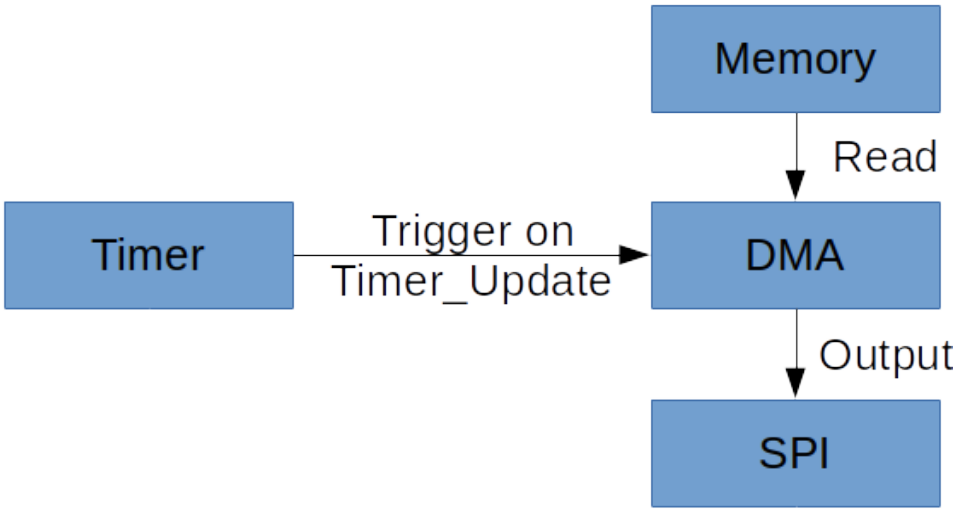

Fig. 6 Timer-triggered DMA algorithm

Controlling the time between bytes is done by setting a blue pill timer's auto reload register (ARR) to create a delay constant and setting the prescaler register divides its 72 MHz clock. To achieve maximum granularity, the prescaler is set to 0, which divides the clock by (prescaler+1) or leaves it at 72 MHz/1 so that each ARR increment adds less delay. For example, at 72 MHz a value of 1 for the ARR results in a time of 1/72 MHz = 14 ns, whereas with the same ARR and a prescaler value of 1 results in a time of 1/36 MHz = 28 ns. To calculate the variable velocity, the following equation is used:

$$velocity\ (m/s)\ = \frac{clkfreq\ of\ timer}{(ARR+1)(Prescaler+1)(\#\ of\ bytes)}. \tag{6}$$

The reason for ARR+1 and prescaler+1 is because counting starts at and includes 0. In this case, substituting for constants,

$$velocity(m/s)\ = \frac{72Mhz}{(ARR+1)(585)}, \tag{7}$$

where ARR must be greater than or equal to 31 since the timer should not trigger DMA faster than a byte can be outputted by the SPI peripheral or each byte will get truncated before it gets fully transmitted. The reason for ARR = 31 or counting to 32 is because the SPI peripheral operates at 18 Mhz, which is 4× slower than the timer and transfers 8 bits, which takes 4 × 8 = 32 times. Then, by using the velocity equation in Table 2 and Fig. 7, various ARR values and velocities are demonstrated. The largest change in velocity of –16 m/s is at the smallest ARR value, but as the ARR increases, the velocity differences become more precise. Figure 7 shows velocity decreases as ARR increases to its maximum of 65,536, which results in a velocity of 1.9 m/s as the slowest speed possible in this situation.

Table 2 Velocity (m/s) per ARR value

ARR	Velocity (m/s)	Velocity change (m/s)
31	3846.2	…
32	3729.6	–116.6
33	3619.9	–109.7
34	3516.5	–103.4
35	3418.8	–97.7
36	3326.4	–92.4
37	3238.9	–87.5
38	3155.8	–83.0
39	3076.9	–78.9
...	...	...
65,536	1.9	…

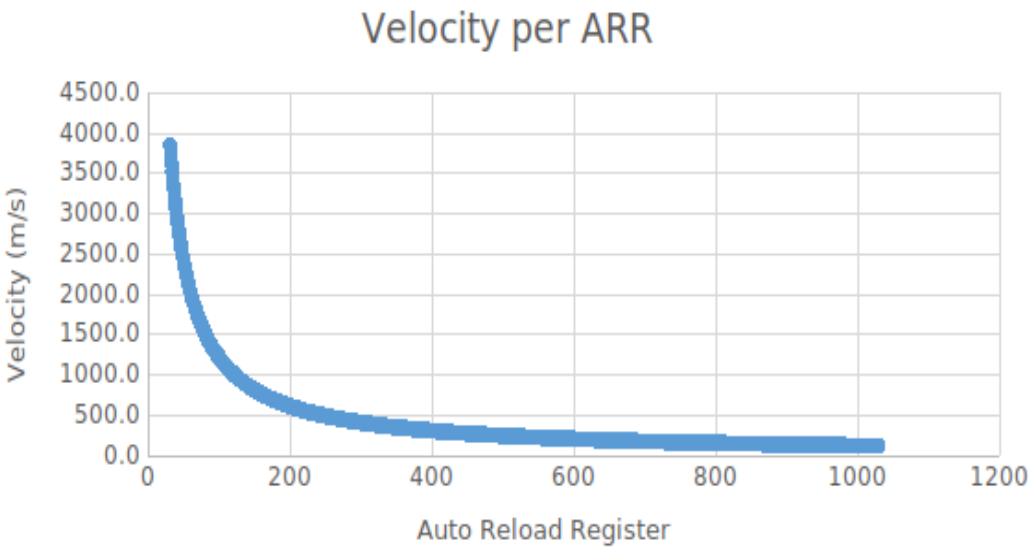

Fig. 7 Velocity (m/s) per ARR value

Appendix B contains the code for this project.

4. Improvements/Added Features

For continuously updating the LEDs so they do not need to be power cycled to reset, a modification to the algorithm was added to toggle between sending an all-on byte sequence and an all-off byte sequence every second. This was accomplished by adding a timer that interrupts a second after DMA has completed to send a byte sequence. The ARR and prescaler value for this timer were picked given that

$$time = \frac{ClkFreq}{(Arr+1)(prescaler+1)}. \quad (8)$$

Plugging in 72 MHz for ClkFreq, 9999 for ARR, and 71,999 for prescaler results in a time of 1 s. The process is displayed in Fig. 8.

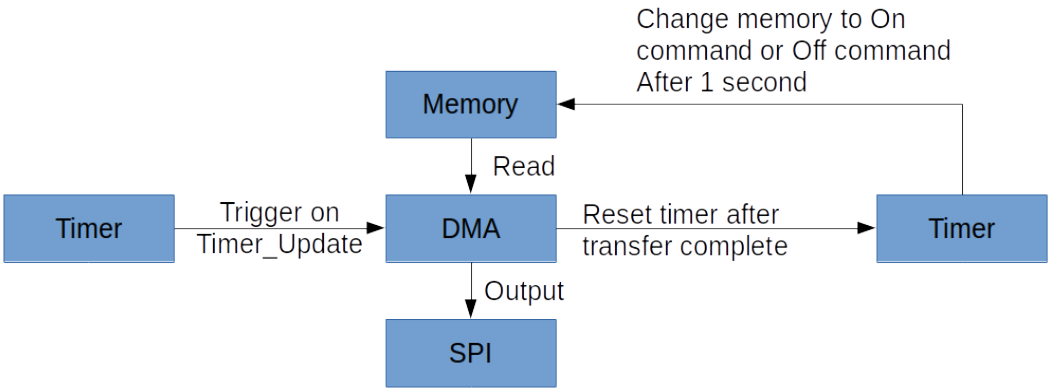

Fig. 8 Auto on/off algorithm

For changing the speed at which bytes are transferred without recompiling the code, four buttons were added to increase ARR by 1, 10, 100, and 1000. To reset ARR back to 31, the reset button can be used. To view the current ARR value and velocity in meters per second, a serial adapter was attached to USART1 and viewed in a serial monitor as shown in Fig. 9.

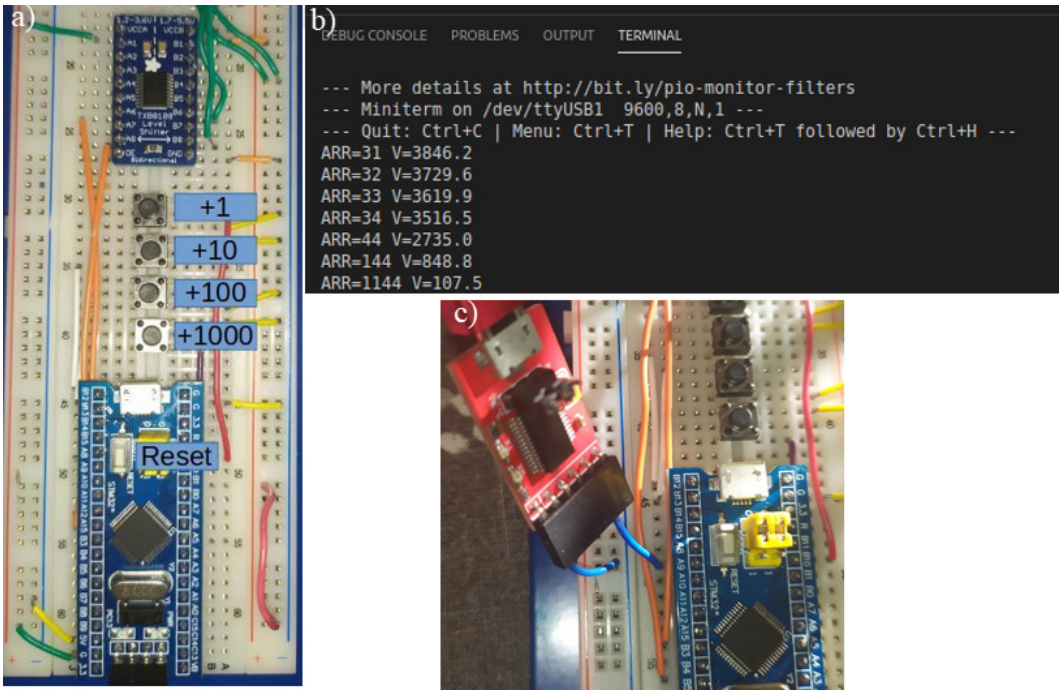

Fig. 9 a) Buttons to adjust ARR, b) serial monitor, and c) serial adapter connected to blue pill

To improve the project, an LCD was added to view the ARR and velocity values without needing a serial monitor, and to make the setup more portable, it was soldered instead of breadboarded (Fig. 10). The power-in switch must be in the correct position before connecting the ST-LINK debugger and LEDs. When debugging the blue pill, the switch must be in the left position so power only comes from the ST-LINK and not also from the LEDs. If the switch is in the wrong position, the ST-LINK could attempt to power the LEDs or the LED power supply could feed current into the ST-LINK, which could cause damage.

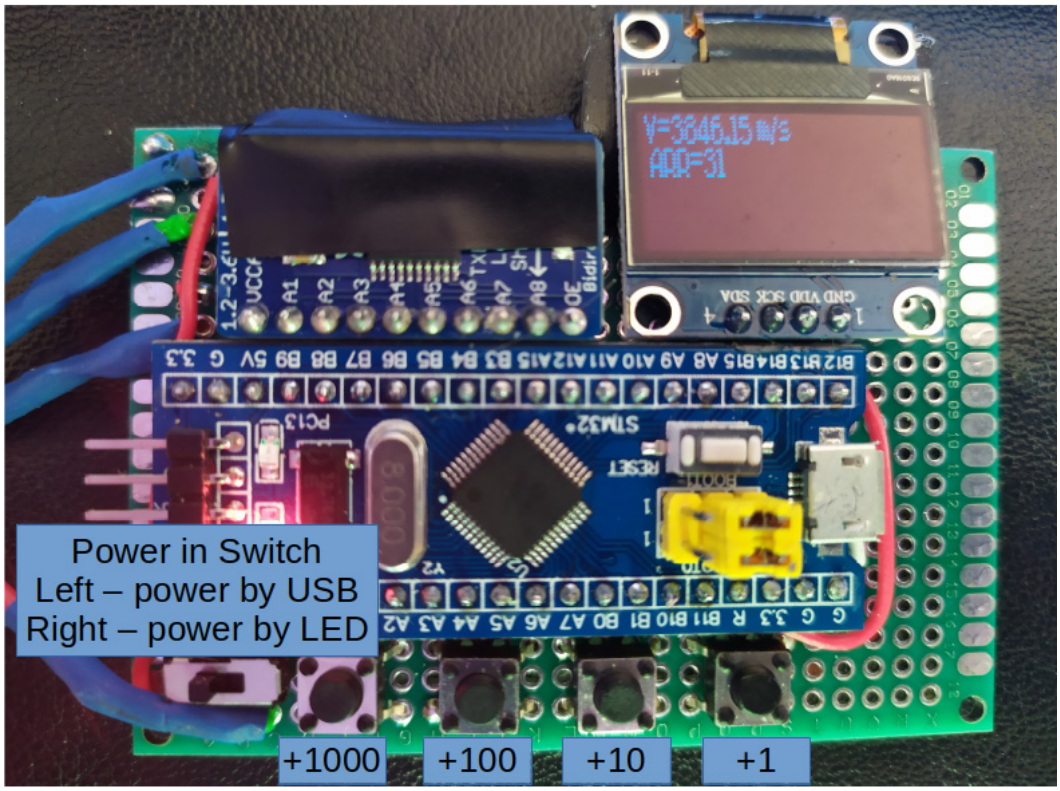

Fig. 10 Final protoboard version

5. Integrated Development Environment (IDE) Setup

For programming and debugging the blue pill, an ST-LINK/V2 is used to connect it to a computer. It is included in the BOM when purchasing a blue pill. For Windows, the linked driver is used for the ST-LINK/V2,[2] and at this link for Linux.[3] For the IDE and serial monitor, install Microsoft Visual Studio Code and then the PlatformIO extension from the extension manager. Installing the extension in Linux may result in an error that can be fixed with this solution.[4]

[2] https://www.st.com/en/development-tools/stsw-link009.html
[3] https://docs.platformio.org/en/latest/faq.html#platformio-udev-rules
[4] https://github.com/platformio/platformio-vscode-ide/issues/907

Appendix A. Wiring Diagram and Bill of Materials

Once PlatformIO is working, everything needed to create this project is found in the wiring diagram in Fig. A-1 and the bill of materials (BOM) in Table A-1.

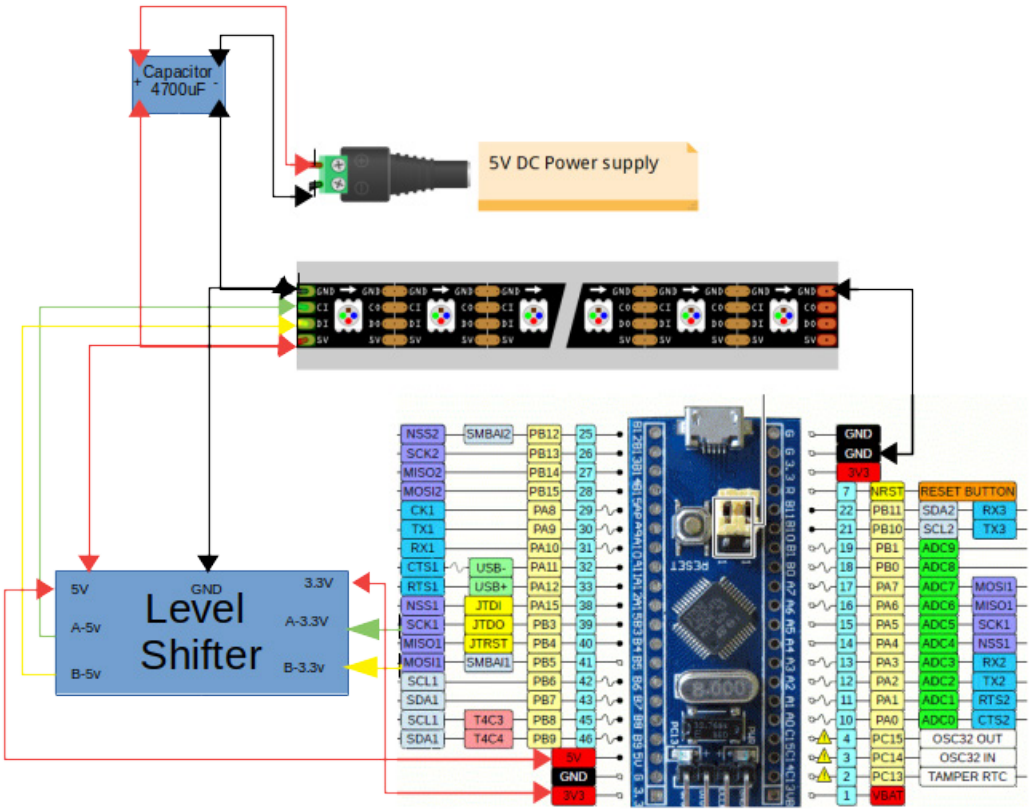

Fig. A-1 Wiring diagram

Table A-1 BOM

Product names	Price	Link
Adafruit DotStar digital LED strip – black 144 LED/m – 1 m, black	$49.95	https://www.adafruit.com/product/2241
Female DC power adapter; 2.1-mm jack-to-screw terminal block	$2.00	https://www.adafruit.com/product/368
5-V 10-A switching power supply	$29.95	https://www.adafruit.com/product/658
8-channel bi-directional logic level converter (TXB0108)	$8.00	https://www.adafruit.com/product/395
2× BLUE pill + 1 ST-LINK/V2	$18.99	https://www.amazon.com/gp/product/B079B95L9Y
4700-uF 10-V electrolytic capacitor	$1.95	https://www.adafruit.com/product/1589
Total Price	**$110.84**	

Appendix B. Project Code

This appendix contains the project code. There is also a large library used that can be found at https://github.com/StanislavLakhtin/ssd1306_libopencm3.

MAIN.C

```
#include <helper.h>
#include <libopencm3/stm32/dma.h>
#include <libopencm3/stm32/timer.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/cm3/nvic.h>
#include <libopencm3/cm3/systick.h>
uint8_t on[NUMBYTES];
uint8_t off[NUMBYTES];
uint16_t arr;
void dma1_channel3_isr(void) {
dma_clear_interrupt_flags(DMA1, DMA_CHANNEL3, DMA_TCIF);
timer_enable_counter(TIM4);
}
void tim4_isr(void) {
timer_clear_flag(TIM4, TIM_SR_UIF); //Clear update interrupt flag
if(DMA1_CMAR3 == ( uint32_t ) &on){
dma_disable_channel(DMA1, DMA_CHANNEL3);
dma_set_memory_address(DMA1, DMA_CHANNEL3, (uint32_t) off);
dma_set_number_of_data(DMA1, DMA_CHANNEL3, NUMBYTES);
dma_enable_channel(DMA1, DMA_CHANNEL3);
}else{
dma_disable_channel(DMA1, DMA_CHANNEL3);
dma_set_memory_address(DMA1, DMA_CHANNEL3, (uint32_t) on);
dma_set_number_of_data(DMA1, DMA_CHANNEL3, NUMBYTES);
dma_enable_channel(DMA1, DMA_CHANNEL3);
}
}
//polls the gpio pins every 20ms
void sys_tick_handler(void)
{
system_millis += 20;
static bool pressed = false;
if(!gpio_get(GPIOB, GPIO1) && !pressed){
printARRVelocity(++TIM3_ARR);
pressed = true;
}else if(!gpio_get(GPIOA, GPIO7) && !pressed){
printARRVelocity(TIM3_ARR+=10);
pressed = true;
}else if(!gpio_get(GPIOA, GPIO6) && !pressed){
printARRVelocity(TIM3_ARR+=100);
pressed = true;
}else if(!gpio_get(GPIOA, GPIO2) && !pressed){
printARRVelocity(TIM3_ARR+=1000);
pressed = true;
}else if( gpio_get(GPIOB, GPIO1) &&
gpio_get(GPIOA, GPIO7) &&
gpio_get(GPIOA, GPIO6) &&
gpio_get(GPIOA, GPIO2) ){
pressed = false;
}
}
int main(void)
{
//72MHz / 8 => 9000000 counts per second
//9000000/180000 = 50 overflows per second - every 20ms one interrupt
//SysTick interrupt every N clock pulses: set reload to N-1
//Start counting.
systick_set_clocksource(STK_CSR_CLKSOURCE_AHB_DIV8);
systick_set_reload(179999);
systick_interrupt_enable();
systick_counter_enable();
nvic_enable_irq(NVIC_DMA1_CHANNEL3_IRQ);
nvic_enable_irq(NVIC_TIM4_IRQ);
initializeColors(on, off);
configureClocks();
configureAlternateFunctionIO();
configureGPIO();
configureTimers();
configureSPI();
configureUSART();
configureI2C();
ssd1306_init(I2C2, DEFAULT_7bit_OLED_SLAVE_ADDRESS, 128, 64);
delayMillis(100);
oled_write("created by ben linne");
delayMillis(500);
configureDMA(on); //enables DMA which begins transfers
printARRVelocity(TIM3_ARR);
while(1);
}
```

HELPER.H

```
#include <stdint.h>
#include <ssd1306_i2c.h>
```

```
#define NUMBYTES 589
#define OLED_ATTACHED
volatile uint32_t system_millis;
void configureClocks();
void configureAlternateFunctionIO();
void configureGPIO();
void configureTimers();
void configureDMA(uint8_t *on);
void configureSPI();
void configureUSART();
void configureI2C();
void initializeColors(uint8_t *on, uint8_t *off);
void usart_send_blocking_string(const char str[]);
void oled_write(char str[]);
void printARRVelocity(uint16_t arr);
void delayMillis(uint32_t system_millis);
```

HELPER.C

```
#include <helper.h>
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/timer.h>
#include <libopencm3/stm32/spi.h>
#include <libopencm3/stm32/dma.h>
#include <libopencm3/stm32/usart.h>
#include <libopencm3/stm32/exti.h>
#include <libopencm3/stm32/i2c.h>
#include <string.h>
#include <stdio.h>
void configureClocks(){
rcc_clock_setup_in_hse_8mhz_out_72mhz(); //setup clocks for stm32 blue pill
//Enable peripheral clocks: DMA, GPIO port A and B, SPI1, Alternate Function IO, Timer3, Timer4
rcc_periph_clock_enable(RCC_DMA1);
rcc_periph_clock_enable(RCC_GPIOA);
rcc_periph_clock_enable(RCC_GPIOB);
rcc_periph_clock_enable(RCC_USART1);
rcc_periph_clock_enable(RCC_SPI1);
rcc_periph_clock_enable(RCC_AFIO);
rcc_periph_clock_enable(RCC_TIM3);
rcc_periph_clock_enable(RCC_TIM4);
rcc_periph_clock_enable(RCC_I2C2);
}
void configureAlternateFunctionIO(){
//Remap SPI1 to use PB3(SCK), PB4(MISO) PB5(MOSI)
AFIO_MAPR |= AFIO_MAPR_SPI1_REMAP;
//JTAG-DP Disabled, SW-DP Enabled
AFIO_MAPR |= AFIO_MAPR_SWJ_CFG_JTAG_OFF_SW_ON;
}
void configureGPIO(){
//configure plus1button and plus2button
gpio_set_mode(GPIOB, GPIO_MODE_INPUT, GPIO_CNF_INPUT_PULL_UPDOWN, GPIO1);
gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_PULL_UPDOWN, GPIO2);
gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_PULL_UPDOWN, GPIO6);
gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_PULL_UPDOWN, GPIO7);
GPIO_ODR(GPIOB) |= GPIO1; //set pull-up
GPIO_ODR(GPIOA) |= GPIO2 | GPIO6 | GPIO7; //set pull-up
gpio_set_mode(GPIOB, GPIO_MODE_OUTPUT_50_MHZ, GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO_SPI1_RE_SCK);
gpio_set_mode(GPIOB, GPIO_MODE_OUTPUT_50_MHZ, GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO_SPI1_RE_MOSI);
gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_50_MHZ, GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO_USART1_TX);
gpio_set_mode(GPIOB, GPIO_MODE_OUTPUT_50_MHZ, GPIO_CNF_OUTPUT_ALTFN_OPENDRAIN, GPIO_I2C2_SCL | GPIO_I2C2_SDA);
}
void configureTimers(){
// Timer3 configuration - configures timing between sending bytes
TIM3_ARR = 31; //auto reload register
TIM3_PSC = 0; //prescaler value
TIM3_DIER |= TIM_DIER_UDE; //enable update DMA request
TIM3_CR1 |= TIM_CR1_CEN; //enable timer3
//Timer4 configuration - configures timing delay toggling LEDs on and off
TIM4_ARR = 9999;
TIM4_PSC = 7199;
TIM4_DIER |= TIM_DIER_UIE; //enable update interrupt
TIM4_CR1 |= TIM_CR1_OPM; //one pulse mode
}
void configureDMA(uint8_t *on){
dma_disable_channel(DMA1, DMA_CHANNEL3);
dma_set_read_from_memory(DMA1, DMA_CHANNEL3);
dma_set_priority(DMA1, DMA_CHANNEL3, DMA_CCR_PL_VERY_HIGH);
dma_set_memory_size(DMA1, DMA_CHANNEL3, DMA_CCR_MSIZE_8BIT);
dma_set_peripheral_size(DMA1, DMA_CHANNEL3, DMA_CCR_PSIZE_8BIT);
dma_enable_memory_increment_mode(DMA1, DMA_CHANNEL3);
dma_disable_peripheral_increment_mode(DMA1, DMA_CHANNEL3);
dma_set_memory_address(DMA1, DMA_CHANNEL3, (uint32_t) on);
dma_set_peripheral_address(DMA1, DMA_CHANNEL3, (uint32_t)&SPI1_DR);
dma_set_number_of_data(DMA1, DMA_CHANNEL3, NUMBYTES);
dma_enable_transfer_complete_interrupt(DMA1, DMA_CHANNEL3);
dma_enable_channel(DMA1, DMA_CHANNEL3);
}
void configureSPI(){
spi_set_dff_8bit(SPI1);
spi_send_msb_first(SPI1);
```

```
spi_set_baudrate_prescaler(SPI1, SPI_CR1_BR_FPCLK_DIV_4);
spi_set_clock_polarity_0(SPI1);
spi_set_clock_phase_0(SPI1);
spi_set_master_mode(SPI1);
spi_enable(SPI1);
}
void configureUSART(){
usart_set_baudrate(USART1, 9600);
usart_set_databits(USART1, 8);
usart_set_stopbits(USART1, USART_STOPBITS_1);
usart_set_mode(USART1, USART_MODE_TX);
usart_set_parity(USART1, USART_PARITY_NONE);
usart_set_flow_control(USART1, USART_FLOWCONTROL_NONE);
usart_enable(USART1);
}
void configureI2C(){
/* Disable the I2C before changing any configuration. */
i2c_peripheral_disable(I2C2);
/* APB1 is running at 36MHz. */
i2c_set_clock_frequency(I2C2, I2C_CR2_FREQ_36MHZ);
/* 400KHz - I2C Fast Mode */
i2c_set_fast_mode(I2C2);
/*
* fclock for I2C is 36MHz APB2 -> cycle time 28ns, low time at 400kHz
* incl trise -> Thigh = 1600ns; CCR = tlow/tcycle = 0x1C,9;
* Datasheet suggests 0x1e.
*/
i2c_set_ccr(I2C2, 0x1e);
/*
* fclock for I2C is 36MHz -> cycle time 28ns, rise time for
* 400kHz => 300ns and 100kHz => 1000ns; 300ns/28ns = 10;
* Incremented by 1 -> 11.
*/
i2c_set_trise(I2C2, 0x0b);
i2c_enable_ack(I2C2);
/*
* This is our slave address - needed only if we want to receive from
* other masters.
*/
i2c_set_own_7bit_slave_address(I2C2, 0x32);
/* If everything is configured -> enable the peripheral. */
i2c_peripheral_enable(I2C2);
}
void initializeColors(uint8_t *on, uint8_t *off){
for(int i=0; i<NUMBYTES; i++){
if(i <= 3){// [0-3] start frame
on[i] = 0x00;
} else if(4 <= i && i <= 579){// [4-579] color frames
on[i++] = 0xE1; //brightness must be > E0
on[i++] = 0xFF; //blue
on[i++] = 0xFF; //green
on[i] = 0xFF; //red
} else if(i >= 580){// [580-588] end n/2 bits
on[i] = 0x00;
}
}
for(int i=0; i<NUMBYTES; i++){
if(i <= 3){// [0-3] start frame
off[i] = 0x00;
} else if(4 <= i && i <= 579){// [4-579] color frames
off[i++] = 0xE1; //brightness must be > E0
off[i++] = 0x00; //blue
off[i++] = 0x00; //green
off[i] = 0x00; //red
} else if(i >= 580){// [580-588] end n/2 bits
off[i] = 0x00;
}
}
}
void usart_send_blocking_string(const char str[]){
int len = strlen(str);
while(len--){
usart_send_blocking(USART1, *str);
str++;
}
}
void oled_write(char str[]){
uint32_t strlen = sizeof(&str)*8;
ssd1306_clear();
wchar_t vs[strlen];
swprintf(vs, strlen, L"%hs", str);
ssd1306_drawWCharStr(0, 0, white, wrapDisplay, vs);
ssd1306_refresh();
}
char ARRVelocityString[22];
void printARRVelocity(uint16_t arr){
sprintf(ARRVelocityString, "V=%.2f m/s\nARR=%d\n", 123076.923/(arr+1), arr);
#ifndef OLED_ATTACHED
usart_send_blocking_string(ARRVelocityString);
#else
oled_write(ARRVelocityString);
#endif
}
```

```
/*
*millis only updates every 20ms or defined by the systick interrupt
*/
void delayMillis(uint32_t millis){
uint32_t delayed = system_millis + millis;
while(system_millis < delayed) __asm__("nop");
}
```

List of Symbols, Abbreviations, and Acronyms

ARR	auto reload register
BOM	bill of materials
CPU	central processing unit
DC	direct current
DMA	direct memory access
ID	identification
IDE	integrated development environment
LCD	liquid-crystal display
LED	light-emitting diode
SPI	serial peripheral interface
SRAM	static random access memory

1 (PDF) DEFENSE TECHNICAL INFORMATION CTR DTIC OCA

1 (PDF) CCDC ARL FCDD RLD CL TECH LIB

21 (PDF) CCDC ARL FCDD RLW PD
A E BARD
N D BRUCHEY
M DEVINE
R L DONEY
M L DUFFY
S T HALSEY
M J KEELE
D S KLEPONIS
B KRZEWINSKI
B LINNE
K MASSER
F MURPHY
D W PETTY
C RANDOW
S J SCHRAML
K A STOFFEL
G VUNNI
V S WAGONER
D WEBB
M B ZELLNER
FCDD RLW PG
R GUPTA