

Projekt - Mikroprocesory

Student: Dorian Zasada 122278

Prowadzący: dr inż. Sławomir Bujnowski

Rok akademicki: 2024/2025

Politechnika Bydgoska im. Jana i Jędrzeja Śniadeckich

Spis treści

1 Założenia części projektowej	2
2 Specyfikacja projektu	2
3 Komunikacja USART	3
4 Konfiguracja Projektu (STM32CubeIDE)	7
5 Zapoznanie się z działaniem wyświetlacza	9
5.1 Wyświetlanie	9
5.1.1 Piksel	9
5.1.2 Sterowanie sygnałami SPI	9
5.2 Piksel + SPI, co z tego wyjdzie?	11
5.2.1 Rysowanie linii - algorytm Bresenhama.	11
5.2.2 Rysowanie koła - algorytm Midpoint Circle Algorithm	14
5.2.3 Wyświetlanie tekstu.	16
5.2.4 Co to jest glyph?	17
6 Projekt protokołu komunikacyjnego	19
6.1 Ramka	19
6.2 Opis ramki	19
6.3 Komendy	21
6.4 Komunikaty zwrotne:	26
7 Działanie oraz wykonanie projektu	27
7.1 Bufor kołowy	27
7.1.1 Odbiór ramki	30
7.2 Wyświetlacz	35
7.2.1 Wyświetlanie	40
8 Przykładowe ramki oraz odpowiedzi	63
8.1 Przykładowe ramki	63
8.2 Przykładowe odpowiedzi	63
8.3 Zużycie zasobów STM32L476RG - ciekawostka	65
9 Źródła	65

1 Założenia części projektowej

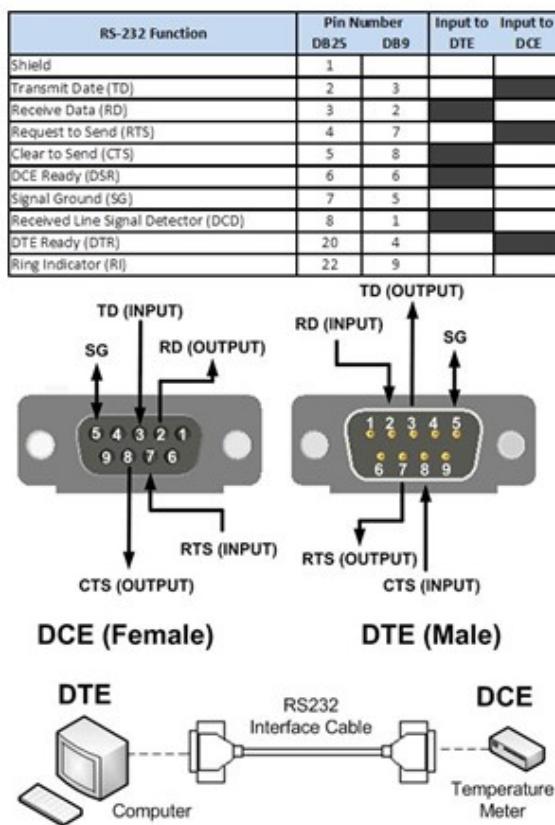
- Student potrafi pozyskać i zastosować w praktyce informacje zdobyte ze studiowania danych katalogowych i podręcznikowy dotyczących interfejsów procesora STM i ich programowania.
- Student potrafi opracować dokumentację dotyczącą zadania projektowego zarówno części związanej z wiedzą jak i oprogramowaniem.
- Student potrafi przedstawić opracowane przez siebie zadanie projektowe i odpowiedzieć na związane z nim pytania; ma umiejętność samokształcenia się, w celu podnoszenia kompetencji zawodowych.
- Student potrafi zaprojektować prosty system mikroprocesorowy i wyposażyć go w zadane przez prowadzącego układy.

2 Specyfikacja projektu

- Oprogramować komunikację mikroprocesora z PC poprzez interfejs asynchroniczny z wykorzystaniem przerwań i buforów kołowych.
- Zaprojektować i zaimplementować protokół pozwalający na adresowanie ramek, przekazywanie dowolnych danych, weryfikacja poprawności wysyłanych danych z uwzględnieniem ich kolejności.
- Obsłużyć wyświetlacz ST7735S poprzez interfejs SPI, umożliwić rysowanie figur koła prostokąta, trójkąta po zadanych parametrach wypełnionych bądź nie. Zdefiniować trzy czcionki o różnych rozmiarach, umożliwić wyświetlanie napisu w zdefiniowanym oknie z możliwością przewijania zadaną prędkością.

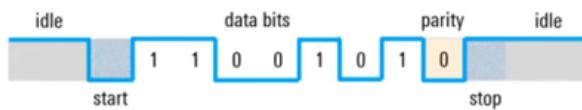
3 Komunikacja USART

Protokół komunikacyjny z PC będziemy opierać na interfejsie USART (ang. Universal Synchronous Asynchronous Receive-Transmitter). Interfejs ten charakteryzuje się tym, że możemy korzystać zarówno z modelu asynchronicznego a także synchronicznego. Obie komunikacje różnią się tym, że synchroniczny działa w oparciu o sygnał zegarowy, który jest jednakowy dla nadajnika i odbiornika. W sytuacji komunikacji asynchronicznej zegar również występuje, lecz jest on osobny zarówno dla odbiornika jak i nadajnika. Warto wspomnieć, że USART jest wbudowany w płytę. Jest to imitacja starszego już interfejsu RS-232, który był wykorzystywany w np. starszym systemie dydaktycznym mikroprocesorowym DSM-51 firmy MicroMade. Przy połączeniu płytki STM widnieje ona jako COM(odpowiedni Numer) co ukazuje nam, że jest widoczny jako połączenie portu szeregowego.



Rysunek 1: RS-232

Aby korzystać z USART należy uprzednio zapoznać się ze sposobem działania tego interfejsu. Zaczniemy od przykładowego ukazania przesyłanej ramki:



Rysunek 2: Przykład komunikacji USART

Wyróżniamy tutaj:

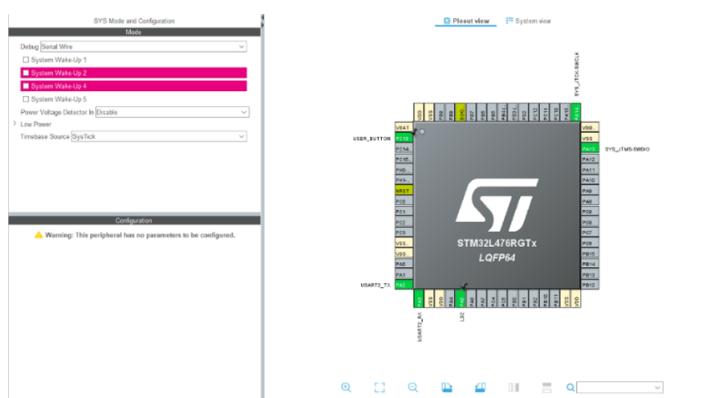
- Bit Startu (ang. Start Bit)
 - Bity Danych (ang. Data Bits)
 - Bit Stopu (ang. Stop Bit)
 - Bit Parzystosci (ang. Parity Bit) - opcjonalny

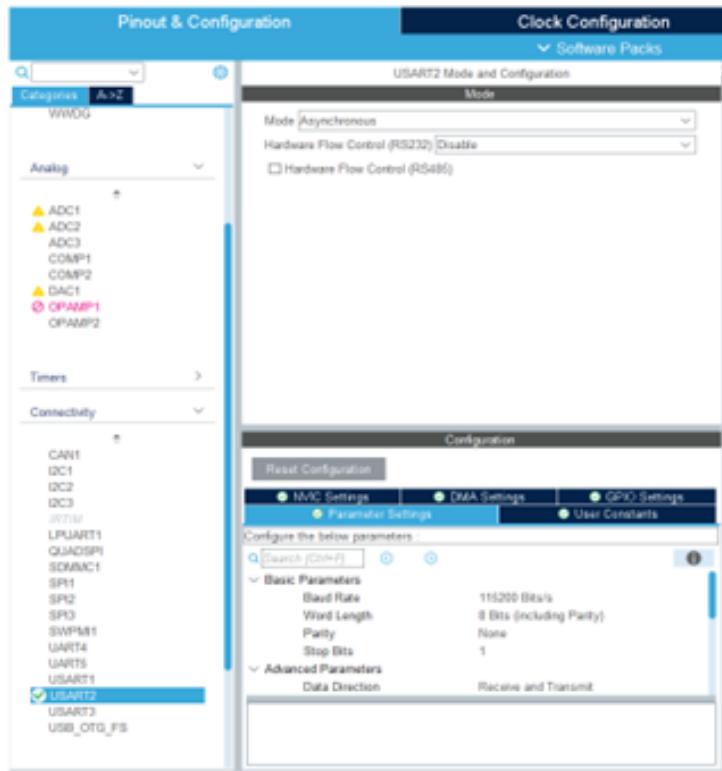
Bity startu oraz stopu mają przeważnie predefiniowane wartości:

- Bit Startu - 0
 - Bit Stopu - 1

Bity danych zawierają właściwe informacje, które chcemy przesyłać. Warto zauważyc, że podczas transmisji przetwarzane są nie tylko dane użytkowe, ale również bity kontrolne, co będzie szczególnie widoczne przy omawianiu protokołu komunikacyjnego.

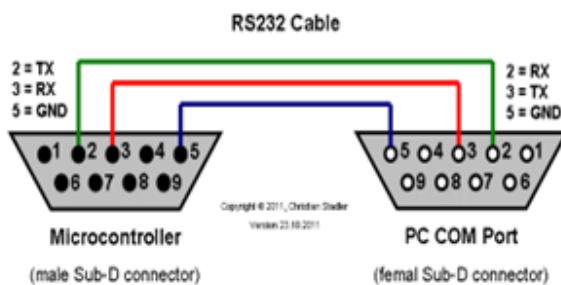
Dzięki bitowi synchronizacji możemy uzyskać odpowiedź na to czy nasze dane zostały jakkolwiek uszkodzone w procesie transmisi. Działa to tak, że dla liczby nieparzystej jedynek w danych bit jest ustawiany na wartość 1, natomiast dla liczby parzystej jedynek w danych ustawiany jest na 0. Podczas odbioru, odbiornik zlicza liczbę jedynek wraz z bitem parzystości. Jeśli suma jest parzysta, prawdopodobnie transmisja przebiegła bez błędów. Suma nieparzysta może wskazywać na wystąpienie błędu w transmisji, choć zależy to od konkretnego przypadku.





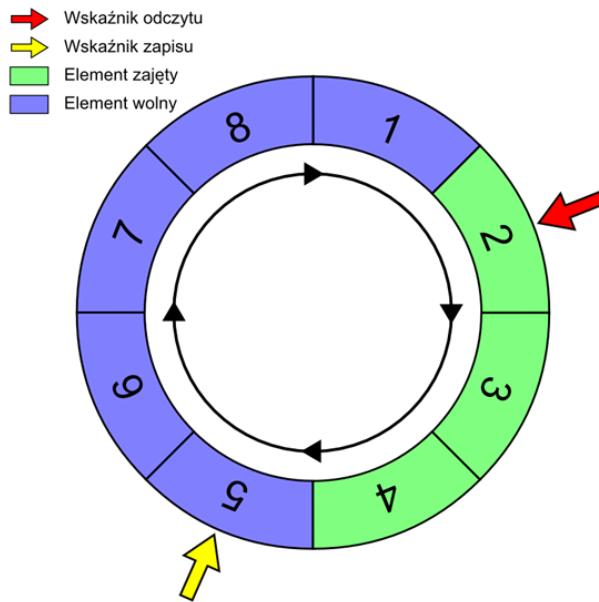
Rysunek 3: Ustawienie w widoku CubeMX

Jak widać, zostały ustawione piny PA13 oraz PA14. Dodatkowo można zauważyć, że PA2 oraz PA3 są odpowiedzialne za USART2. Jeden jest od odbierania informacji RX (Receive) a drugi od wysyłania TX (Transmit). Wynika to ze specyfikacji USART a raczej jego poprzednika RS-232.



Rysunek 4: Ukażanie złącz RS-232

Bardzo, ale to bardzo ważne jest także o pamiętaniu, aby włączyć możliwość przerwań, które będą wywoływanie przez USART'a. Tę opcję możemy ustawić w NVIC settings lub w zakładce NVIC. Jest to o tyle ważne, że bez zaznaczenia nasz późniejszy bufor kołowy nie będzie działać, ponieważ nie będą się wywoływać przerwania. Bez wywoływanego przerwań nie będziemy mieli możliwości ani nic zapisać do bufora ani nic wysłać z powrotem. Jak już wymieniłem, do komunikacji będziemy również używać bufora kołowego. Jest to struktura służąca do zapisywania danych jak sama nazwa mówi „kołowo”. Początkowo może to być niezrozumiałe, lecz jest bardzo łatwym zagadnieniem. Poniżej przedstawiam koncept:



Rysunek 5: Reprezentacja buforu kołowego

Jak widać, jest to nic innego jak tablica, która charakteryzuje się dwoma wskaźnikami – zapisu oraz odczytu danych. Gdy dojdziemy do końca tablicy to zerujemy wskaźnik i zaczynamy od początku. Jest to mechanizm prosty w implementacji i bardzo zachwalany w środowisku programistycznym, lecz także posiada swoje wady, ponieważ może się zdarzyć sytuacja gdy np. dane będą szybciej zapisywane niż odczytywane i dojdziemy do momentu gdzie wskaźniki spotkają się w tym samym miejscu w tablicy. Spowoduje to utratę ważnych danych, gdyż program pomyśli, że jeżeli wskaźniki są w jednym miejscu to oznacza że tablica jest pusta, aby temu zapobiec trzeba zostawić albo jedne miejsce puste, tzn. przesunąć od początku jeden wskaźnik albo dobrać odpowiednie wielkości buforów aby taka sytuacja nie miała miejsca. Taki bufor można zrealizować na wiele sposobów, lecz na ten moment jedyne mi znane to poprzez tablicę i wskaźniki, poprzez strukturę i tablice (struktura zawiera wskaźniki oraz wskaźnik do bufora) oraz poprzez DMA (Ang. Direct Memory Access). Ostatnia metoda jest o tyle fajna, że nie przeciąża aż tak bardzo procesora, ponieważ jest to wykonywane w tle a nie obliczane poprzez funkcje.

4 Konfiguracja Projektu (STM32CubeIDE)

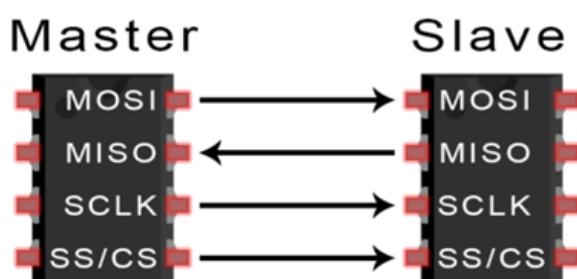
Aby odpowiednio skonfigurować projekt potrzebuję stosować się wytycznymi mojej specyfikacji. Rąbek konfiguracji już został przedstawiony, lecz pozostała dalsza część do konfiguracji tj. interfejs SPI oraz podłączenie wyświetlacza pod płytę. Moja płytką to NUCLEO-L476RG. Moduł SPI włączamy w pliku .ioc, ja wybrałem SPI2 oraz metodę komunikacji transmit only master. Wynika to z użytkowania takiego wyświetlacza, ponieważ nie będę z niego dostawać żadnych informacji, będę jedynie je wysyłać. Parametry Settings zostawiłem domyślne, zmieniłem tylko prescaler, ponieważ mój wyświetlacz obsługuje szybkość transmisji na poziomie 15Mbits/s a ja ustawiłem ten poziom na 10Mbits/s za pomocą prescalera 8. Wybrałem prescaler 8, ponieważ taktowanie mojego procesora wynosi 80MHz a reszta prescalerów jest albo za mała albo za wielka i nie spełniałaby wymogu odnośnie szybkości komunikacji. Przejdźmy teraz do podłączenia mojego wyświetlacza.

1.8inch LCD	NUCLEO-L476RG	DESCRIPTION
VCC	3V3	Power
GND	GND	Ground
DIN	PC3	SPI data input
CLK	PB10	SPI clock input
CS	PB12	Chip Selection, low active
DC	PB11	Data/Command
RST	PB2	Reset, low active
BL	PB1	Backlight

Tabela 1: Połączenie wyświetlacza z płytą STM32 NUCLEO-L476RG.

Skoro jesteśmy już przy podłączeniu, warto wspomnieć jak działa SPI i o co w nich w ogóle chodzi, ponieważ z tego też wywodzą się tutaj niektóre nazwy. SPI to nic innego jak szeregowy interfejs urządzeń peryferyjnych. Jest to jeden z najczęściej wykorzystywanych interfejsów komunikacyjnych pomiędzy systemami mikroprocesorowymi. Komunikacja odbywa się synchronicznie za pomocą 3 linii, które w naszym przypadku są bardzo ważne:

- MOSI (master output slave input) – dane dla układu slave
- MISO (master input slave output) – dane dla układu master
- SCLK (Clock) – zegar układu master



Rysunek 6: Komunikacja SPI

Do aktywacji wybranego układu służy linia SS lub CS (Slave Select lub Chip Select). W moim przypadku jak widać posiadam linię nazwaną CS i to przez nią będę wybierać mój wyświetlacz. Oczywiście trzeba mieć ciągle na uwadze, że moje urządzenie odbiega trochę od zasad podstawowej komunikacji przez protokół SPI. Zaznacza to nawet producent:

Note: there is a difference from traditional SPI. Here we only need display, so sine wires come from slave to host are hidden

Oznacza to dla nas tyle, że komunikacja odbywa się w trybie jednokierunkowym, czyli tak jak już wcześniej wspomniałem. Linia MISO pozostaje bez użytku. W przypadku tego wyświetlacza wybrałem wsparcie na 16-bitowy format pixeli. Oznacza to, że będę korzystać z RGB565.

5 Zapoznanie się z działaniem wyświetlacza

5.1 Wyświetlanie

Wyświetlanie różnych rzeczy na ekranie jest trochę rozbudowanym tematem, dlatego trzeba zacząć od początku, czyli w jakim trybie działa nasz wyświetlacz, jak są do niego przesyłane dane oraz jak on je realizuje. Wyświetlacz, w który jestem wyposażony, komunikuje się po linii SPI, a ja sam wybrałem tryb RGB565, co oznacza, że kolor każdego piksela będzie opisany za pomocą 16-bitowego słowa (5 bitów dla czerwonego, 6 dla zielonego oraz 5 dla niebieskiego). Każdy piksel jest ustawiany za pomocą odpowiedniego zapisu do pamięci wyświetlacza.

5.1.1 Piksel

Początkowo, aby zmienić piksel, trzeba odpowiednio poinformować wyświetlacz. Sterownik używa do tego rejestrów:

- Rejestr kolumny (adres X)
- Rejestr wiersza (adres Y).

Dla przykładu mojego wyświetlacza jest to odpowiednio:

- 0x2A (Column Address Ser) - ustawia zakres kolumn
- 0x2B (Row Address Set) - ustawia zakres wierszy
- 0x2C (Memory Write) - zapisuje dane kolorów do pamięci piksela

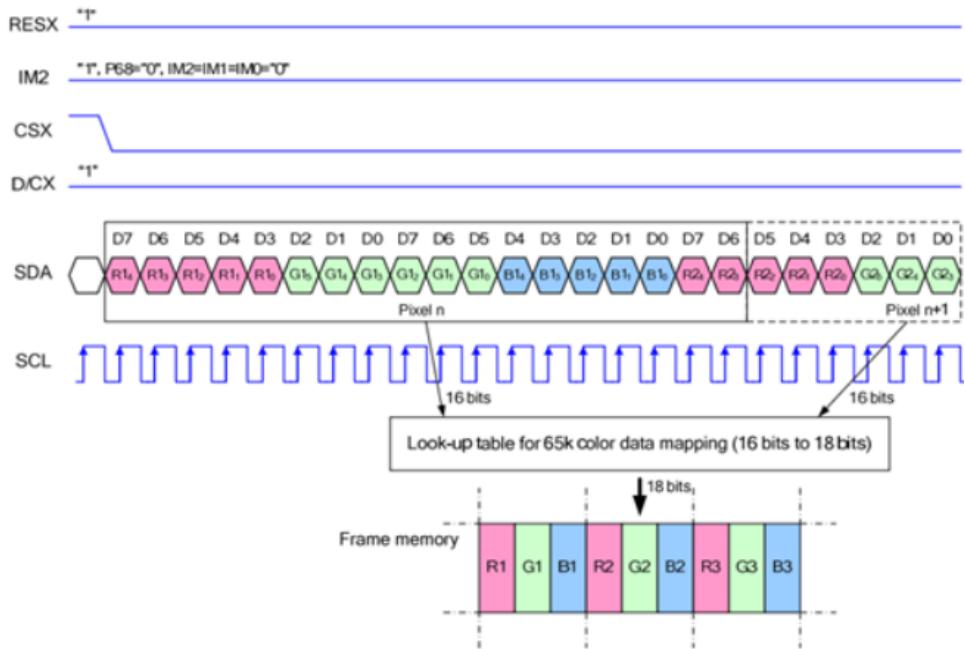
Informacje przesyłane przez SPI będą przez pojedyńcze bajty (8 bitów), dlatego też kolor musimy rozbijać na wartości 1 Bajtowe. Ze względu na wybraną obsługę kolorów RGB565 będą inaczej one zapisywane niż standardowe RGB. Kolor o wartości 0xF800 w RGB byłby kolorem zielonym, u nas byłby kolorem czerwonym.

5.1.2 Sterowanie sygnałami SPI

W poprzednim punkcie mówiłem o przesyłaniu danych przez SPI - warto byśmy w to wejść trochę głębiej. W moim przypadku do obsługi muszę wykorzystać:

- CS (Chip select) - aktywuje wyświetlacz. Muszę ustawić pin CS na stan niski, aby go wybrać.
- DC (Data/Command) - gdy ustawimy na 0 to dajemy znak, że przesyłamy komendy sterownika a w przypadku ustawienia DC na 1 wysyłamy dane.
- SCLK - zegar SPI
- MOSI - linia danych od STM32 do wyświetlacza

Niestety komunikacja SPI nie jest zbytnio ustandaryzowana, dlatego też nazwy się bardzo różnią od siebie, ale zobaczymy to na przykładzie z dokumentacji wyświetlacza.



Rysunek 7: Protokół Komunikacyjny

Na pierwszy rzut oka może się wydawać, że to kompletnie coś innego niż to co było przedstawiane, ale przyjrzymy się temu jeszcze dokładniej i odczytajmy co przygotował dla nas producent tego wyświetlacza.

Na samym wstępnie producent dał nam znać, że sterownik ST7735S obsługuje większe rozdzielczości i należy zastosować offset w kierunku poziomim 2px oraz w pionowym 1px. Wtedy dostaniemy dokładne odzworowanie punktu np. (0,0) i każdego innego.

Także dostajemy informację, że sterownik obsługuje formaty wejściowe 12-bitowe, 16-bitowe oraz 18-bitowe ale my mamy korzystać z formatu 16-bitowego.

Teraz przejdźmy do linii

- RESX - jest to linia reset. Podczas włączania modułu. Ogólnie jest ustawione na 1.
- IM2 - Pin trybu komunikacji danych, który określa użycie SPI.
- CSX - Pin kontroli wyboru chipu. Jeżeli CS = 0 wtedy chip jest wybrany i jest to częsta metodyka spotykana w komunikacji SPI.
- D/CX - Pin kontroli danych/komend. Jeżeli jest ustawione na 0 to zapisywana jest komenda, w przeciwnym razie są zapisywane dane.
- SDA - przesyłane dane RGB
- SCL - zegar SPI.

Protokół komunikacji SPI do transmisji danych używa bitów kontrolnych:

- fazy zegara (CPHA)
- polaryzacji zegara (CPOL)

Wartość CPOL określa nam poziom, gdy zegar synchroniczny szeregowy jest w stanie bezczynności. CPOL=0 oznacza, że jego poziom bezczynności wynosi 0.

Wartość CPHA określa czasowanie bitów danych w stosunku do impulsów zegara. CPHA=0, dane są próbkowane przy pierwszej krawędzi impulsu zegara.

I właśnie kombinacja tych dwóch parametrów zapewnia 4 tryby transmisji danych SPI. Najczęściej można spotkać SPI0, który ma CPOL = 0 i CPHA = 0. Z powyższego rysunku wynika, że SCLK zaczyna przesyłać dane przy pierwszej opadającej krawędzi. 8 bitów danych jest przesyłanych w jendym okresie zegara i producent zachęca nas do użycia trybu SPI0, gdzie najpierw przesyłane są bity wysokie, a następnie bity niskie.

5.2 Piksel + SPI, co z tego wyjdzie?

Dobrze, mamy już omówione, jak przesyłamy dane odnośnie piksela oraz ogólną komunikację po SPI. Posłużę się tutaj przykładem, do przedstawienia, jak to wygląda.

Załóżmy, że mamy ustawić piksel, którego koordynaty to (64, 80) oraz kolor jest kolorem zielonym (0x00F8). Teraz aby odpowiednio poradzić sobie z tymi danymi STM32 wykona następujące kroki:

- Wyślij 0x2A: Dane: 0x00 0x40 0x00 0x40 (adres kolumny 64)
- Wyślij 0x2B: Dane: 0x00 0x50 0x00 0x50 (adres wiersza 80)
- Wyślij 0x2C: Dane: 0xF8 0x00 (kolor czerwony)

Po tym sterownik LCD używa danych przesłanych przez SPI do zmiany napięcia w odpowiednich komórkach (pikselach) wyświetlacza.

! Niestety nasz układ pracuje w trybie little-endian, więc w pamięci najpierw jest zapisany młodszy bajt, a następnie starszy. Dlatego w programie trzeba odwrócić kolejność bajtów kolorów.

5.2.1 Rysowanie linii - algorytm Bresenhama.

Jak widać w tej sekcji od razu zdradziłem, czym się będę zajmować. Będzie to znany w grafice komputerowej algorytm Bresenhama, który pozwala nam na narysowanie linii, nie używając przy tym wartości zmiennoprzecinkowych. Jest bardzo prostym, ale przy tym także bardzo skutecznym algorymem. Aby zrozumieć, jak działa algorytm, musimy na początek także zapoznać się z podstawami matematyki.

Jak wszyscy wiemy, bądź nie, linia prosta w przestrzeni dwuwymiarowej może być opisana równaniem:

$$y = mx + b$$

Nachylenie linii m opisuje stosunek zmiany współrzędnych y do zmiany współrzędnych x :

$$m = \frac{\Delta y}{\Delta x}$$

gdzie:

$$\Delta y = y_1 - y_0 \quad \text{oraz} \quad \Delta x = x_1 - x_0$$

Zatem nachylenie można wyrazić jako:

$$m = \frac{y_1 - y_0}{x_1 - x_0}$$

Zasada działania algorytmu

Algorytm działa w przestrzeni dyskretnej i decyduje, który piksel najbliższej idealnej linii wybrać w kolejnych krokach, minimalizując błąd odwzorowania.

Założenia:

- Linia zaczyna się w punkcie (x_0, y_0) i kończy w (x_1, y_1) .
- Ruch odbywa się o jedną jednostkę w osi x lub w osi y.
- W każdym kroku decydujemy, czy przemieszczać się w góre (w osi y) lub w prawo (w osi x) w zależności od błędu odwzorowania.

Podstawowe definicje i wzory

Aby obliczyć nachylenie linii oraz śledzić zmiany pozycji, definiujemy różnice współrzędnych:

$$\Delta x = x_1 - x_0$$

$$\Delta y = y_1 - y_0$$

Inicjalizacja błędu:

Początkowa wartość błędu, gdy $|\Delta x| \geq |\Delta y|$, jest obliczana jako:

$$d = 2\Delta y - \Delta x$$

Decyzja w pętli:

Algorytm podejmuje decyzję o aktualizacji współrzędnej y w zależności od wartości d:

- Jeśli $d > 0$:

$$\begin{aligned} y &= y + 1 \\ d &= d + 2(\Delta y - \Delta x) \end{aligned}$$

- Jeśli $d \leq 0$:

$$d = d + 2\Delta x$$

Gdy linia jest stroma ($|\Delta y| > |\Delta x|$)

W przypadku stromych linii zamieniamy rolę osi x i y. Obliczenia są analogiczne:

- Inicjalizacja błędu:

$$d = 2\Delta x - \Delta y$$

- Jeśli $d > 0$:

$$\begin{aligned}x &= x + 1 \\d &= d + 2(\Delta x - \Delta y)\end{aligned}$$

- Jeśli $d \leq 0$:

$$d = d + 2\Delta x$$

Podsumowanie obliczeń w pętli

Każdy krok algorytmu można opisać za pomocą wzoru:

$$d_{\text{nowy}} = \begin{cases} d_{\text{stary}} + 2\Delta y, & \text{jeśli } d_{\text{stary}} \leq 0, \\ d_{\text{stary}} + 2(\Delta y - \Delta x), & \text{jeśli } d_{\text{stary}} > 0. \end{cases}$$

Przykład - Bresenham

Już tyle teorii za nami, więc już czas coś pokazać na przykładach. Mamy rysować linię od punktu $(2, 2)$ do punktu $(8, 5)$. Parametry tej linii to:

$$\Delta x = 8 - 2 = 6, \quad \Delta y = 5 - 2 = 3$$

Inicjalizacja:

$$d = 2\Delta y - \Delta x = 2 \cdot 3 - 6 = 0$$

Iteracje

- **Punkt $(2, 2)$:**

$$d = 0, \quad d \leq 0 \quad \Rightarrow \quad x = 3, y = 2, d = d + 2\Delta y = 0 + 6 = 6$$

- **Punkt $(3, 2)$:**

$$d = 6, \quad d > 0 \quad \Rightarrow \quad x = 4, y = 3, d = d + 2(\Delta y - \Delta x) = 6 + 2(3 - 6) = 6 - 6 = 0$$

- **Punkt $(4, 3)$:**

$$d = 0, \quad d \leq 0 \quad \Rightarrow \quad x = 5, y = 3, d = d + 2\Delta y = 0 + 6 = 6$$

- **Punkt $(5, 3)$:**

$$d = 6, \quad d > 0 \quad \Rightarrow \quad x = 6, y = 4, d = d + 2(\Delta y - \Delta x) = 6 + 2(3 - 6) = 6 - 6 = 0$$

- **Punkt $(6, 4)$:**

$$d = 0, \quad d \leq 0 \quad \Rightarrow \quad x = 7, y = 4, d = d + 2\Delta y = 0 + 6 = 6$$

- **Punkt $(7, 4)$:**

$$d = 6, \quad d > 0 \quad \Rightarrow \quad x = 8, y = 5, d = d + 2(\Delta y - \Delta x) = 6 + 2(3 - 6) = 6 - 6 = 0$$

Wynik

Wynikowe punkty na linii to:

$$(2,2), (3,2), (4,3), (5,3), (6,4), (7,4), (8,5)$$

BLA for slope +ve			
$ M \leq 1$		$ M > 1$	
If $P_k < 0$	If $P_k \geq 0$	If $P_k < 0$	If $P_k \geq 0$
$X_{k+1} = X_k + 1$	$X_{k+1} = X_k + 1$	$X_{k+1} = X_k$	$X_{k+1} = X_k + 1$
$Y_{k+1} = Y_k$	$Y_{k+1} = Y_k + 1$	$Y_{k+1} = Y_k + 1$	$Y_{k+1} = Y_k + 1$
$P_k = P_k + 2 \Delta y$	$P_k = P_k + 2 \Delta y - 2\Delta x$	$P_k = P_k + 2 \Delta x$	$P_k = P_k + 2 \Delta x - 2\Delta y$
$P_0 = 2 \Delta y - \Delta x$			$P_0 = 2 \Delta x - \Delta y$

BLA for slope -ve			
$ M \leq 1$		$ M > 1$	
If $P_k < 0$	If $P_k \geq 0$	If $P_k < 0$	If $P_k \geq 0$
$X_{k+1} = X_k - 1$	$X_{k+1} = X_k - 1$	$X_{k+1} = X_k$	$X_{k+1} = X_k - 1$
$Y_{k+1} = Y_k$	$Y_{k+1} = Y_k + 1$	$Y_{k+1} = Y_k + 1$	$Y_{k+1} = Y_k + 1$
$P_k = P_k + 2 \Delta y$	$P_k = P_k + 2 \Delta y - 2\Delta x$	$P_k = P_k + 2 \Delta x$	$P_k = P_k + 2\Delta x - 2 \Delta y$
$P_0 = 2 \Delta y - \Delta x$			$P_0 = 2 \Delta x - \Delta y$

Rysunek 8: Zbiór wzorów i przypadków

Złożoność Obliczeniowa

Złożoność obliczeniowa algorytmu to $O(d)$, gdzie d to długość linii, a w najgorszym przypadku jest to różnica pomiędzy współrzędnymi końcowych punktów wzdłuż osi x lub y. Jest to bardzo wydajny algorytm, ponieważ każda operacja w iteracji ma stały czas $O(1)$.

5.2.2 Rysowanie koła - algorytm Midpoint Circle Algorithm

W tym punkcie tak samo od razu podałem nazwę algorytmu, który będzie tutaj omawiany. Algorytm jest nazywany również algorytmem do rysowania okręgu przy użyciu punktu środkowego. Jego zadaniem jest wyznaczanie punktów, które tworzą okrąg, bazując na równaniu okręgu i stosując podejście "punktu środkowego", który jest bardziej wydajne niż tradycyjne obliczenia trygonometryczne.

Zasada działania algorytmu

Algorytm bazuje na zasadzie, że okrąg jest symetryczny względem czterech ćwiartek, więc wystarczy obliczyć tylko jedną ćwiartkę okręgu, a pozostałe punkty można uzyskać poprzez odbicia lustrzane.

Równanie Okręgu

Dla okręgu o promieniu r i środkowym punkcie (h, k) , jego równanie to:

$$(x - h)^2 + (y - k)^2 = r^2$$

gdzie:

- (h, k) to współrzędne środka okręgu,
- r to promień okręgu,
- (x, y) to punkty leżące na okręgu.

Inicjalizacja

Na początku algorytmu:

$$x = 0, \quad y = r$$

oraz

$$d = 1 - r$$

gdzie d jest wartością błędu, którą będziemy aktualizować w trakcie iteracji.

Punkty Symetrii

Rysowanie punktów okręgu przy użyciu symetrii pozwala na ograniczenie liczby obliczeń. Zaczynając od punktu (x, y) , możemy wyznaczyć inne punkty poprzez symetrię względem osi x , y oraz przekątnych:

$$P_1 = (h + x, k + y), \quad P_2 = (h - x, k + y), \quad P_3 = (h + x, k - y), \quad P_4 = (h - x, k - y)$$

$$P_5 = (h + y, k + x), \quad P_6 = (h - y, k + x), \quad P_7 = (h + y, k - x), \quad P_8 = (h - y, k - x)$$

Pętla Algorytmu

Pętla algorytmu przebiega do momentu, gdy $x \geq y$, co oznacza, że przeszliśmy przez całą ćwiartkę okręgu. Każdy krok pętli polega na wyrysowaniu punktów oraz aktualizacji zmiennej błędu d .

- Jeśli $d \leq 0$, wtedy przesuwamy się w poziomie, czyli zwiększamy x :

$$d_{\text{next}} = d + 2x + 1$$

- Jeśli $d > 0$, przesuwamy się zarówno w poziomie, jak i w pionie (przekątna):

$$d_{\text{next}} = d + 2(x - y) + 1$$

- Wartość y jest zwiększana, gdy $d > 0$.

Przykład

Załóżmy, że mamy okrąg o środku w punkcie $(0,0)$ i promieniu $r = 5$. Początkowy punkt to $(x_0, y_0) = (0, 5)$.

- $d = 1 - 5 = -4$
- Rysujemy punkt $(0, 5)$ i aktualizujemy $d = d + 2x + 1 = -4 + 0 + 1 = -3$.
- Następnie $x = 1, y = 5, d = -3 + 2 \cdot 1 + 1 = 0$.
- Rysujemy punkt $(1, 5)$ i $d = 0 + 2 \cdot 1 + 1 = 3$.
- Następnie $x = 2, y = 5, d = 3 + 2 \cdot (2 - 5) + 1 = 3 - 6 + 1 = -2$.
- Kontynuujemy proces, aż $x \geq y$.

W wyniku iteracji będziemy mieć punkty, które rysują okrąg w pełnej symetrii względem osi.

Złożoność Obliczeniowa

Złożoność obliczeniowa algorytmu to $O(r)$, gdzie r to promień okręgu. Dzięki zastosowaniu techniki symetrii oraz obliczeń przy użyciu prostych operacji arytmetycznych, algorytm jest bardzo efektywny.

5.2.3 Wyświetlanie tekstu.

W tym przykładzie posłużę się biblioteką HAGL, która jest udostępniona na serwisie github.com oraz na licencji MIT, czyli daje nam prawo do używania, kopowania oraz upowszechniania oryginalnego kodu. Wyświetlanie tekstu na wyświetlaczu polega na renderowaniu znaków przy użyciu bitmap. Biblioteka HAGL dostarcza funkcje takie jak `hagl_put_char` i `hagl_put_text`, które umożliwiają narysowanie pojedynczego znaku lub całego ciągu tekstowego na ekranie.

Funkcja `hagl_put_char` jest odpowiedzialna za wyświetlanie pojedynczego znaku na wyświetlaczu. Wykonuje następujące kroki:

1. Pobranie `glyph`: Funkcja `fontx_glyph` jest wywoływana w celu pobrania informacji o znaku. Zwracana jest bitmapa zawierająca dane o pikselach, które mają zostać wyświetlone.
2. Inicjalizacja bufora: Bufor jest inicjowany tylko podczas pierwszego wywołania funkcji. Bufor ten przechowuje dane o pikselach.
3. Rysowanie piksela: Funkcja sprawdza każdy piksel w bitmapie i przypisuje odpowiednią wartość kolorów do bufora.
4. Blitowanie obrazu: Po zakończeniu renderowania bitmapy, zawartość bufora jest przenoszona na ekran.

Funkcja `hagl_put_text` jest odpowiedzialna za wyświetlanie całego tekstu. Wywołuje funkcję `hagl_put_char` dla każdego znaku w łańcuchu tekstowym. Obsługuje również przejście do nowej linii przy napotkaniu znaków CR (powrót karetki) lub LF (nowa linia).

Wszystko jak na razie zostało powieszchownie omówione, dlatego teraz przejdźmy do szczegółów:

5.2.4 Co to jest glyph?

Glyph to termin używany w kontekście wyświetlania tekstu, który odnosi się do reprezentacji graficznej znaku. Jest to bitmapa, która przedstawia wygląd pojedynczego znaku na wyświetlaczu.

- **Bitmapa:** Reprezentacja obrazu jako siatki pikseli, gdzie każdy piksel ma określoną wartość (np. czarny lub biały, lub w przypadku kolorowych wyświetlaczów – wartość RGB).
- **Charakterystyka** **glyph:** Każdy **glyph** ma przypisaną szerokość i wysokość, które określają wymiary bitmapy w pikselach. **Glyph** zawiera także dane o samej bitmapie, czyli tablicę bitów, gdzie każdy bit wskazuje, czy dany piksel jest włączony (1) czy wyłączony (0).

Przed wyświetleniem tekstu na ekranie trzeba mieć czcionkę, która zawiera wszystkie możliwe znaki, które mogą zostać wyświetlane. Czcionka ta jest zapisywana w specjalnym formacie, który umożliwia łatwe przetwarzanie przez funkcję wyświetlającą tekst.

- **Font:** Jest to zestaw znaków (liter, cyfr itp.) zapisanych jako bitmapy. Font może być zapisany w tablicy lub w specjalnej strukturze danych, która przechowuje informacje o poszczególnych **glyph**.
- **Pobieranie** **glyph:** Funkcja `fontx_glyph` służy do pobrania odpowiednich danych dla konkretnego znaku (czyli **glyph**) z czcionki. Dla każdego znaku przekazywana jest szerokość i wysokość bitmapy, a także wskaźnik do samej bitmapy (czyli dane o pikselach).

Pobieranie **glyph** dla znaku

Funkcja `hagl_put_char` jako pierwszy krok wywołuje funkcję `fontx_glyph`, która pobiera bitmapę znaku. Działa to w następujący sposób:

```
1 fontx_glyph(&glyph, code, font);
```

Funkcja `fontx_glyph` jest odpowiedzialna za pobranie bitmapy z czcionki. Zmienna `glyph` zawiera wszystkie informacje o znaku, w tym:

- `width`: szerokość bitmapy w pikselach,
- `height`: wysokość bitmapy w pikselach,
- `buffer`: wskaźnik do tablicy, która przechowuje dane o każdym pikselu znaku. Jeśli piksel jest włączony, w tablicy jest zapisany odpowiedni bit, jeśli jest wyłączony – inny.

Rysowanie pikseli

Następnie następuje iteracja po każdym pikselu bitmapy i przypisanie odpowiednich kolorów do bufora:

```
1 for (uint8_t y = 0; y < glyph.height; y++) {  
2     for (uint8_t x = 0; x < glyph.width; x++) {  
3         set = *(glyph.buffer + x / 8) & (0x80 >> (x % 8));  
4         if (set) {  
5             *(ptr++) = color;  
6         } else {  
7             *(ptr++) = 0x0000;
```

```
8      }
9  }
10     glyph.buffer += glyph.pitch;
11 }
```

- `glyph.buffer`: Zawiera dane bitmapy. Bitmapa jest zapisana jako tablica bitów, gdzie każdy bit reprezentuje pojedynczy piksel.
- `(0x80 » (x % 8))`: Ta operacja sprawdza, czy konkretny bit w bajcie jest ustawiony (czy piksel jest włączony).
- `color`: Jeśli piksel jest włączony, przypisywany jest kolor, w przeciwnym przypadku kolor tła (np. czarny).

Funkcja `hagl_put_text` – Wyświetlanie całego tekstu

Funkcja `hagl_put_text` odpowiada za rysowanie całego ciągu tekstowego, iterując po każdym znaku i wywołując `hagl_put_char` dla każdego z nich. Proces ten obejmuje:

- Iterację po znakach tekstu.
- Wywołanie funkcji `hagl_put_char` dla każdego znaku.
- Zmianę pozycji na następny znak.

Podsumowanie - wyświetlanie tekstu

Proces wyświetlania tekstu na wyświetlaczu obejmuje kilka kluczowych etapów, od pobierania danych o znakach (`glyph`), przez rysowanie pikseli na buforze, aż po przesyłanie obrazu na wyświetlacz. Każdy z tych kroków jest istotny, żeby tekst był poprawnie wyświetlany na ekranie.

6 Projekt protokołu komunikacyjnego

6.1 Ramka

Bazą dla mojego protokołu był protokół HDLC (ang. High-Level Data Link Control), który został zmieniony pod wymagania opisane w specyfikacji projektu (tj. adresowanie ramek, przekazywanie dowolnych danych oraz weryfikację poprawności wysyłanych danych, z uwzględnieniem jej kolejności).

Pole	START FRAME	NADAWCA	ODBIORCA	KOMENDA	DANE	CRC	END FRAME
Rozmiar	1 BAJT	1 BAJT	1 BAJT	3 BAJTY	1–128 BAJTY	2 BAJTY	1 BAJT
Kodowanie	ASCII	ASCII	ASCII	ASCII	ASCII, UTF-8, HEX *	HEX **	ASCII

Tabela 2: Ramka protokołu

6.2 Opis ramki

Założenia:

- Minimalna długość ramki wynosi 10 Bajtów.
- Maksymalna długość ramki przed zakodowaniem to 135 Bajtów.
- Maksymalna przewidziana długość ramki po zakodowaniu to 270 Bajty.

Opis pól ramki:

- Początek ramki ustalony jest jako znak '~'. Tym znakiem rozpoczynamy wypełnianie ramki. Wartość heksadecymalna tego znaku wynosi 0x7E.
- Nadawca jest to pole w którym umieszczamy adres urządzenia z którego nadajemy ramkę. W przypadku mojego projektu jest to PC a znak ustalony jest na 'g' (0x67).
- Odbiorca jest to pole w który zostało określone urządzenie odbierające ramkę. W przypadku tego projektu jest to STM32 a jego adres został ustalony na 'h' (0x68).
- Pole komendy służy do obsługi wyświetlacza. Każda z komend ma stałą długość 3 bajtów i składa się ze znaków ASCII. Komendy oraz wartości przekazywane są opisane po opisie ramki.
- Pole danych jest przeznaczone do przesyłania danych, które są potrzebne do wywołanej komendy. Jest to skrót myślowy, ponieważ dane można przesyłać bez komendy i tak samo komendy bez danych, lecz grozi to odrzuceniem ramki zgodnie z założeniami projektu protokołu. Znaki zapisujemy w kodzie ASCII natomiast wartości liczbowe będziemy zapisywać jako liczby heksadecymalne oraz dane będą oddzielane znakiem ',' dla poprawnej detekcji wysyłanych danych, lecz nie dajemy znaku przecinka przed rozpoczęciem pisania danych oraz po ostatnim zmiennej w polu danych.

* Zapisałem ASCII, UTF-8 oraz HEX, chociaż wystarczyłoby UTF-8 oraz HEX ale chciałbym wyszczególnić parę zależności z tym związanych (aby użytkownik miał jasność). Generalnie kodowanie UTF-8 jest wstecznie kompatybilne z ASCII, ponieważ wartości znaków nie rozszerzonych są takie same.

- Dla znaków ASCII (0x00-0x7F): Używa dokładnie tych samych kodów co ASCII
Zapisywane jako jeden bajt Np.: 'A' = 0x41 (tak samo jak w ASCII)

- Dla znaków spoza ASCII (np. polskie znaki): Używa sekwencji wielobajtowych (2 bajty) Np.: 'Ż' = 0xC5 0xBB (dwa bajty)

Od razu tutaj zaznaczę, że wszystkie znaki ASCII możemy zapisywać normalnie jako znak, natomiast znaki rozszerzone musimy zapisywać już jako dwa bajty, ponieważ wynika to z możliwości terminala jak i naszego UARTA, ponieważ jak to mamy na uwadze, wysyłamy wartości 1-bajtowe. Później w programie będzie to odpowiednio łączone. Jeżeli chodzi o kolor to również kodujemy to jako dwa bajty odpowiednio po przecinkach. Zrobiłem tak, ponieważ daje to użytkownikowi pełną paletę barw RGB565. Kolor zapisujemy w notacji **BIG-ENDIAN**. Np. kolor czerwony zapiszemy normalnie 0xF8,0x00 Wartości, które są w systemie decymalnym również zapisujemy jako 1-bajtowe wartości heksadecymalne. Liczbę 150 zapiszemy jako 0x96.

- CRC (ang. Cyclic Redundancy Code). Jest to pole, które odpowiada za sprawdzanie poprawności przesyłanych danych w ramce (czy kolejność danych została zachowana przy przesłanej ramce oraz czy niczego nie zabrakło). Przygotowując ramkę do wysłania użytkownik powinien obliczyć CRC z pól nadawcy, odbiorcy, komendy oraz danych i dodać do ramki w formie 2 wartości heksadecymalnych odpowiadających obliczonemu CRC. Odbiorca ma za zadanie odebrać ramkę, sprawdzić początkową jej poprawność sprawdzając czy występuje początek oraz koniec ramki i następnie sprawdza jeszcze czy CRC obliczone przez użytkownika pokrywa się z CRC obliczonym przez odbiorcę. CRC jakie tutaj wykorzystuje odbiorca to CRC-16/IBM-3740 a obliczenia są dokonywane za pomocą tablicy "look-up", która to posiada u wcześnie obliczone wartości dla każdego bajtu. Jest to usprawnienie obliczania CRC. Warto również wspomnieć, że dzielnikiem w tym przypadku jest wielomian 0x1021, a wartość początkowa wynosi 0xFFFF.

! Co ważne, aby obliczyć CRC poprawnie należy zamienić wszystkie znaki ASCII na odpowiedniki bajtowe.

The screenshot shows a web-based CRC calculator interface. At the top, there is a large input field containing a long hex string: 0x670x680x4F0x4E0x4E0x000x2C0x000x2C0x030x2C0x960x2C0xF80x2C0x000x2C0xC50xBB0xC30x930xC50x810x540x59. Below this, there are two rows of radio buttons for selecting the input and output formats: 'Input: ASCII' (unchecked), 'HEX' (checked), 'DEC' (unchecked), 'OCT' (unchecked), 'BIN' (unchecked); and 'Output: HEX' (checked), 'DEC' (unchecked), 'OCT' (unchecked), 'BIN' (unchecked). There is also a checkbox 'Show processed data (HEX)' which is unchecked. In the center, there are two buttons: 'Back to all algos' and 'CRC-16/IBM-3740'. Below these buttons, there is a small note: 'My friend's Telegram channel with cute content for every day.' Underneath the buttons, there is a table with the following data:

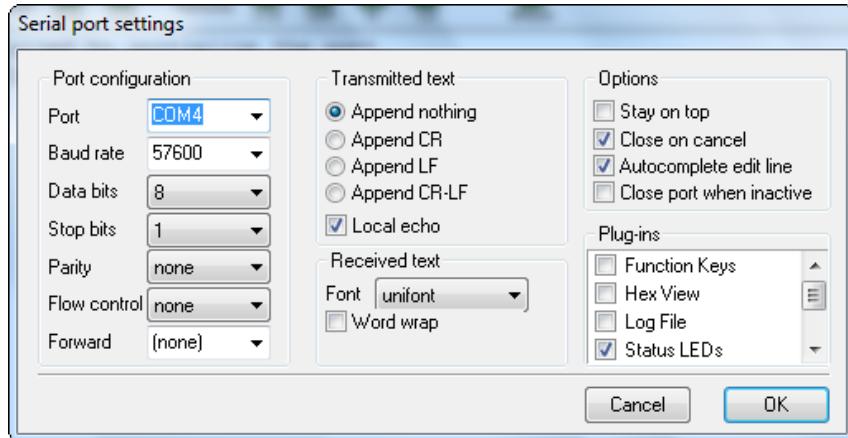
CRC-16/IBM-3740	Result	Check	Poly	Init	RefIn	RefOut	XorOut
CRC-16/IBM-3740 CRC-16/AUTOSAR CRC-16/CCITT-FALSE	0x2423	0x29B1	0x1021	0xFFFF	false	false	0x0000

Rysunek 9: Reprezentacja ramki w bajtach

Przykład obliczania CRC dla ramki:

~ghONN0x00,0x00,0x03,0x96,0xF8,0x00,0xC50xBB0xC30x930xC50x81TY0x240x23‘

** CRC-16 zapisujemy jako dwie wartości 8 bitowe z obliczonego CRC 16 bitowego. Korzystamy tutaj z notacji Big Endian, czyli jeżeli mamy obliczone CRC-16 jako 0x11F0 to kodujemy to jako 0x11 0xF0 (w ramce już bez spacji) i wysyłamy w ramce. Aby wysyłać wartości heksadecymalne można użyć terminala pt. "Termite", który obsługuje wysyłanie takich danych po włączeniu opcji HexView.



Rysunek 10: Widok ustawień terminala Termite

Ustawiamy odpowiednią konfigurację dla naszego STM'a i włączamy Hex View. Najlepiej włączyć tylko wysyłanie, ponieważ jest też opcja zwrotna, ale przez nią bardzo dużo pojawia się na terminalu i ciężko jest odczytać komunikaty.

! CRC ZAPISUJEMY W NOTACJI BIG-ENDIAN

- Koniec ramki ustalony jest jako znak " `` ". Który w systemie heksadecymalnym wynosi 0x60.

Warto od razu nadmienić, że może wystąpić sytuacja, gdy będziemy zmuszeni w ramce wpisać znak początku lub końca ramki. Wtedy program odbierający mógłby pomyśleć, że początek lub koniec jest tam, gdzie go nie powinno być, dlatego realizujemy byte stuffing, który objawia się następującym działaniem:

- zmień `` na '}' oraz dodaj '^',
 - zmień ' ' na '}' oraz dodaj '&',
- Gdy w ramce znajdzie się '}' to:
- dodaj ']'

Jeżeli chcemy odpowiednik na bajty to będzie wyglądać następująco:

- zmień '0x7E' na '0x5D' oraz dodaj '0x5E',
 - zmień '0x60' na '0x5D' oraz dodaj '0x26',
- Gdy w ramce znajdzie się '0x7D' to:
- dodaj '0x5D'

6.3 Komendy

Trzymając się specyfikacji, komendy mają pozwalać na wyświetlanie: koła, prostokąta, trójkąta - po zadanych parametrach, wypełnionych bądź nie. Zdefiniować trzy czcionki o różnych roz-

miarach. Umożliwić wyświetlanie napisu w zdefiniowanym oknie, z możliwością przewijania zadaną prędkością.

- ONK - komenda pozwalająca na wyświetlenie koła. Przyjmuje ona takie argumenty jak (x, y, r, color, filling). Pierwsze 3 argumenty to wymiary koła, jaki chcemy narysować na wyświetlaczu. Następnie color jest wartością, która jest możliwa do wyboru z listy, która będzie przedstawiona po opisie komend. Ostatni atrybut to atrybut filling, który będzie informował, czy koło ma być wypełnione czy nie.

uint8_t x	1 B
uint8_t y	1 B
uint8_t r	1 B
uint8_t filling	1 B
uint16_t color	2 B
Suma komendy	6 B
Z przecinkami	11 B

Tabela 3: Wielkość ONK

- ONP - komenda, która umożliwia wyświetlenie prostokąta. Przyjmuje takie argumenty jak (x, y, width, height, color, filling). Analogicznie jak do pierwszej komendy 4 pierwsze atrybuty dają informację jakich parametrów ma być prostokąt. Następne atrybuty określają dokładnie to samo co w komendzie ONK.

uint8_t x	1 B
uint8_t y	1 B
uint8_t width	1 B
uint8_t height	1 B
uint8_t filling	1 B
uint16_t color	2 B
Suma komendy	7 B
Z przecinkami	13 B

Tabela 4: Wielkość ONP

- ONT - komenda, umożliwiająca stworzenie trójkąta i jego wyświetlenie. Atrybuty to (x1, y1, x2, y2, x3, y3, color, filling). Pierwsze 6 atrybutów określają wierzchołki trójkąta, następnie color i filling będzie określało to samo co u poprzednich komend.

uint8_t x1	1 B
uint8_t y1	1 B
uint8_t x2	1 B
uint8_t y2	1 B
uint8_t x3	1 B
uint8_t y3	1 B
uint8_t filling	1 B
uint16_t color	2 B
Suma komendy	9 B
Z przecinkami	17 B

Tabela 5: Wielkość ONT

- ONN - komenda umożliwiająca wypisanie napisu na wyświetlaczu. Atrybuty, które przyjmuje to (x, y, text, fontSize, speed). Pierwsze dwa atrybuty to współrzędne od których zaczniemy wyświetlać nasz tekst. Atrybut text jak już sama nazwa wskazuje będzie tekstem wpisanym przez użytkownika. Atrybut fontSize będzie wyborem pomiędzy 1 - 3 predefiniowanej czcionki, które wynoszą po kolei 5x7, 5x8 oraz 6x9. Obliczenia do wyświetlacza:

Dla czcionek o wymiarach:

- Czcionka 1: 5×7 pikseli,
- Czcionka 2: 5×8 pikseli,
- Czcionka 3: 6×9 pikseli,

chcemy obliczyć:

1. Liczbę znaków mieszczących się w poziomie.
2. Liczbę znaków mieszczących się w pionie.
3. Całkowitą liczbę znaków na ekranie.

Czcionka 1 (5×7)

$$\text{Znaki w poziomie: } \left\lfloor \frac{128}{5} \right\rfloor = 25, \quad \text{Znaki w pionie: } \left\lfloor \frac{160}{7} \right\rfloor = 22$$

$$\text{Całkowita liczba znaków: } 25 \times 22 = 550$$

Czcionka 2 (5×8)

$$\text{Znaki w poziomie: } \left\lfloor \frac{128}{5} \right\rfloor = 25, \quad \text{Znaki w pionie: } \left\lfloor \frac{160}{8} \right\rfloor = 20$$

$$\text{Całkowita liczba znaków: } 25 \times 20 = 500$$

Czcionka 3 (6×9)

$$\text{Znaki w poziomie: } \left\lfloor \frac{128}{6} \right\rfloor = 21, \quad \text{Znaki w pionie: } \left\lfloor \frac{160}{9} \right\rfloor = 17$$

$$\text{Całkowita liczba znaków: } 21 \times 17 = 357$$

Dla użytkownika przeznaczam wyświetlanie tylko znaków w poziomie, dlatego ma możliwość wyświetlania 25 znaków, w trybie przewijania ma możliwość wyświetlania 50 znaków. Z możliwością na wyświetlenie polskich znaków, ponieważ biblioteka oraz wchar_t na to pozwala.

uint8_t x	1 B
uint8_t y	1 B
uint8_t fontSize	1 B
uint8_t speed	1 B
uint16_t color	2 B
uint8_t text	50/100 B
Suma komendy	56 / 106 B
Z przecinkami	62 / 112 B

Tabela 6: Wielkość ONN

- OFF - komenda umożliwiająca użytkownikowi reset lub wyłączenie wyświetlacza (wyłączenie podświetlania i reset) w razie błędów. Komenda ta przyjmuje jeden atrybut state i oznacza tylko tyle, że gdy użytkownik wyśle 0, to oznacza wyłączenie wyświetlacza, a gdy 1, to resetuje wyświetlacz.

uint8_t state	1 B
---------------	-----

Tabela 7: Wielkość OFF

Dostępne kolorystyki:

Kolory RGB565

Przedstawienie tabeli ASCII oraz przykład ramki

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	,	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	-	127	7F	[DEL.]

Rysunek 11: Tabela wartości

Polskie znaki w UTF-8

- Małe polskie znaki:

- a (0xC4 0x85)
- á (0xC4 0x87)
- é (0xC4 0x99)
- í (0xC5 0x82)
- ñ (0xC5 0x84)
- ó (0xC3 0xB3)
- š (0xC5 0x9B)
- ž (0xC5 0xBA)
- ż (0xC5 0xBC)

- Wielkie polskie znaki:

- Á (0xC4 0x84)
- Č (0xC4 0x86)
- É (0xC4 0x98)
- Ł (0xC5 0x81)
- Ñ (0xC5 0x83)
- Ó (0xC3 0x93)
- Š (0xC5 0x9A)
- Ž (0xC5 0xB9)
- Ż (0xC5 0xBB)

Poniżej przykładowo napisana ramka, wyświetla ona tekst "WITAM SERDECZNIE" o kolorze czerwonym i prędkości przewijania 150 oraz czcionce nr. 3, czyli 6x9. Współrzędne rozpoczęcia tekstu to (0,0):

gh0NN0x00,0x00,0x03,0x96,0x00,0xF8,WITAM SERDECZNIE0x6C0x69

oczywiście trzeba dodać znak początku oraz końca ramki.

6.4 Komunikaty zwrotne:

Komunikaty zwrotne mają za zadanie informować użytkownika o przebiegu wysłanej ramki bądź wykonywanego działania. Ramka zwrotna będzie charakteryzowała się tym, że w polu komendy ujrzymy ciąg znaków 'BCK' a w polu danych odpowiedni komunikat przeznaczony do napotkanej sytuacji.

Zacznijmy od komunikatów zwrotnych przy odbiorze ramki.

- GOOD - ramka została poprawnie odebrana
- FAIL - ramka nie została poprawnie odebrana
- WRONG_SENDER - nadawca jest nieprawidłowy
- WRONG_CRC - suma kontrolna obliczona przez użytkownika nie zgadza się z sumą kontrolną obliczoną przez program

Teraz rozpatrzmy komunikaty zwrotne dla komend służących do obsługi wyświetlacza:

- DISPLAY_AREA - komunikat informujący o współrzędnych, które są poza obszarem wyświetlacza
- WRONG_OFF_DATA - niepoprawna wartość dla komendy OFF
- WRONG_DATA - komunikat informujący o tym, że podane dane do stworzenia figury są niepoprawne
- INVALID_TRIANGLE - z podanych parametrów nie może powstać trójkąt
- TOO MUCH_TEXT - komunikat informująca, że użytkownik przekroczył ilość dozwolonych znaków dla wybranej wielkości czcionki.
- NOT_RECOGNIZED - komunikat informujący o niekorzystnym przebiegu detekcji podanych wartości.

7 Działanie oraz wykonanie projektu

7.1 Bufor kołowy

Bufor kołowy, który został wykorzystany, pochodzi z wykładu. Został on jedynie przerobiony tak, aby był sterowany za pomocą struktury. Struktura posiada w sobie wskaźnik na bufor, indeks zapisu oraz odczytu, oraz maskę, która pilnuje indeksów.

```
1 typedef struct {
2     uint8_t* buffer;
3     uint32_t readIndex;
4     uint32_t writeIndex;
5     uint32_t mask;
6 }ring_buffer;
```

Pełna inicjalizacja buforów następuje dopiero w funkcji ringBufferSetup(), gdzie przypisujemy wskaźnik na bufor, inicjalizujemy zmienne na 0 oraz ustawiamy maskę na wielkość bufora.

```
1 void ringBufferSetup(ring_buffer* rb, uint8_t* buffer, uint32_t size)
2 {
3     rb->buffer = buffer;
4     rb->readIndex = 0;
5     rb->writeIndex = 0;
6     rb->mask = size;
7 }
```

Bufor u mnie akurat nie jest wielkością 2, dlatego nie ma tutaj size - 1. Gdyby był potęgą 2, to wtedy byśmy mogli wykonać size - 1 i kontrolować wskaźnik bufora za pomocą operacji AND (&). Teraz niestety będziemy operować na operacji modulo (%). Poniżej przykład jak to wygląda:

- Operacja AND

```
1 #define BUFFER_SIZE 8      // Rozmiar bufora (potęga 2)
2 #define BUFFER_MASK (BUFFER_SIZE - 1) // Maska do operacji AND (0
3                                b0111 = 7)

4 uint8_t buffer[BUFFER_SIZE];
5 uint8_t head = 0;    // Indeks zapisu
6 uint8_t tail = 0;    // Indeks odczytu
7
8 // Operacja modulo przy użyciu AND
9 head = (head + 1) & BUFFER_MASK;

10 /* Przykład działania:
11 Gdy head = 7, następna pozycja:
12 (7 + 1) & 7 = 8 & 7 = 0b1000 & 0b0111 = 0b0000 = 0
13
14 Binarnie:
15 7 + 1 = 8      -> 0b1000
16 MASK = 7       -> 0b0111
17 8 & 7          -> 0b1000 & 0b0111 = 0b0000 (0)
18 */
19
```

Tutaj mamy operacje AND. Gdy spojrzymy na operacje bitowe, wszystko powinno być jasne dlaczego to działa.

- Operacja modulo

```
1 #define BUFFER_SIZE 10      // Rozmiar bufora (nie potęga 2)
2
3 uint8_t buffer[BUFFER_SIZE];
4 uint8_t head = 0;
5 uint8_t tail = 0;
6
7 // Musimy użyć operacji modulo
8 head = (head + 1) % BUFFER_SIZE;
9
10 /* Przykład działania:
11 Gdy head = 9, następna pozycja:
12 (9 + 1) % 10 = 10 % 10 = 0
13 */
14
```

Tutaj mamy zwykłą operację modulo, więc myślę, że z tym również nie ma problemów. Jest to dzielenie bez reszty.

Reszta kodu już służy do obsługi wysyłania danych, ale do tego przejdziemy za chwilę. Na tę chwilę zajmijmy się przerwaniami wywoływanymi przez USART.

```
1 void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart){
2     if(huart==&huart2){
3         if(txRingBuffer.writeIndex!=txRingBuffer.readIndex){
4             uint8_t tmp = USART_TxBuf[txRingBuffer.readIndex];
5             txRingBuffer.readIndex = (txRingBuffer.readIndex + 1) %
6             txRingBuffer.mask;
6             HAL_UART_Transmit_IT(&huart2, &tmp, 1);
7         }
8     }
9 }
10 void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart){
11     if(huart==&huart2){
12         rxRingBuffer.writeIndex = (rxRingBuffer.writeIndex + 1) %
13         rxRingBuffer.mask;
14         HAL_UART_Receive_IT(&huart2,&USART_RxBuf[rxRingBuffer.writeIndex],1);
15     }
16 }
```

Omówmy sobie ten kod:

- TxCpltCallback

Funkcja ta jest wywoływana po zakończeniu wysyłania poprzedniego bajtu. Sprawdza ona czy jest to usart2, ponieważ możemy mieć ich więcej na pokładzie STM32. Sprawdza jeszcze, czy są jeszcze dane do wysyłania (czy indeksy zapisu i odczytu są różne). Jeżeli tak to pobiera kolejny bajt z bufora, aktualizuje indeks odczytu oraz wysyła bajt.

- RxCpltCallback

Funkcja jest wywoływana po odebraniu bajtu danych. Oczywiście sprawdza czy usart jest prawidłowy oraz aktualizuje indeks zapisu w buforze kołowym. Po tym rozpoczyna oczekивание na kolejny bajt już w nowej pozycji bufora.

Tak jak już wcześniej wspominałem, możemy zauważać zmianę indeksów z pomocą działania modulo, aby w razie czego bufor zaczął od indeksu zerowego i nadpisywał dane.

Żeby korzystać z kodu trzeba również zainicjalizować odpowiednie rzeczy. Ja to robię przy okazji funkcji MX_USART2_UART_Init(), gdzie pojawiają się odpowiednie linijki:

```
1 // inicjalizacja buforu odbiorczego
2 ringBufferSetup(&rxRingBuffer, USART_RxBuf, RX_BUFFER_SIZE);
3 // inicjalizacja buforu nadawczego
4 ringBufferSetup(&txRingBuffer, USART_TxBuf, TX_BUFFER_SIZE);
5 // włączenie przerwań
6 HAL_UART_Receive_IT(&huart2, &USART_RxBuf[0], 1);
```

No dobrze, omówiliśmy już dosyć sporo funkcji, które służą do komunikacji PC -> STM32 oraz STM32 -> PC lecz to jeszcze nie koniec. Zostały nam 3 i to bardzo ważne funkcje, które akurat już wchodzą w tematy odbioru ramki.

Pierwszą funkcją z pozostałej 3 jest USART_kbhit(), jest to funkcja, która sprawdza, czy w buforze odbiorczym znajdują się dane.

```
1 uint8_t USART_kbhit(){
2     if(rxRingBuffer.writeIndex == rxRingBuffer.readIndex){
3         return 0;
4     }else{
5         return 1;
6     }
7 }
```

Następną funkcją jest USART_getchar(). Móglbym powiedzieć, że jest to funkcja kluczowa przy odbiorze ramki, ponieważ to dzięki niej jesteśmy w stanie odebrać wysłany do nas bajt oraz zapisać do bufora pomocniczego.

```
1 int16_t USART_getchar() {
2     if (rxRingBuffer.writeIndex != rxRingBuffer.readIndex) {
3         int16_t tmp = USART_RxBuf[rxRingBuffer.readIndex];
4         rxRingBuffer.readIndex = (rxRingBuffer.readIndex + 1) %
5             rxRingBuffer.mask;
5         return tmp;
6     }
7     return -1;
8 }
```

Funkcja zwraca liczbę 16-bitową ze znakiem. Sprawdza na początku dostępność danych. Gdy są one dostępne, to odczytuje i zapisuje do zmiennej tmp, później mamy już tylko przesunięcie indeksu odczytu o jeden.

I już ostatnia funkcja nazywa się USART_SendFrame(). Poziom skomplikowania ma najwyższy ze wszystkich.

```
1 void USART_sendFrame(const uint8_t* data, size_t length) {
2     int idx = txRingBuffer.writeIndex;
3
4     __disable_irq();
5     for(size_t i = 0; i < length; i++) {
6         USART_TxBuf[idx] = data[i];
7         idx = (idx + 1) % txRingBuffer.mask;
8     }
9
10    if((txRingBuffer.writeIndex == txRingBuffer.readIndex) &&
11        (_HAL_UART_GET_FLAG(&huart2, UART_FLAG_TXE) == SET)) {
12        txRingBuffer.writeIndex = idx;
13        uint8_t tmp = USART_TxBuf[txRingBuffer.readIndex];
14        txRingBuffer.readIndex = (txRingBuffer.readIndex + 1) %
15        txRingBuffer.mask;
16        HAL_UART_Transmit_IT(&huart2, &tmp, 1);
17    } else {
18        txRingBuffer.writeIndex = idx;
19    }
20
21    __enable_irq();
}
```

Do funkcji przekazujemy wskaźnik na tablicę danych do wysłania oraz jej długość. Na samym początku następuje inicjalizacja lokalnej kopii indeksu, po której możemy zauważyc linijkę `__disable_irq()`. Ona powoduje nam to, że wyłączaemy wszystkie przerwania oraz zapewniamy atomowość operacji na buforze (ochrona przed jednoczesnym dostępem). Po wyłączeniu przerwań następuje przekopiowanie danych do bufora nadawczego. Następnie mamy sprawdzenie czy bufor jest pusty oraz czy USART jest gotowy do transmisji (flaga TXE - Transmit Data Register Empty). Jeżeli tak to aktualizujemy indeks zapisu, pobieramy pierwszy bajt do wysłania i przesuwamy indeks. Rozpoczynamy transmisję w trybie przerwań. Jeżeli któryś warunek by się nie spełnił to tylko aktualizuje się indeks zapisu i transmisja będzie kontynuowana przez przerwania. Na koniec przywracamy obsługę przerwań `__enable_irq()`.

7.1.1 Odbiór ramki

Odbiór ramki został zrealizowany poprzez maszynę stanów. Odpowiednio po wykryciu danego znaku przechodziliśmy do konkretnej czynności i tak aż do znaku końca ramki.

```
1 static void resetFrameState() {
2     inFrame = false;
3     escapeDetected = false;
4     bxIndex = 0;
5 }
6
7 static void stopAnimation(void)
{
8     text.isScrolling = false;
9 }
10
11 void processReceivedChar(uint8_t receivedChar) {
12     if (receivedChar == FRAME_START) {
13         resetFrameState();
```

```

15     inFrame = true;
16 } else if (receivedChar == FRAME_END && escapeDetected == false) {
17     if (inFrame) {
18         if (decodeFrame(bx, &frame, bxIndex)) {
19             stopAnimation();
20             sendStatus(ERR_GOOD);
21             handleCommand(&frame);
22         } else {
23             sendStatus(ERR_FAIL);
24         }
25         resetFrameState();
26         return;
27     }
28 } else if (inFrame) {
29     if(bxIndex <= MAX_FRAME_WITHOUT_STUFFING){
30         if (escapeDetected) {
31             if (receivedChar == FRAME_START_STUFF) {
32                 bx[bxIndex++] = FRAME_START;
33             } else if (receivedChar == ESCAPE_CHAR_STUFF) {
34                 bx[bxIndex++] = ESCAPE_CHAR;
35             } else if (receivedChar == FRAME_END_STUFF) {
36                 bx[bxIndex++] = FRAME_END;
37             } else {
38                 sendStatus(ERR_FAIL);
39                 resetFrameState();
40             }
41             escapeDetected = false;
42         } else if (receivedChar == ESCAPE_CHAR) {
43             escapeDetected = true;
44         } else {
45             bx[bxIndex++] = receivedChar;
46         }
47     } else {
48         resetFrameState();
49     }
50 }
51 }
```

W programie wykorzystuję kilka zmiennych globalnych:

- inFrame - flaga oznaczająca czy jesteśmy w ramce
- bx - bufro na dane ramki
- bxIndex - indeks w buforze
- escapeDetected - flaga wykrycia znaku escape
- frame - struktura przechowująca zakodowaną ramkę

Gdy odbierze znak początku ramki, to ustawiamy flagę inFrame na true, resetujemy indeks oraz ustawiamy flagę escapeDetected na false. Jeżeli zdarzy się tak, że odbierzemy znak początku ramki i ponownie natrafimy na początek, to resetujemy indeksy i ponownie ustawiamy flagę na odbiór ramki.

W trakcie odbierania ramki musimy te dane odebrać i rozkodować jeżeli zajdzie taka potrzeba. Do tego jest cały warunek else if(in_frame). To tam następuje sprawdzenie, czy ramka nie jest

za długa oraz czy otrzymany znak nie jest znakiem ucieczki. Jeżeli jest to odpowiednio, to dekodujemy. Jeżeli nie ma żadnych "problemów" ze znakiem, to zapisujemy prosto do bufora.

Co do długości, to chodzi głównie o to, że długość niezakodowanej ramki to 135 znaków zgodnie z protokołem komunikacyjnym, lecz zakładając, że ktoś będzie chciał wysłać całą ramkę zakodowaną, to wyjdzie dwukrotność ramki niezakodowanej. Oczywiście ta ramka nic nie odpali oraz nie ma żadnych danych zgodnych z protokołem, ale musimy ją przyjąć i przetworzyć, bo jeżeli jest ona po odkodowaniu poprawnej wielkości, to nie możemy jej odrzucić. Dopiero później, przy okazji przetwarzania ramki.

Jeżeli natrafimy na koniec ramki, to próbujemy przetworzyć ramkę w funkcji decodeFrame(), która będzie później ukazana. Następnie wysyłamy ramkę zwrotną, że dekodowanie się powiodło i przetwarzamy komendę w funkcji handleCommand().

W tej funkcji mamy zachowane bezpieczeństwo poprzez sprawdzanie rozmiaru bufora, obsługę nieprawidłowych sekwencji oraz resetowanie stanu w przypadku błędów.

Przejdźmy do funkcji decodeFrame().

```
1 static bool decodeFrame(uint8_t *bx, Frame *frame, uint8_t len) {
2     uint8_t ownCrc[2];
3     uint8_t incCrc[2];
4
5     if(len >= MIN_DECODED_FRAME_LEN && len <= MAX_FRAME_WITHOUT_STUFFING)
6     {
7         uint8_t k = 0;
8         frame->sender = bx[k++];
9         frame->receiver = bx[k++];
10        if(frame->sender != PC_ADDR)
11        {
12            sendStatus(ERR_WRONG_SENDER);
13            return false;
14        }
15
16        memcpy(frame->command, &bx[k], COMMAND_LENGTH);
17        k += COMMAND_LENGTH;
18
19        uint8_t dataLen = len - MIN_DECODED_FRAME_LEN; //to sprawdzić
20        memcpy(frame->data, &bx[k], dataLen);
21        k += dataLen;
22
23        memcpy(incCrc, &bx[k], 2);
24        calculateCrc16((uint8_t *)frame, k, ownCrc);
25        if(ownCrc[0] != incCrc[0] || ownCrc[1] != incCrc[1])
26        {
27            sendStatus(ERR_WRONG_CRC);
28            return false;
29        }
30        return true;
31    }
32    return false;
33}
```

W tej funkcji dużo komplikacji nie ma. Po prostu bierzemy nasz bufor pomocniczy, w którym jest zapisana ramka, i odpowiednio przydzielamy dane do struktury. Najpierw idzie przypisanie danych do nadawcy i odbiorcy, gdzie dodatkowo następuje sprawdzenie, czy nadawca jest prawidłowy. Następnie przypisujemy komendę, sprawdzamy wielkość danych i obliczamy dłu-

gość danych w ten sposób, że zdekodowana ramka bez pola danych jest długości 7, więc od całej długości ramki (bez początku i końca) odejmujemy długość 7. Kopiujemy dane i sprawdzamy CRC. Jeżeli CRC się zgadza, to ramka przechodzi dalej do wykonania komendy (to jest w funkcji poprzedniej).

Jak już jesteśmy przy CRC to spójrzmy na kod.

```

1 void calculateCrc16(uint8_t *data, size_t length, char crc_out[2]) {
2     uint16_t crc = 0xFFFF;
3
4     for (size_t i = 0; i < length; i++) {
5         uint8_t byte = data[i];
6         uint8_t table_index = (crc >> 8) ^ byte;
7         crc = (crc << 8) ^ crc16_table[table_index];
8     }
9     crc_out[0] = ((crc >> 8) & 0xFF);
10    crc_out[1] = (crc & 0xFF);
11 }
```

Do obliczeń jest wykorzystywana tablica look-up table. Nie wklejam jej bo każdy może sobie zobaczyć, jeżeli wejdzie np. na stronę CRC Online i wybierze CRC-16/IBM-3740. Tutaj działanie wygląda tak, że pętla przechodzi przez każdy bajt danych, wyciąga go oraz oblicza indeks tabeli. Jest on obliczany poprzez przesunięcie CRC o 8 bitów w prawo i wykonanie operacji XOR z bieżącym bajtem. Wartość CRC jest aktualizowana w taki sposób, że CRC przesuwamy o 8 bitów w lewo i wykonywana jest operacja XOR z wartością z tabeli crc16_table dla obliczonego indeksu.

Jak już idziemy po kolej z kodem, to przejdźmy już do ostatniej ważnej funkcji w dekodowaniu ramki - handleCommand().

```

1 static bool isWithinBounds(int x, int y)
2 {
3     return (x >= 0 && x < LCD_WIDTH) && (y >= 0 && y < LCD_HEIGHT);
4 }
5
6
7 static bool safeCompare(const char* str1, const char* str2, size_t len)
8 {
9     if(str1 == NULL || str2 == NULL)
10    {
11        return false;
12    }
13    return memcmp(str1, str2, len) == 0;
14 }
15
16 void handleCommand(Frame *frame) {
17     if (frame == NULL) {
18         return;
19     }
20     CommandEntry commandTable[COMMAND_COUNT] = {
21         {"ONK", executeONK},
22         {"ONP", executeONP},
23         {"ONT", executeONT},
24         {"ONN", executeONN},
25         {"OFF", executeOFF}
26     };
27     HAL_GPIO_WritePin(BL_GPIO_Port, BL_Pin, GPIO_PIN_SET);
```

```

28     for (int i = 0; i < COMMAND_COUNT; i++) {
29         if (safeCompare(frame->command, commandTable[i].command,
30                         COMMAND_LENGTH)) {
31             if (safeCompare(commandTable[i].command, "OFF",
32                             COMMAND_LENGTH)) {
33                 lcdClear();
34                 commandTable[i].function(frame);
35                 lcdCopy();
36                 clearFrame(frame);
37                 return;
38             }
39
40             int x, y;
41             if (parseCoordinates(frame->data, &x, &y)) {
42                 if (isWithinBounds(x, y)) {
43                     lcdClear();
44                     commandTable[i].function(frame);
45                     lcdCopy();
46                     clearFrame(frame);
47                     return;
48                 } else {
49                    .sendStatus(ERR_DISPLAY_AREA);
50                     return;
51                 }
52             }
53         }
54     }
55 }
56 }
57 }
58 }
```

Tutaj dużego stopnia skomplikowania również nie ma, ponieważ przekazujemy ramkę, a następnie inicjalizujemy tablicę z komendami oraz funkcjami do nich przydzielonymi. Pętla przechodzi przez każdy element tablicy sprawdzając funkcją safeCompare() czy jakaś komenda jest równa którejś z tablicy. Funkcja safeCompare() nie ma nic wielkiego w swoim zastosowaniu. Jest tylko sprawdzenie czy przekazywane wskaźniki nie wskazują na NULL oraz zwracana jest funkcja memcmp(), która porównuje dwa obszary w pamięci. Dzięki temu jeżeli znajdzie komendę to sprawdza koordynaty czy mieszczą się na wyświetlaczu (funkcją isWithinBound()). Jako, że każda z komend x oraz y (oprócz OFF) ma argumenty w tym samym miejscu to została stworzona funkcja parseCoordinates, która przypisuje po prostu:

```

1 bool parseCoordinates(const uint8_t* data, int* x, int* y)
2 {
3     *x = data[0]; // Pierwszy bajt to x
4     *y = data[2]; // Drugi bajt to y
5     return true;
6 }
```

Widnieje tam data[2], ponieważ data[1] to byłby przecinek zgodnie z rozdzieleniem wartości w protokole.

I to właściwie tyle z odbioru ramki. Wiem, była to długa droga, która mogła wydawać się nudna i dosyć trudna, ale teraz możemy przejść do przyjemniejszych rzeczy, czyli rezultatów na wy-

świetlaczu. Oczywiście nie może być za łatwo i musimy jeszcze trochę rzeczy poustawić, ale umówmy się - jest bliżej niż dalej.

7.2 Wyświetlacz

Jak już wcześniej nadmieniłem, wyświetlacz jest wielkości 128 x 160 px oraz posiada sterownik ST7735. Ramki pozwalają na wyświetlenie figury (wypełnionej bądź nie) koła, prostokąta, trójkąta oraz wyświetlenie tekstu, co ukażę poniżej na fotografiach poglądowych, które nie będą zawierać wszystkich przypadków, ale na pewno pozwoli to uzmysłowić sobie działanie.



Rysunek 12: Ramka na wypełnione koło



Rysunek 13: Ramka na trójkąt



Rysunek 14: Ramka na prostokąt



Rysunek 15: Tekst statyczny



Rysunek 16: Tekst z polskimi znakami



Rysunek 17: Przewijanie tekstu

Jak widać, aparat był w stanie uchwycić moment przesuwania się tekstu po wyświetlaczu (moment wygaszania i zapalenia kolejnych pozycji), ludzkim okiem jest to trochę ciężkie do za-uważenia, ponieważ umysł wypełnia puste klatki.

Niestety, aby w ogóle przystąpić do wyświetlania czegokolwiek, najpierw musimy zapoznać się z funkcjami ustawiającymi wyświetlacz. Wcześniej trochę wspominałem o tym, że nasz sterownik wyświetlacza ma swoje rejestry do których musimy zapisać odpowiednie wartości. W kodzie jest taka tablica inicjalizacyjna, myślę, że tylko ją wy tłumaczymy i wszystko powinno być jasne. Reszta to już tylko ustawienie odpowiednich pinów lub też odpowiednie operacje bitowe.

```
1 static const uint16_t init_table[] = {  
2     CMD(ST7735S_FRMCTR1), 0x01, 0x2c, 0x2d,  
3     CMD(ST7735S_FRMCTR2), 0x01, 0x2c, 0x2d,  
4     CMD(ST7735S_FRMCTR3), 0x01, 0x2c, 0x2d, 0x01, 0x2c, 0x2d,
```

```

5   CMD(ST7735S_INVCTR), 0x07,
6   CMD(ST7735S_PWCTR1), 0xa2, 0x02, 0x84,
7   CMD(ST7735S_PWCTR2), 0xc5,
8   CMD(ST7735S_PWCTR3), 0x0a, 0x00,
9   CMD(ST7735S_PWCTR4), 0x8a, 0x2a,
10  CMD(ST7735S_PWCTR5), 0x8a, 0xee,
11  CMD(ST7735S_VMCTR1), 0x0e,
12  CMD(ST7735S_GAMCTRP1), 0x0f, 0x1a, 0x0f, 0x18, 0x2f, 0x28, 0x20, 0
13  x22,
14  0x1f, 0x1b, 0x23, 0x37, 0x00, 0x07, 0x02, 0x10,
15  CMD(ST7735S_GAMCTRN1), 0x0f, 0x1b, 0x0f, 0x17, 0x33, 0x2c, 0x29, 0
16  x2e,
17  0x30, 0x30, 0x39, 0x3f, 0x00, 0x07, 0x03, 0x10,
18  CMD(0xf0), 0x01,
19  CMD(0xf6), 0x00,
CMD(ST7735S_COLMOD), 0x05,
CMD(ST7735S_MADCTL), 0xa0,
20 };
```

Nie będę wchodzić w duże szczegóły, ponieważ ta tablica inicjalizacyjna jest z kodu dołączanego do wyświetlacza przez producenta i wgłębianie się myślę, że odchodziłyby od tematu mikroprocesorów i tego projektu. No to zaczynamy:

- ST7735S_FRMCTR1. Przekazujemy tam wartości 0x01, 0x2C, 0x2D. Pierwsza z nich wpływa na częstotliwość odświeżania. Jest to wartość, która powoduje zbalansowaną częstotliwość, która jest stabilna i nie powoduje migania. Druga z nich oraz trzecia to wartości inwersji linii, aby zminimalizować zakłocenia obrazu i zapewnić stabilny obraz bez artefaktów.
- ST7745S_PWCTR1 Widzimy przekazywane wartości 0xA2, 0x02, 0x84. Pierwsza z nich ustawia AVDD na około 5V co jest optymalnym napięciem dla tego wyświetlacza (choćiąż my go zasilamy z lini 3V3). 0x02 Kontroluje GVDD (napięcie referencyjne gamma) a 0x84 ustawia VGH na około 14.7V (napięcie bramki).
- ST7745S_GAMCTRP1 Te 16 wartości, które możemy wyczytać tworzą krzywą gamma, która jest kluczowa dla jakości obrazu. Niższe wartości kontrolują ciemne tony, środkowe odpowiadają za średnie tonny a wyższe wartości za jasne tony. Wartości są dobrane tak aby zapewnić naturalne odwzorowanie kolorów, uniknąć "wypalania" jasnych obszarów oraz zachować szczegóły w ciemnych partiach obrazu.
- ST7735_COLMOD Tutaj ustawiamy nasz format koloru na RGB565. Przekazujemy 5 bitów na czerwony, 6 bitów na zielony oraz 5 bitów na niebieski. W ramach ciekawostki ludzkie oko najbardziej wrażliwe jest na kolor zielony.
- ST7735_MADCTL Wartość przekazywana konfiguruje kierunek skanowania, kolejność kolorów oraz orientację wyświetlacza.
- Może zastanawiać nas co to jest CMD(0xF0) oraz CMD(0xF6). Otóż pierwsza komenda służy do kontroli zestawu poleceń. Dzięki wysłanej wartości 0x01 włączamy rozszerzony zestaw poleceń. Druga to komenda kontroli interfejsu. Ustawiamy tutaj standardowy tryb interfejsu, włączamy specjalne tryby komunikacji oraz konfigurujemy standardowe parametry czasowe interfejsu.

Jest to tylko tablica inicjalizacyjna, do tego jest również funkcja lcd_init(). Przyjrzyjmy się jej.

```

1 void lcdInit(void) {
2     int i;
3     HAL_GPIO_WritePin(RST_GPIO_Port, RST_Pin, GPIO_PIN_RESET);
4     delay(100);
5     HAL_GPIO_WritePin(RST_GPIO_Port, RST_Pin, GPIO_PIN_SET);
6     delay(100);
7     for (i = 0; i < sizeof(init_table) / sizeof(uint16_t); i++) {
8         lcdSend(init_table[i]);
9     }
10    delay(200);
11    lcdCmd(ST7735S_SLPOUT);
12    delay(120);
13    lcdCmd(ST7735S_DISPON);
14    HAL_GPIO_WritePin(BL_GPIO_Port, BL_Pin, GPIO_PIN_SET);
15    memset(frameBuffer, 0, sizeof(frameBuffer));
16 }

```

Na samym początku wykonujemy reset sprzętowy, oczekujemy zalecany okres czasu przez producenta (pozwala wyświetlaczowi się ustabilizować). Pętla for służy do wysyłania sekwencji komend oraz wartości przesyłanych do komend. Po przesłaniu następuje włączenie wyświetlacza oraz włączenie podświetlenia. Na koniec jest czyszczony bufor wyświetlacza za pomocą funkcji memset.

Czym jest funkcja lcdSend()? Jest to dosyć krótka funkcja, lecz ona przydziela nam, które funkcje muszą zostać wykonane - funkcja od wysyłania komend czy może wartości.

```

1 static void lcdSend(uint16_t value)
2 {
3     if (value & 0x100) {
4         lcdCmd(value);
5     } else {
6         lcdData(value);
7     }
8 }

```

Funkcja sprawdza bit o numerze 8 (9 w kolejności). Jeżeli bit jest ustawiony na jedynkę, to wartość traktowana jest jako komenda i wysyłana przez lcdCmd(); jeżeli jest zerem, to jest traktowana jako dane i wysyłana przez lcdData(). Dzieje się tak, ponieważ jak wcześniej w tablicy można było zauważyc, komendy były w makrzu CMD.

```

1 #define CMD(x) ((x) | 0x100)

```

Tak ono wygląda. Dzieje się tak ponieważ robimy lekkie uproszczenie i przyjmujemy liczbę 16 bitową a dobrze nam wiadomo, że do wyświetlacza wysyłamy wartości 8 bitowe. Tutaj mniej znaczący bajt będzie zawierał wartość do wysłania a bardziej znaczący będzie określał rodzaj transmisiJI.

Po funkcji lcdSend() mamy już tylko lcdCmd() oraz lcdData() ale one robią to samo - kopiąją wartości do sterownika, tylko zgodnie z tym, czy jest to komenda, czy dane.

```

1 static void lcdCmd(uint8_t cmd)
2 {
3     HAL_GPIO_WritePin(DC_GPIO_Port, DC_Pin, GPIO_PIN_RESET);
4     HAL_GPIO_WritePin(CS_GPIO_Port, CS_Pin, GPIO_PIN_RESET);
5     HAL_SPI_Transmit(&hspi2, &cmd, 1, HAL_MAX_DELAY);

```

```

6   HAL_GPIO_WritePin(CS_GPIO_Port, CS_Pin, GPIO_PIN_SET);
7 }
8 static void lcdData(uint8_t data)
9 {
10  HAL_GPIO_WritePin(DC_GPIO_Port, DC_Pin, GPIO_PIN_SET);
11  HAL_GPIO_WritePin(CS_GPIO_Port, CS_Pin, GPIO_PIN_RESET);
12  HAL_SPI_Transmit(&hspi2, &data, 1, HAL_MAX_DELAY);
13  HAL_GPIO_WritePin(CS_GPIO_Port, CS_Pin, GPIO_PIN_SET);
14 }
```

Jak widać, funkcja różni się jedną subtelną linijką z pinem DC. W przypadku komendy jest ona w stanie RESET, a w przypadku danych jest w stanie SET.

I to tyle, jeżeli chodzi o fundamenty obsługi wyświetlacza. Reszta to przygotowanie funkcji, które będą obsługiwać wyświetalnie rzeczy. Można zrealizować to na kilka sposobów. Wybrałem pojedyńczy bufor, ponieważ nie posiadam skomplikowanych rzeczy do wyświetlania, a na odświeżaniu też mi nie zależy bardzo mocno. Można także zastosować podwójny bufor, wtedy rzeczy wyświetlane na ekranie będą o wiele płynniejsze tak, jakby w ogóle nie mijał czas do przesyłania danych. Ta metoda ma jedną wadę w moim przypadku - zajmuje 2 x 40kB Ramu a ja mam RAM wielkości 96kB. Drugi jest 32kB lecz napotkałem małe problemy z jego konfiguracją i dlatego zrezygnowałem.

Omówmy jeszcze kilka funkcji potrzebnych do wyświetlania. Jak już wcześniej gdzieś napisałem, korzystam z biblioteki HAGL dostępnej na serwisie github. Musimy ją dostosować pod siebie i zrobić kilka usprawnień. Na początek musimy stworzyć sobie funkcje, która będzie "klaść" piksele na ekran. Wtedy tą funkcję łączymy w pliku hagl_hal.h z funkcją hagl_hal_put_pixel i definiujemy uint16_t jako color_t.

```

1 #pragma once
2 #include "lcd.h"
3 #include "bitmap.h"
4 #define DISPLAY_WIDTH (LCD_WIDTH)
5 #define DISPLAY_HEIGHT (LCD_HEIGHT)
6 #define DISPLAY_DEPTH 16
7
8 typedef uint16_t color_t;
9
10 #define hagl_hal_put_pixel lcdPutPixel
```

```

1 void lcdPutPixel(int x, int y, uint16_t color)
2 {
3     frameBuffer[y * LCD_WIDTH + x] = color;
4 }
```

Wyrażenie $y * LCD_WIDTH + x$ oblicza pozycję w tablicy frameBuffer.

Jeżeli już wszystko się przekopiuje to wywołujemy funkcję lcdCopy(), która to sprawdza czy przypadkiem nie jesteśmy w trakcie przesyłania danych do wyświetlacza przez SPI. Wysłanie danych jest realizowane poprzez DMA w celu odciążenia procesora.

```

1 static bool lcdIsBusy(void) {
2     return transferInProgress;
3 }
4
5 void lcdCopy(void)
```

```

6  {
7      if (lcdIsBusy()) {
8          return;
9      }
10
11     lcdSetWindow(0, 0, LCD_WIDTH, LCD_HEIGHT);
12     lcdCmd(ST7735S_RAMWR);
13     HAL_GPIO_WritePin(DC_GPIO_Port, DC_Pin, GPIO_PIN_SET);
14     HAL_GPIO_WritePin(CS_GPIO_Port, CS_Pin, GPIO_PIN_RESET);
15     HAL_SPI_Transmit_DMA(&hspi2, (uint8_t*)frameBuffer, sizeof(
16     frameBuffer));
17 }
18 void HAL_SPI_TxCpltCallback(SPI_HandleTypeDef *hspi) {
19     if (hspi == &hspi2) {
20         HAL_GPIO_WritePin(CS_GPIO_Port, CS_Pin, GPIO_PIN_SET);
21         transferInProgress = false;
22     }
23 }
```

W przerwaniu ustawiamy pin CS na stan wysoki, ponieważ jest to sygnał, że transmisja się zakończyła. Flaga transferInProgress to akurat także flaga zakończenia, ale jest ona pomocna przy funkcjach.

Jest jeszcze ostatnia funkcja odnośnie LCD, ale to jest już ustawienie wymiarów okna.

```

1 static void lcdSetWindow(int x, int y, int width, int height)
2 {
3     lcdCmd(ST7735S_CASET);
4     lcdData16(LCD_OFFSET_X + x);
5     lcdData16(LCD_OFFSET_X + x + width - 1);
6
7     lcdCmd(ST7735S_RASET);
8     lcdData16(LCD_OFFSET_Y + y);
9     lcdData16(LCD_OFFSET_Y + y + height - 1);
10 }
```

Tak jak wcześniej przy wspominałem, musimy tutaj zastosować offset kilku pikseli, aby ustawić dobrze okno i współrzędne się zgadzały.

7.2.1 Wyświetlanie

W końcu możemy przejść do rezultatów. Była to długa droga. Przypomnę jak działa program.

Program działa tak, że użytkownik (ty) wybiera sobie odpowiednią komendę ze spisu komend, podaje także odpowiednie wartości przeznaczone dla komendy. Wszystko wysyła za pomocą ramki, opisanej w protokole komunikacyjnym. Zgodnie z ramką, najpierw sobie ją szukujemy, czyli:

- Nadawcę
- Odbiorcę
- Komendę
- Dane

Następnie liczymy z tego CRC, które następnie dołączamy do ramki i mamy postać:

- Nadawcę
- Odbiorcę
- Komendę
- Dane
- CRC

I na sam koniec zamykamy to za pomocą znaku początku oraz końca. Uprzednio sprawdzając czy w ramce już te znaki nie występują, bo jeżeli tak to należy je zakodować w celu poprawnego odebrania i zdekodowania ramki.

- Znak początku
- Nadawcę
- Odbiorcę
- Komendę
- Dane
- CRC
- Znak końca

Tak przygotowaną ramkę wysyłamy, a STM ją dekoduje, zmienia zakodowane dane w odpowiednią wiadomość i z niej liczy CRC. Gdy te wartości się zgadzają, następuje analiza ramki tzn. kto jest nadawcą, kto ma być odbiorcą, jaką komendę chce wywołać użytkownik oraz jakie dane przesłać. Gdy ramka jest już przeanalizowana, następuje wywołanie odpowiednich funkcji obsługujących komendę oraz przypisanie wartości. Rezultatem końcowym jest ujrzenie działania wyświetlacza dla danego zadania.

I po tym przypomnieniu możemy już przejść do ostatniej fazy programu - biblioteka HAGL oraz funkcje wykonawcze. Jak wcześniej zauważyłeś w funkcji handleCommand() była tablica z funkcjami i to właśnie były funkcje, które kryły w sobie funkcjonalności. Najbardziej rozbudowana jest funkcja od tekstu ponieważ rozbija się ona na dwie funkcje tak naprawdę. Jedna, która jest głównie do tekstu statycznego a druga do przewijania tekstu po wyświetlaczu i jest wykonywana w SysTickHandler(), ale zacznijmy omawiać wszystko od początku. Możliwe figury do wyświetlenia to:

- Koło
- Trójkąt
- Prostokąt

Realizacja tego zadania spoczywa na barkach biblioteki hagl a dokładnie na funkcjach hagl_draw_circle, hagl_draw_triangle oraz hagl_draw_rectangle. Wersje z wypełnieniami realizują funkcje o taki samym początku oraz końcu, ale środek z draw zmienia się na fill. Przyjrzyjmy na początek prostokątowi:

```
1 void hagl_draw_rectangle(int16_t x0, int16_t y0, int16_t x1, int16_t y1,  
2                           color_t color)  
3 {
```

```

3  /* Make sure x0 is smaller than x1. */
4  if (x0 > x1) {
5      x0 = x0 + x1;
6      x1 = x0 - x1;
7      x0 = x0 - x1;
8  }
9
10 /* Make sure y0 is smaller than y1. */
11 if (y0 > y1) {
12     y0 = y0 + y1;
13     y1 = y0 - y1;
14     y0 = y0 - y1;
15 }
16
17 /* x1 or y1 is before the edge, nothing to do. */
18 if ((x1 < clip_window.x0) || (y1 < clip_window.y0)) {
19     return;
20 }
21
22 /* x0 or y0 is after the edge, nothing to do. */
23 if ((x0 > clip_window.x1) || (y0 > clip_window.y1)) {
24     return;
25 }
26
27 uint16_t width = x1 - x0 + 1;
28 uint16_t height = y1 - y0 + 1;
29
30 hagl_draw_hline(x0, y0, width, color);
31 hagl_draw_hline(x0, y1, width, color);
32 hagl_draw_vline(x0, y0, height, color);
33 hagl_draw_vline(x1, y0, height, color);
34 }
35 void hagl_fill_rectangle(int16_t x0, int16_t y0, int16_t x1, int16_t y1,
36                         color_t color)
37 {
38     /* Make sure x0 is smaller than x1. */
39     if (x0 > x1) {
40         x0 = x0 + x1;
41         x1 = x0 - x1;
42         x0 = x0 - x1;
43     }
44
45     /* Make sure y0 is smaller than y1. */
46     if (y0 > y1) {
47         y0 = y0 + y1;
48         y1 = y0 - y1;
49         y0 = y0 - y1;
50     }
51
52     /* x1 or y1 is before the edge, nothing to do. */
53     if ((x1 < clip_window.x0) || (y1 < clip_window.y0)) {
54         return;
55     }
56
57     /* x0 or y0 is after the edge, nothing to do. */
58     if ((x0 > clip_window.x1) || (y0 > clip_window.y1)) {
59         return;
60     }

```

```

60
61     x0 = max(x0, clip_window.x0);
62     y0 = max(y0, clip_window.y0);
63     x1 = min(x1, clip_window.x1);
64     y1 = min(y1, clip_window.y1);
65
66     uint16_t width = x1 - x0 + 1;
67     uint16_t height = y1 - y0 + 1;
68
69     for (uint16_t i = 0; i < height; i++) {
70 #ifdef HAGL_HAS_HAL_HLINE
71         /* Already clipped so can call HAL directly. */
72         hagl_hal_hline(x0, y0 + i, width, color);
73 #else
74         hagl_draw_hline(x0, y0 + i, width, color);
75 #endif
76     }
77 }
```

Jak widać i jak wspominałem wcześniej są tutaj dwie funkcje. Pierwsza z nich rysuje tylko obrrys prostokąta. Na samym początku upenia się, że $x0 < x1$ oraz $y0 < y1$. To jest taki mały algorytm zamiany wartości bez użycia zmiennej tymczasowej. Później sprawdza czy prostokąt jest w ogóle widoczny i rysuje krawędzie za pomocą linii. Opowiednio po kolejno jest to górna poziomia, dolna poziomia, lewa pionowa oraz prawa pionowa.

Druga funkcja robi prawie to samo co pierwsza ale przy okazji drugiej mamy przycinanie współrzędnych do obszaru widocznego oraz wypełnia prostokąt rysując kolejne linie poziomie.

Funkcje korzystają również z innej funkcji hagl_draw_hline oraz hagl_draw_line. One będą się często powielać, dlatego wyjaśniamy sobie jak one działają:

```

1 void hagl_draw_hline(int16_t x0, int16_t y0, uint16_t w, color_t color)
2 {
3 #ifdef HAGL_HAS_HAL_HLINE
4     int16_t width = w;
5
6     /* x0 or y0 is over the edge, nothing to do. */
7     if ((x0 > clip_window.x1) || (y0 > clip_window.y1) || (y0 <
8     clip_window.y0)) {
9         return;
10    }
11
12    /* x0 is left of clip window, ignore start part. */
13    if (x0 < clip_window.x0) {
14        width = width + x0;
15        x0 = clip_window.x0;
16    }
17
18    /* Everything outside clip window, nothing to do. */
19    if (width < 0) {
20        return;
21    }
22
23    /* Cut anything going over right edge of clip window. */
24    if (((x0 + width) > clip_window.x1)) {
25        width = width - (x0 + width - clip_window.x1);
26    }
27 }
```

```

25     hagl_hal_hline(x0, y0, width, color);
26 #else
27     hagl_draw_line(x0, y0, x0 + w, y0, color);
28 #endif
29 }
30 */
31 /*
32 * Draw a vertical line with given color. If HAL supports it uses
33 * hardware vline drawing. If not falls back to vanilla line drawing.
34 */
35 void hagl_draw_vline(int16_t x0, int16_t y0, uint16_t h, color_t color)
36 {
37 #ifdef HAGL_HAS_HAL_VLINE
38     int16_t height = h;
39
40     /* x0 or y0 is over the edge, nothing to do. */
41     if ((x0 > clip_window.x1) || (x0 < clip_window.x0) || (y0 >
42         clip_window.y1)) {
43         return;
44     }
45
46     /* y0 is top of clip window, ignore start part. */
47     if (y0 < clip_window.y0) {
48         height = height + y0;
49         y0 = clip_window.y0;
50     }
51
52     /* Everything outside clip window, nothing to do. */
53     if (height < 0) {
54         return;
55     }
56
57     /* Cut anything going over right edge. */
58     if (((y0 + height) > clip_window.y1)) {
59         height = height - (y0 + height - clip_window.y1);
60     }
61
62     hagl_hal_vline(x0, y0, height, color);
63 #else
64     hagl_draw_line(x0, y0, x0, y0 + h, color);
65 #endif
}

```

Funkcja hline sprawdza czy linia jest poza obszarem wyświetlania i używa funkcji hagl_draw_line() zresztą jak hagl_hal_vline() tylko vline sprawdza to wszystko dla linii pionowych.

Później dowodzenie przejmuje algorytm Bresenhama o którym wcześniej pisałem w ramach zapoznania się z stosowanymi algorytmami.

```

1 /*
2  * Draw a line using Bresenham's algorithm with given color.
3  */
4 void hagl_draw_line(int16_t x0, int16_t y0, int16_t x1, int16_t y1,
5   color_t color)
{

```

```

6  /* Clip coordinates to fit clip window. */
7  if (false == clip_line(&x0, &y0, &x1, &y1, clip_window)) {
8      return;
9  }
10
11 int16_t dx;
12 int16_t sx;
13 int16_t dy;
14 int16_t sy;
15 int16_t err;
16 int16_t e2;
17
18 dx = ABS(x1 - x0);
19 sx = x0 < x1 ? 1 : -1;
20 dy = ABS(y1 - y0);
21 sy = y0 < y1 ? 1 : -1;
22 err = (dx > dy ? dx : -dy) / 2;
23
24 while (1) {
25     hagl_put_pixel(x0, y0, color);
26
27     if (x0 == x1 && y0 == y1) {
28         break;
29     };
30
31     e2 = err + err;
32
33     if (e2 > -dx) {
34         err -= dy;
35         x0 += sx;
36     }
37
38     if (e2 < dy) {
39         err += dx;
40         y0 += sy;
41     }
42 }
43 }
```

Na początek jest inicjalizacja zmiennych w kolejności:

- Różnica w osi X (wartość bezwzględna)
- Kierunek kroku w osi X
- Różnica w osi Y (wartość bezwzględna)
- Początkowy błąd

Pętla while() jest główną pętlą odpowiedzialną za rysowanie. Rysuje ona aktualny piksel i stosuje odpowiednie obliczenia czy robić krok w kierunku X czy Y i czy oczywiście dotarła do końca linii.

Teraz funkcje dla koła:

```

1 void hagl_draw_circle(int16_t xc, int16_t yc, int16_t r, color_t color)
{
2     int16_t x = 0;
3     int16_t y = r;
```

```

4 int16_t d = 3 - 2 * r;
5
6 hagl_put_pixel(xc + x, yc + y, color);
7 hagl_put_pixel(xc - x, yc + y, color);
8 hagl_put_pixel(xc + x, yc - y, color);
9 hagl_put_pixel(xc - x, yc - y, color);
10 hagl_put_pixel(xc + y, yc + x, color);
11 hagl_put_pixel(xc - y, yc + x, color);
12 hagl_put_pixel(xc + y, yc - x, color);
13 hagl_put_pixel(xc - y, yc - x, color);
14
15 while (y >= x) {
16     x++;
17
18     if (d > 0) {
19         y--;
20         d = d + 4 * (x - y) + 10;
21     } else {
22         d = d + 4 * x + 6;
23     }
24
25     hagl_put_pixel(xc + x, yc + y, color);
26     hagl_put_pixel(xc - x, yc + y, color);
27     hagl_put_pixel(xc + x, yc - y, color);
28     hagl_put_pixel(xc - x, yc - y, color);
29     hagl_put_pixel(xc + y, yc + x, color);
30     hagl_put_pixel(xc - y, yc + x, color);
31     hagl_put_pixel(xc + y, yc - x, color);
32     hagl_put_pixel(xc - y, yc - x, color);
33 }
34
35
36 void hagl_fill_circle(int16_t x0, int16_t y0, int16_t r, color_t color)
{
37     int16_t x = 0;
38     int16_t y = r;
39     int16_t d = 3 - 2 * r;
40
41     while (y >= x) {
42         hagl_draw_hline(x0 - x, y0 + y, x * 2, color);
43         hagl_draw_hline(x0 - x, y0 - y, x * 2, color);
44         hagl_draw_hline(x0 - y, y0 + x, y * 2, color);
45         hagl_draw_hline(x0 - y, y0 - x, y * 2, color);
46         x++;
47
48         if (d > 0) {
49             y--;
50             d = d + 4 * (x - y) + 10;
51         } else {
52             d = d + 4 * x + 6;
53         }
54     }
55 }
```

Tutaj tak samo jak przy okazji Bresenhama ten algorytm był tłumaczony wcześniej. Tutaj używamy właściwość symetrii okręgu. Ze względu na nią, wystarczy obliczyć punkty dla 1/8 okręgu (od 0 do 45 stopni), a pozostałe punkty można uzyskać poprzez odbicie.

```
1 hagl_put_pixel(xc + x, yc + y, color); // Punkt 1 (0 stopni)
2 hagl_put_pixel(xc - x, yc + y, color); // Punkt 2 (180 stopni)
3 hagl_put_pixel(xc + x, yc - y, color); // Punkt 3
4 hagl_put_pixel(xc - x, yc - y, color); // Punkt 4
5 hagl_put_pixel(xc + y, yc + x, color); // Punkt 5 (90 stopni)
6 hagl_put_pixel(xc - y, yc + x, color); // Punkt 6 (270 stopni)
7 hagl_put_pixel(xc + y, yc - x, color); // Punkt 7
8 hagl_put_pixel(xc - y, yc - x, color); // Punkt 8
```

Funkcja, która wypełnia koło po prostu rysuje dla każdego y 4 linie poziome. Obydwie funkcje posiadają parametr decyzyjny czy następny punkt powinien iść w dół czy pozostać na tym samym punkcie.

I teraz funkcje dla trójkątów:

```
1 void hagl_draw_polygon(int16_t amount, int16_t *vertices, color_t color)
2 {
3     for(int16_t i = 0; i < amount - 1; i++) {
4         hagl_draw_line(
5             vertices[(i << 1) + 0],
6             vertices[(i << 1) + 1],
7             vertices[(i << 1) + 2],
8             vertices[(i << 1) + 3],
9             color
10        );
11    }
12    hagl_draw_line(
13        vertices[0],
14        vertices[1],
15        vertices[(amount <<1) - 2],
16        vertices[(amount <<1) - 1],
17        color
18    );
19 }
20
21 /* Adapted from http://alienryderflex.com/polygon_fill/ */
22 void hagl_fill_polygon(int16_t amount, int16_t *vertices, color_t color)
23 {
24     uint16_t nodes[64];
25     int16_t y;
26
27     float x0;
28     float y0;
29     float x1;
30     float y1;
31
32     int16_t miny = DISPLAY_HEIGHT;
33     int16_t maxy = 0;
34
35     for (uint8_t i = 0; i < amount; i++) {
36         if (miny > vertices[(i << 1) + 1]) {
37             miny = vertices[(i << 1) + 1];
38         }
39         if (maxy < vertices[(i << 1) + 1]) {
40             maxy = vertices[(i << 1) + 1];
41         }
42     }
43 }
```

```

41 }
42
43 /* Loop through the rows of the image. */
44 for (y = miny; y < maxy; y++) {
45
46     /* Build a list of nodes. */
47     int16_t count = 0;
48     int16_t j = amount - 1;
49
50     for (int16_t i = 0; i < amount; i++) {
51         x0 = vertices[(i << 1) + 0];
52         y0 = vertices[(i << 1) + 1];
53         x1 = vertices[(j << 1) + 0];
54         y1 = vertices[(j << 1) + 1];
55
56         if (
57             (y0 < (float)y && y1 >= (float)y) ||
58             (y1 < (float)y && y0 >= (float)y)
59         ) {
60             nodes[count] = (int16_t)(x0 + (y - y0) / (y1 - y0) * (x1
61 - x0));
62             count++;
63         }
64     }
65
66     /* Sort the nodes, via a simple Bubble sort. */
67     int16_t i = 0;
68     while (i < count - 1) {
69         if (nodes[i] > nodes[i + 1]) {
70             int16_t swap = nodes[i];
71             nodes[i] = nodes[i + 1];
72             nodes[i + 1] = swap;
73             if (i) {
74                 i--;
75             }
76         } else {
77             i++;
78         }
79     }
80
81     /* Draw lines between nodes. */
82     for (int16_t i = 0; i < count; i += 2) {
83         int16_t width = nodes[i + 1] - nodes[i];
84         hagl_draw_hline(nodes[i], y, width, color);
85     }
86 }
87
88 void hagl_draw_triangle(int16_t x0, int16_t y0, int16_t x1, int16_t y1,
89 int16_t x2, int16_t y2, color_t color) {
90     int16_t vertices[6] = {x0, y0, x1, y1, x2, y2};
91     hagl_draw_polygon(3, vertices, color);
92 };
93
94 void hagl_fill_triangle(int16_t x0, int16_t y0, int16_t x1, int16_t y1,
95 int16_t x2, int16_t y2, color_t color) {
96     int16_t vertices[6] = {x0, y0, x1, y1, x2, y2};

```

```

96     hagl_fill_polygon(3, vertices, color);
97 }
```

Tak jak przy poprzednich funkcjach mamy kontur oraz wypełnienie.

W konturze tworzy się tablica 6 elementowa zawierająca współrzędne 3 wierzchołków. Wywołuje ona funkcję hagl_draw_polygon() do jego narysowania i tak samo ma funkcja z wypełnieniem trójkąta.

Zacznijmy od draw oraz fill_polygon(). Funkcja przyjmuje liczbę wierzchołków, liczbę współrzędnych oraz oczywiście kolor. Działa ona tak, że rysuje linie między kolejnymi wierzchołkami oraz zamyka wielokąt rysując linię od ostatniego do pierwszego punktu.

Wypełnienie polega na znalezieniu minimalnej oraz maksymalnej wartości y. Dla każdej linii skanowania (od góry do dołu) znajduje punkty przecięcia linii skanowania z krawędziami. Sortuje przecięcia metodą sortowania bąbelkowego oraz rysuje linie między parami punktów.

Teraz przejdźmy sobie już przez ostatnią funkcję - do wyświetlania tekstu.

```

1 uint8_t hagl_put_char(wchar_t code, int16_t x0, int16_t y0, color_t
2   color, const uint8_t *font) {
3   uint8_t set, status;
4   color_t buffer[HAGL_CHAR_BUFFER_SIZE];
5   bitmap_t bitmap;
6   fontx_glyph_t glyph;
7
8   status = fontx_glyph(&glyph, code, font);
9   if (0 != status) {
10     return 0;
11   }
12
13   bitmap.width = glyph.width;
14   bitmap.height = glyph.height;
15   bitmap.depth = DISPLAY_DEPTH;
16
17   bitmap_init(&bitmap, (uint8_t *)buffer);
18   color_t *ptr = (color_t *) bitmap.buffer;
19
20   for (uint8_t y = 0; y < glyph.height; y++) {
21     for (uint8_t x = 0; x < glyph.width; x++) {
22       set = *(glyph.buffer) & (0x80 >> (x % 8));
23       if (set) {
24         *(ptr++) = color;
25       } else {
26         *(ptr++) = 0x0000;
27       }
28     }
29     glyph.buffer += glyph.pitch;
30   }
31
32   // obsługa częściowego wyświetlania znaku
33   int16_t start_x = max(0, -x0);
34   int16_t start_y = max(0, -y0);
35   int16_t end_x = min(bitmap.width, LCD_WIDTH - x0);
36   int16_t end_y = min(bitmap.height, LCD_HEIGHT - y0);
37
38   if (start_x < end_x && start_y < end_y) {
39     for (int16_t y = start_y; y < end_y; y++) {
```

```

39         for (int16_t x = start_x; x < end_x; x++) {
40             color_t pixel = buffer[y * bitmap.width + x];
41             if (pixel != 0x0000) { // tylko nieprzezroczyste piksele
42                 hagl_put_pixel(x0 + x, y0 + y, pixel);
43             }
44         }
45     }
46 }
47
48     return bitmap.width;
49 }
50
51 /*
52 * Write a string of text by calling hagl_put_char() repeadetly. CR and
53 * LF
54 * continue from the next line.
55 */
56
57 uint16_t hagl_put_text(const wchar_t *str, int16_t x0, int16_t y0,
58                       color_t color, const unsigned char *font)
59 {
60     wchar_t temp;
61     uint8_t status;
62     uint16_t original = x0;
63     fontx_meta_t meta;
64
65     status = fontx_meta(&meta, font);
66     if (0 != status) {
67         return 0;
68     }
69
70     do {
71         temp = *str++;
72         if (13 == temp || 10 == temp) {
73             x0 = 0;
74             y0 += meta.height;
75         } else {
76             x0 += hagl_put_char(temp, x0, y0, color, font);
77         }
78     } while (*str != 0);
79
80     return x0 - original;
81 }
```

Funkcje odnosnie gylphu zostały wy tłumaczone w rozdziale "Co to gylph?", więc te funkcję odpuszcze.

Funkcja put_char została trochę zmieniona, ponieważ nie umożliwiała przewijania i znikania napisu z lewej oraz prawej krawędzi wyświetlania a taki efekt chciałem uzyskać. Rysuje ona pojedyńczy znak, gdzie początkowo pobiera informacje o gylphie. Następnie inicjalizuje bitmapy dla znaku oraz konwertujr gylph na bitmapę kolorową. Później jest to co dodałem, czyli rysowanie znaku z przycinaniem do ekranu. Obliczam punkt początkowy rysowania oraz punkt końcowy rysowania. Dla przykładu dla punktu początkowego:

Jeśli $x_0 = -5$ (znak częściowo poza lewą krawędzią)
 $start_x = \max(0, -(-5)) = \max(0, 5) = 5$

Czyli zaczniemy rysować od 5 piksela znaku
 Jeśli $x_0 = 10$ (znak w całości na ekranie)
 $start_x = \max(0, -(10)) = \max(0, -10) = 0$
 Czyli rysujemy od początku znaku

a dla punktu końcowego:

Jeśli znak ma szerokość 10px i $x_0 = 150$, a $LCD_WIDTH = 160$:
 $end_x = \min(10, 160-150) = \min(10, 10) = 10$
 Znak zmieści się w całości
 Jeśli znak ma szerokość 10px i $x_0 = 155$:
 $end_x = \min(10, 160-155) = \min(10, 5) = 5$
 Tylko 5 pierwszych pikseli znaku będzie widoczne

Później warunek oraz pętla odpowiedzialna za rysowanie.

`hagl_put_text` pobiera metadane fontu oraz zawiera pętle do rysowania kolejnych znaków. Korzysta z poprzednio wymienionej funkcji `hagl_put_char()`.

I to na tyle, jeżeli chodzi o bibliotekę HAGL. Z reszty nie korzystam, dlatego nie ma co tego tłumaczyć.

Przejdźmy teraz do ostatnich kodów. Czyli połączenie biblioteki HAGL oraz ramki.

```

1 static void executeONK(Frame *frame)
2 {
3     uint8_t x = 0, y = 0, r = 0, filling = 0;
4     color_t color = 0;
5     if (!parseParameters(frame->data, "uuuuC", &x, &y, &r, &filling, &
6         color))
7     {
8         sendStatus(ERR_NOT_RECOGNIZED);
9         return;
10    }
11    switch(filling)
12    {
13        case 0:
14            hagl_draw_circle(x, y, r, color);
15            break;
16        case 1:
17            hagl_fill_circle(x, y, r, color);
18            break;
19    }
20}
21 static void executeONP(Frame *frame)
22 {
23     uint8_t x = 0, y = 0, width = 0, height = 0, filling = 0;
24     color_t color = 0;
25     if (!parseParameters(frame->data, "uuuuuC", &x, &y, &width, &height, &
26         filling, &color)) {
27         sendStatus(ERR_NOT_RECOGNIZED);
28         return;
29    }
30    switch(filling)
31    {
32        case 0:

```

```

33     hagl_draw_rectangle(x, y, width, height, color);
34     break;
35 case 1:
36     hagl_fill_rectangle(x, y, width, height, color);
37     break;
38 }
39 }
40
41
42 static void executeONT(Frame *frame)
43 {
44     uint8_t x1 = 0, y1 = 0, x2 = 0, y2 = 0, x3 = 0, y3 = 0, filling = 0;
45     color_t color = 0;
46     if (!parseParameters(frame->data, "uuuuuuuC", &x1, &y1, &x2, &y2, &
47     x3, &y3, &filling, &color))
48     {
49         sendStatus(ERR_NOT_RECOGNIZED);
50         return;
51     }
52     if (!isValidTriangle(x1, y1, x2, y2, x3, y3)) {
53         sendStatus(ERR_INVALID_TRIANGLE);
54         return;
55     }
56     switch(filling)
57     {
58         case 0:
59             hagl_draw_triangle(x1, y1, x2, y2, x3, y3, color);
60             break;
61         case 1:
62             hagl_fill_triangle(x1, y1, x2, y2, x3, y3, color);
63             break;
64     }
65 }
66
67 static void executeONN(Frame *frame)
68 {
69     if (!parseParameters(frame->data, "uuuuCt", &text.x, &text.y, &text.
70     fontSize,
71                     &text.scrollSpeed, &text.color, text.displayText
72 )) {
73         sendStatus(ERR_NOT_RECOGNIZED);
74         return;
75     }
76
77     text.startX = text.x;
78     text.startY = text.y;
79     text.textLength = 0;
80     while(text.displayText[text.textLength] != L'\0') {
81         text.textLength++;
82     }
83     text.firstIteration = true;
84
85     text.isScrolling = (text.scrollSpeed > 0);
86     text.lastUpdate = HAL_GetTick();
87
88     const uint8_t* font;
89     switch(text.fontSize) {
90         case 1: font = font5x7; break;

```

```

88     case 2: font = font5x8; break;
89     case 3: font = font6x9; break;
90     default: font = font5x7;
91 }
92
93 if(!text.scrollSpeed) {
94     hagl_put_text((wchar_t*)text.displayText, text.x, text.y, text.
95 color, font);
96 }
97
98 static void executeOFF(Frame *frame)
99 {
100
101 switch(frame->data[0])
102 {
103 case 0:
104     HAL_GPIO_WritePin(BL_GPIO_Port, BL_Pin, GPIO_PIN_RESET);
105     break;
106 case 1:
107     lcdClear();
108     break;
109 default:
110     sendStatus(ERR_WRONG_OFF_DATA);
111 }
112 }
```

To wszystkie funkcje podstawowe wykonywane na podstawie komendy. Tutaj nie ma dużego poziomu skomplikowania, ponieważ tu tylko są inicjalizacje zmiennych oraz wywołania funkcji parseParameters(), która akurat już jest skomplikowana i trochę nad nią siedziała, aby działała.

Zanim jeszcze do niej przejdziemy, to trzeba wytlumaczyć co to za funkcja sendStatus() oraz isValidTriangle().

```

1 static bool isValidTriangle(int16_t x1, int16_t y1, int16_t x2, int16_t
2 y2, int16_t x3, int16_t y3) {
3     int32_t a2 = (int32_t)(x2 - x1) * (x2 - x1) + (int32_t)(y2 - y1) * (
4         y2 - y1);
5     int32_t b2 = (int32_t)(x3 - x2) * (x3 - x2) + (int32_t)(y3 - y2) * (
6         y3 - y2);
7     int32_t c2 = (int32_t)(x1 - x3) * (x1 - x3) + (int32_t)(y1 - y3) * (
8         y1 - y3);
9
10    if (a2 + b2 <= c2 || b2 + c2 <= a2 || c2 + a2 <= b2) {
11        return false;
12    }
13
14    int32_t cross = (int32_t)(x2 - x1) * (y3 - y1) - (int32_t)(y2 - y1)
15 * (x3 - x1);
16    if (cross == 0) {
17        return false;
18    }
19
20    return true;
21 }
22 static void sendStatus(StatusCode_t status) {
23     if(status < STATUS_COUNT) {
```

```

19     prepareFrame(STM32_ADDR, PC_ADDR, "BCK", "%s", STATUS_MESSAGES [
20     status]);
21 }
22
23 typedef enum {
24     ERR_GOOD = 0,
25     ERR_FAIL,
26     ERR_WRONG_SENDER,
27     ERR_WRONG_CRC,
28
29     ERR_DISPLAY_AREA,
30     ERR_WRONG_OFF_DATA,
31     ERR_WRONG_DATA,
32     ERR_INVALID_TRIANGLE,
33     ERR_TOO MUCH_TEXT,
34     ERR_NOT_RECOGNIZED,
35
36     STATUS_COUNT
37 } StatusCode_t;
38
39
40 static const char* const STATUS_MESSAGES [] = {
41     "GOOD",
42     "FAIL",
43     "WRONG_SENDER",
44     "WRONG_CRC",
45
46     "DISPLAY_AREA",
47     "WRONG_OFF_DATA",
48     "WRONG_DATA",
49     "INVALID_TRIANGLE",
50     "TOO MUCH_TEXT",
51     "NOT_RECOGNIZED"
52 };

```

Jedna z funkcji to sprawdzenie, czy z podanych parametrów można utworzyć trójkąt. Akurat ten sposób znalazłem gdzieś na forach i jest to sprawdzenie jedno z prostszych. Oblicza długości boków i:

- a2 to kwadrat długości boku między punktami (x1, y1) i (x2,y2)
- b2 to kwadrat długości boku między punktami (x2, y2) i (x3, y3)
- c2 to kwadrat długości boku między punktami (x3, y3) i (x1,y1)

Używany jest tutaj typ int32_t aby uniknąć przepełnienia przy mnożeniu oraz używany jest kwadrat, aby uniknąć pierwiastkowania. Później już tylko sprawdzana nierówność trójkąta, ponieważ: **suma kwadratów dwóch boków musi być większa od kwadratu trzeciego boku.** Następnie jest sprawdzenie, czy punkty nie leżą na tej samej linii.

Kod odnośnie sendStatus() jest prosty. Stworzony został enum, gdzie są zdefiniowane błędy oraz tablica komunikatów. Sama funkcja przyjmuje kod statusu jako argument i sprawdza czy kod jest prawidłowy (mniejszy od STATUS_COUNT). I po tym wywołuje zwykłą funkcję prepareFrame(). Z początku miałem powstawiane prepareFrame() w każdą liniatkę, lecz one się nie różniły niczym oprócz komunikatem zwrotnym, dlatego pomyślałem i poszukałem jakie-

goś rozwiązania. Sam STATUS_COUNT jest umiejscowiony na samym końcu, ponieważ w typie wyliczeniowym enum jest ustawione liczenie od zera, dlatego jeżeli będziemy mieć STATUS_COUNT jako ostatnie i jego wartość powiedzmy, że 10, to wtedy wiemy, że tych komunikatów faktycznie jest 10. Kod na prepareFrame() wygląda następująco:

```

1 void prepareFrame(uint8_t sender, uint8_t receiver, const char *command,
2   const char *format, ...) {
3   Frame frame = {0};
4   frame.sender = sender;
5   frame.receiver = receiver;
6   strncpy((char *)frame.command, command, COMMAND_LENGTH);
7
8   uint8_t formattedData[MAX_DATA_SIZE];
9   va_list args;
10  va_start(args, format);
11  vsnprintf((char *)formattedData, MAX_DATA_SIZE, format, args);
12  va_end(args);
13
14  size_t dataLen = strlen((char *)formattedData);
15
16  uint8_t crcInput[MAX_FRAME_WITHOUT_STUFFING];
17  size_t crcInputLen = 0;
18
19  crcInput[crcInputLen++] = frame.sender;
20  crcInput[crcInputLen++] = frame.receiver;
21  memcpy(crcInput + crcInputLen, frame.command, COMMAND_LENGTH);
22  crcInputLen += COMMAND_LENGTH;
23  memcpy(crcInput + crcInputLen, formattedData, dataLen);
24  crcInputLen += dataLen;
25
26  char crcOutput[2];
27  calculateCrc16(crcInput, crcInputLen, crcOutput);
28
29  uint8_t rawPayload[MAX_FRAME_WITHOUT_STUFFING];
30  size_t rawPayloadLen = 0;
31
32  rawPayload[rawPayloadLen++] = frame.sender;
33  rawPayload[rawPayloadLen++] = frame.receiver;
34  memcpy(rawPayload + rawPayloadLen, frame.command, COMMAND_LENGTH);
35  rawPayloadLen += COMMAND_LENGTH;
36  memcpy(rawPayload + rawPayloadLen, formattedData, dataLen);
37  rawPayloadLen += dataLen;
38
39  rawPayload[rawPayloadLen++] = crcOutput[0];
40  rawPayload[rawPayloadLen++] = crcOutput[1];
41
42  uint8_t stuffedPayload[MAX_FRAME_LEN];
43  size_t stuffedLen = byteStuffing(rawPayload, rawPayloadLen,
44  stuffedPayload);
45
46  uint8_t finalFrame[MAX_FRAME_LEN + 2];
47  size_t finalLen = 0;
48
49  finalFrame[finalLen++] = FRAME_START;
50  memcpy(finalFrame + finalLen, stuffedPayload, stuffedLen);
51  finalLen += stuffedLen;
52  finalFrame[finalLen++] = FRAME_END;
53  USART_sendFrame(finalFrame, finalLen);

```

Pozwolę sobie tego bardzo nie rozpisywać. Jest to po prostu funkcja robiąca to, co użytkownik robi przed wysłaniem ramki przez terminal. Mamy wstawianie nadawcy i odbiorcy, wstawianie komendy BCK oraz danych jako komunikat zwrotny. Na koniec liczymy z tego CRC, pakujemy i wysyłamy z powrotem.

```

1 static bool parseParameters(const uint8_t *data, const char *format,
2     ...) {
3     if (!data || !format) {
4         return false;
5     }
6     va_list args;
7     va_start(args, format);
8
9     const uint8_t *data_ptr = data;
10    const char *fmt_ptr = format;
11
12    int u_param_count = 0;
13    uint8_t scrollSpeed = 0;
14
15    while (*fmt_ptr) {
16        switch (*fmt_ptr) {
17            case 'u': {
18                uint8_t *value_ptr = va_arg(args, uint8_t*);
19                *value_ptr = *data_ptr++;
20                u_param_count++;
21                if (u_param_count == 4) {
22                    scrollSpeed = *value_ptr;
23                }
24                if (*data_ptr == ',') {
25                    data_ptr++;
26                }
27                break;
28            }
29            case 'C': {
30                color_t *color_ptr = va_arg(args, color_t*);
31
32                if ((*data_ptr >= 'A' && *data_ptr <= 'Z')
33                    || (*data_ptr >= 'a' && *data_ptr <= 'z')) {
34                    return false;
35                }
36
37                uint8_t lsb = *data_ptr++;
38                if (*data_ptr == ',')
39                    data_ptr++;
40
41                if ((*data_ptr >= 'A' && *data_ptr <= 'Z')
42                    || (*data_ptr >= 'a' && *data_ptr <= 'z')) {
43                    return false;
44                }
45
46                uint8_t msb = *data_ptr++;
47                if (*data_ptr == ',')
48                    data_ptr++;
49                *color_ptr = ((uint16_t) msb << 8) | lsb;
50                break;
51            }
52        }
53    }
54}
```

```

51 case 't': {
52     size_t realCharCount = 0;
53     size_t maxChars = (scrollSpeed == 0) ? 25 : 50;
54     wchar_t *text_ptr = va_arg(args, wchar_t*);
55
56     while (*data_ptr && realCharCount < maxChars) {
57         if ((*data_ptr & 0x80) == 0) {
58             text_ptr[realCharCount++] = (wchar_t) *data_ptr++;
59         } else if ((*data_ptr & 0xE0) == 0xC0) {
60             if (!data_ptr[1])
61                 break;
62             wchar_t wc = ((data_ptr[0] & 0x1F) << 6)
63                 | (data_ptr[1] & 0x3F);
64             text_ptr[realCharCount++] = wc;
65             data_ptr += 2;
66         } else {
67             data_ptr++;
68         }
69     }
70
71     if (realCharCount > maxChars) {
72         va_end(args);
73         sendStatus(ERR_TOO MUCH_TEXT);
74         return false;
75     }
76
77     text_ptr[realCharCount] = L'\0';
78     break;
79 }
80 default:
81     va_end(args);
82     return false;
83 }
84 fmt_ptr++;
85 }
86
87 va_end(args);
88 return true;
89 }
```

To jest funkcja odpowiedzialna za przydzielanie wartości przekazanych w ramce do zmiennych zainicjalizowanych w programie. Wy tłumaczmy sobie, ponieważ to będzie ważne.

Funkcja przyjmuje wskaźnik do danych wejściowych, stringa (łańcuch znaków) określających format danych oraz argumenty zmienne (...). Na dobry początek następuje sprawdzenie parametrów wejściowych. Następnie jest inicjalizacja zmiennych:

```

1 va_list args;
2 va_start(args, format);
3 const uint8_t* data_ptr = data;
4 const char* fmt_ptr = format;
5 uint8_t token[51];
6 size_t token_idx = 0;
7
8 uint8_t scrollSpeed = 0;
9 bool hasScrollSpeed = false;
```

Po tym napotykamy na główną pętlę parsowania, która działa dopóki są znaki w formacie. Mamy tutaj obsługę kilku typów danych:

- u - unsigned byte

```

1      case 'u': {
2          uint8_t* value_ptr = va_arg(args, uint8_t*);
3          *value_ptr = *data_ptr++;
4          u_param_count++;
5          if (u_param_count == 4) {
6              scrollSpeed = *value_ptr;
7          }
8          if (*data_ptr == ',') {
9              data_ptr++;
10         }
11         break;
12     }
13

```

va_arg pobiera następny argument jako wskaźnik na uint8_t. Następnie jest zapisywana wartość z danych do zmiennej docelowej. Jeśli t pierwsza wartość 'u', jest traktowana jako scrollSpeed. Pomijany jest przecinek, jeśli występuje po wartości. Jeszcze mamy warunek, który to z kolei parametr jednobajtowy, ponieważ szybkość przewijania jest na 4 pozycji.

- C - kolor

```

1      case 'C': {
2          color_t* color_ptr = va_arg(args, color_t*);
3
4          if ((*data_ptr >= 'A' && *data_ptr <= 'Z') ||
5              (*data_ptr >= 'a' && *data_ptr <= 'z')) {
6              return false;
7          }
8
9          uint8_t lsb = *data_ptr++;
10         if (*data_ptr == ',') data_ptr++;
11
12         if ((*data_ptr >= 'A' && *data_ptr <= 'Z') ||
13             (*data_ptr >= 'a' && *data_ptr <= 'z')) {
14             return false;
15         }
16
17         uint8_t msb = *data_ptr++;
18         if (*data_ptr == ',') data_ptr++;
19         *color_ptr = ((uint16_t)msb << 8) | lsb;
20         break;
21     }
22

```

Pobiera dwa bajty, składają je w 16-bitową wartość koloru. Obsługuje on format little-endian zgodnie ze specyfikacją wyświetlacza. Pomija przecinki pomiędzy bajtami i zapisuje złożony kolor do zmiennej docelowej. Posiada jeszcze zabezpieczenie przed wpisaniem koloru jako WHITE, ponieważ jak się wpisało i wysłało ramkę to niestety przechodziła i wyświetlało np, ITE,WITAM SERDECZNIE.

- t - tekst

```

1   case 't': {
2       size_t realCharCount = 0;
3       size_t maxChars = (scrollSpeed == 0) ? 25 : 50;
4       wchar_t *text_ptr = va_arg(args, wchar_t*);
5
6       while (*data_ptr && realCharCount < maxChars) {
7           if ((*data_ptr & 0x80) == 0) {
8               text_ptr[realCharCount++] = (wchar_t) *data_ptr++;
9           } else if ((*data_ptr & 0xE0) == 0xC0) {
10              if (!data_ptr[1])
11                  break;
12              wchar_t wc = ((data_ptr[0] & 0x1F) << 6)
13                  | (data_ptr[1] & 0x3F);
14              text_ptr[realCharCount++] = wc;
15              data_ptr += 2;
16          } else {
17              data_ptr++;
18          }
19      }
20  }

```

Kopiuje znaki do bufora token aż do końca danych lub limitu 50 znaków. Sprawdza maksymalną długość tekstu - 25 znaków dla tekstu statycznego i 50 dla tekstu przewijanego. Jeśli tekst jest za długi to zwraca błąd. Kopiuje zebrany tekst do zmiennej docelowej. W tym przypadku mamy sprawdzenie czy wysłany znak to znak ASCII czy znak UTF-8. Polega to na tym, że w przypadku ASCII sprawdza czy najstarszy bit jest jako wartość 0. Jeśli tak, znak jest kopowany oraz wskaźnik jest przesuwany. Jeżeli nie to sprawdzane jest czy to początek 2-bajtowej sekwencji UTF-8. Na początek bierzemy 5-bitów z pierwszego bajtu, przesuwamy o 6 pozycji w lewo oraz bierzemy 6 bitów z drugiego bajtu. Później łączymy oba komponenty w jeden znak. Jeżeli natrafimy na inny przypadek niż wymienione to pomijamy. Ktoś może zapytać, no dobrze, ale dlaczego tak to sprawdzamy? Otóż dlatego bo:

- ASCII używa 7 bitów. Najstarszy bit w ASCII jest zawsze 0. Wykonywana operacja $\& 0x80$ sprawdza najstarszy bit (1000 0000). Jeżeli wartość AND jest 0 to wiemy, że mamy znak ASCII.
- UTF-8 używa 2 bajtów. 0xE0 to inaczej 1110 0000 a 0xC0 to inaczej 1100 0000 w binarnym. Znaki mają sekwencję taką, że pierwszy bajt ma 110xxxxx a drugi 10xxxxxx. Zobaczmy na przykład dla literki 'a':

11000100 (0xC4) $\&$ 00011111 = 00000100 (pierwsze 5 bitów)
 10000101 (0x85) $\&$ 00111111 = 00000101 (kolejne 6 bitów)

00000100 \ll 6 = 00010000000 (przesunięcie)
 | 00000101 = 00010000101 (połączenie)

No dobra, ale czym jest to wynikowe 00010000101? Przecież nam nic to nie daje. No nie do końca. Jest to wartość 261 decimalnie co daje nam punkt kodowy Unicode dla znaku 'a'. System używa tej samej wartości do identyfikacji znaku w tabeli Unicode.

No okej, to nie można wysyłać danych jako Unicode od razu? Niby można, ale może to być źle zinterpretowane przez różne systemy, może powodować problemy z synchronizacją oraz nie ma oznaczenia gdzie zaczyna i kończy znak. UTF-8 jest bardziej popularne w komunikacji i obsługiwany przez różne systemy, dlatego zdecydowałem się na takie rozwiązanie.

Jeszcze mamy tutaj licznik "prawdziwych" znaków. Jak już się dowiedzieliśmy polskie znaki zajmują 2-bajty, dlatego jak byśmy mieli zwykły licznik to by to liczył jako dwa znaki a nie jeden, dlatego ta modyfikacja była nieunikniona.

Jeżeli nie ma żadnego typu, to obsługujemy błąd i kończymy pracę. Warto także wspomnieć, co to jest va_args. Jest to mechanizm w C, który pozwala nam na tworzenie funkcji z zmienną liczbą argumentów. Jest to specjalny typ danych, który służy do przechowywania informacji o zmiennych argumentach. Można o nim trochę myśleć jak o wskaźniku, który będzie się przemieszczał po argumentach.

I już ostatnia funkcja przedstawiona w tym dokumencie będzie funkcją odpowiadającą za animację tekstu.

```
1 void updateScrollingText(void) {
2     if (!text.isScrolling || text.scrollSpeed == 0) {
3         return;
4     }
5
6     uint32_t currentTime = HAL_GetTick();
7     if ((currentTime - text.lastUpdate) >= (text.scrollSpeed >> 1)) {
8         text.lastUpdate = currentTime;
9
10    uint8_t charWidth;
11    uint8_t charHeight;
12    const uint8_t *font;
13    switch (text.fontSize) {
14        case 1:
15            charWidth = 5;
16            charHeight = 7;
17            font = font5x7;
18            break;
19        case 2:
20            charWidth = 5;
21            charHeight = 8;
22            font = font5x8;
23            break;
24        case 3:
25            charWidth = 6;
26            charHeight = 9;
27            font = font6x9;
28            break;
29        default:
30            charWidth = 5;
31            charHeight = 7;
32            font = font5x7;
33            break;
34    }
35
36    int16_t textWidth = text.textLength * charWidth;
37
38    if (!text.firstIteration) {
```

```

39     text.x += text.textLength;
40
41
42     if (text.x > LCD_WIDTH) {
43         text.x = -textWidth;
44         text.y += charHeight;
45
46
47     if (text.y >= LCD_HEIGHT - charHeight) {
48         text.y = text.startY;
49     }
50 }
51 } else {
52
53     text.x = -textWidth;
54     text.y = text.startY;
55     text.firstIteration = false;
56 }
57
58 lcdClear();
59 hagl_put_text(text.displayText, text.x, text.y, text.color, font);
60 lcdCopy();
61 }
62 }
```

I to ta funkcja jest wywoływana w SysTick_Handler() wraz z inkrementacją zmiennej tick++, służącej do liczenia taktów, które są używane w funkcji delay, która się przewinęła przy okazji lcd_init().

```

1 void delay(uint32_t delayMs){
2     uint32_t startTime = tick;
3     while(tick < (startTime+delayMs));
4 }
```

Funkcja do animacji tekstu z budowy przypomina funkcję do wyświetlania statycznego tekstu. Na początek sprawdzamy czy tekst ma być przewijany lub prędkość przewijania jest 0. Przewijanie musimy jakoś kontrolować i korzystamy tutaj z HAL_GetTick(), który pobiera aktualny czas w milisekundach a następnie tworzymy if'a, który określa opóźnienie między aktualizacjami (tekstu). Czym większy scroll tym mniejsze opóźnienie i szybsze przewijanie. Później następuje wybór czcionki i obliczenie całkowitej szerekości tekstu na potrzeby przewijania. Logika przewijania tekstu jest tutaj:

```

1 if (!text.firstIteration) {
2     text.x += text.textLength;
3
4
5     if (text.x > LCD_WIDTH) {
6         text.x = -textWidth;
7         text.y += charHeight;
8
9
10    if (text.y >= LCD_HEIGHT - charHeight) {
11        text.y = text.startY;
12    }
13 }
14 } else {
```

```
15
16     text.x = -textWidth;
17     text.y = text.startY;
18     text.firstIteration = false;
19 }
```

Jest to tak zrobione, ponieważ gdy użytkownik wyśle swoje koordynaty, to jak przejdziemy przez cały ekran, to dobrze by było zacząć od koordynatów (0,0) a nie np. (110, 100).

A cała struktura do tekstu wygląda tak:

```
1 typedef struct {
2     wchar_t displayText[50];
3     int16_t x;
4     int16_t y;
5     int16_t startX;
6     int16_t startY;
7     uint8_t fontSize;
8     uint8_t scrollSpeed;
9     uint16_t color;
10    uint8_t textLength;
11    bool isScrolling;
12    bool firstIteration;
13    uint32_t lastUpdate;
14 } ScrollingTextState;
```

Niektórzy by mogli zdać pytanie, czym jest wchar_t? I dlaczego jest go tyle w tym kodzie? Jest to typ danych używany do przechowywania szerokich znaków. Jest to specjalny typ, który może przechowywać większe wartości niż standardowy char, co jest szczególnie przydatne przy obsłudze znaków Unicode i innych zestawów znaków wielobajtowych. Jest on też wymagany przez bibliotekę hagl. Obsługuje on również polskie znaki.

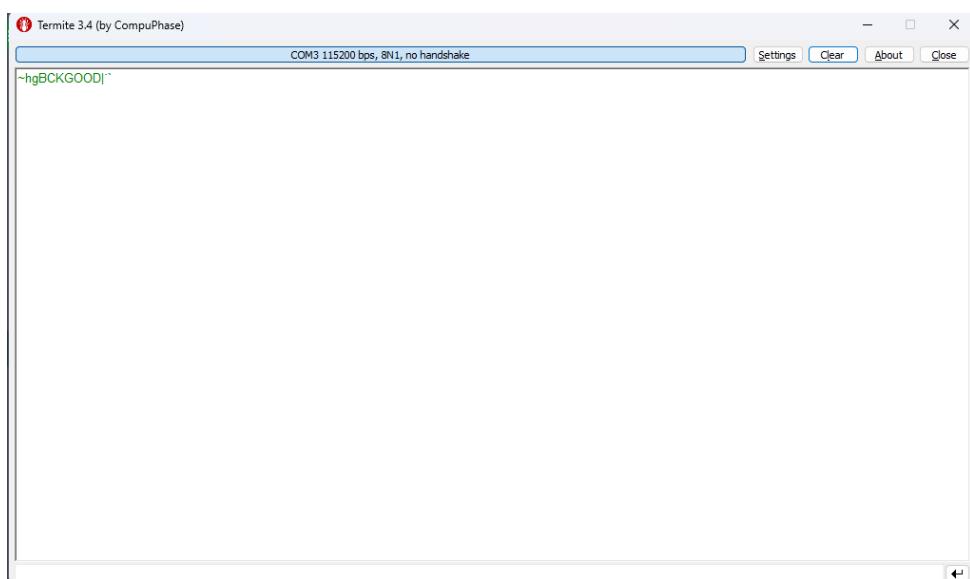
I tym oto sposobem skończyliśmy omawiać cały projekt. Jak widać była to droga bardzo dłużna, wymagająca pomyślnego zastanowienia i wielu poprawek, ale było warto. W końcu jest to obszerny projekt, który pewnie nadal posiada swoje wady, ale droga do realizacji oraz zaliczenia projektu jest czymś, z czego można być zadowolonym i wyciągnąć wiele wniosków do przyszłej pracy nad projektami.

8 Przykładowe ramki oraz odpowiedzi

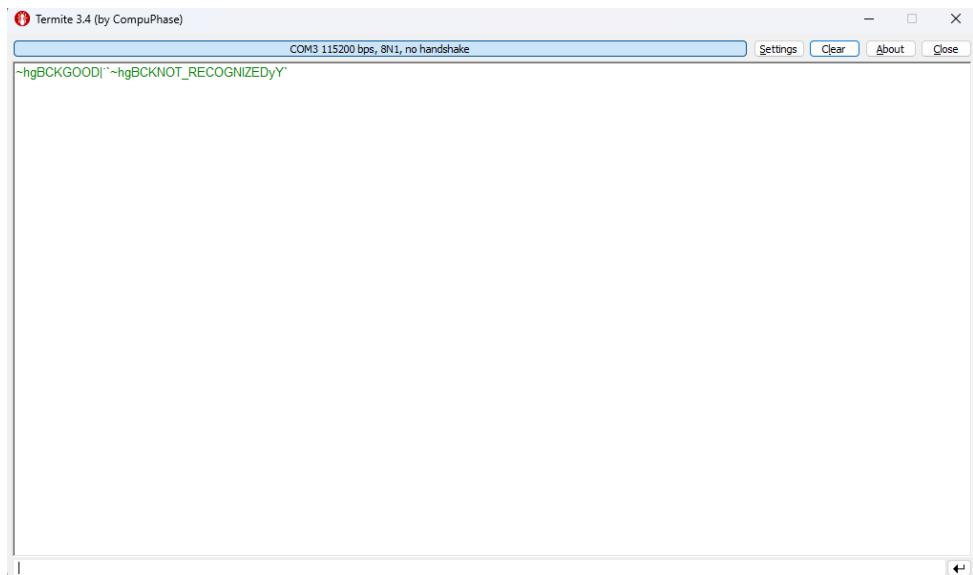
8.1 Przykładowe ramki

- `~ghONN0x00,0x00,0x03,0x96,0xF8,0x00,0xC50xBB0xC30x930xC50x81TY0x240x23``
- ramka z polskimi znakami "Żółty"
- `~ghONN0x00,0x00,0x03,0x96,0x00,0xF8,WITAM SERDECZNIE0x6C0x69`` - wyświetlenie zielonego tekstu "WITAM SERDECZNIE", który się przewija.
- `~ghOFF0x000x300xAD`` - wyłączenie podświetlenia
- `~ghOFF0x010x200x8C`` - reset ekranu

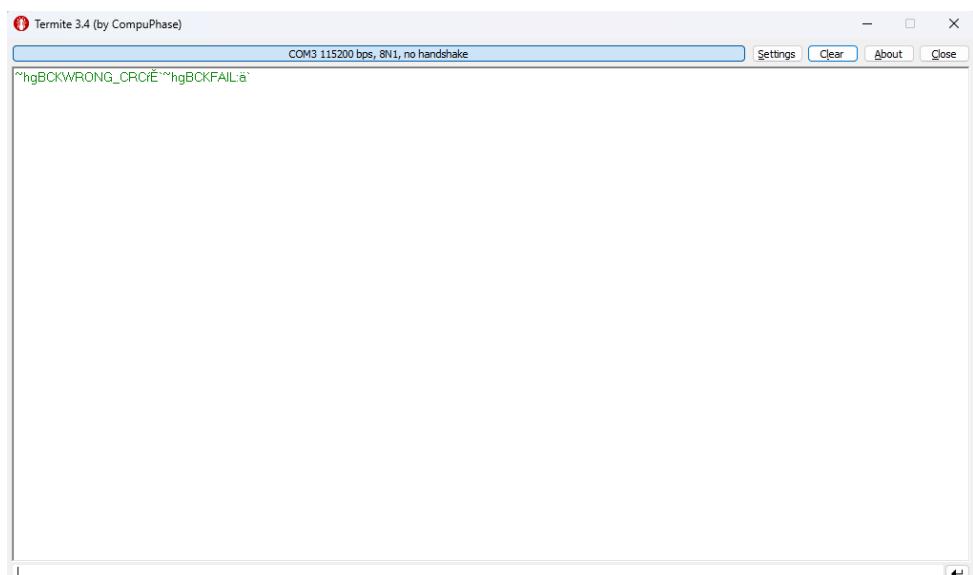
8.2 Przykładowe odpowiedzi



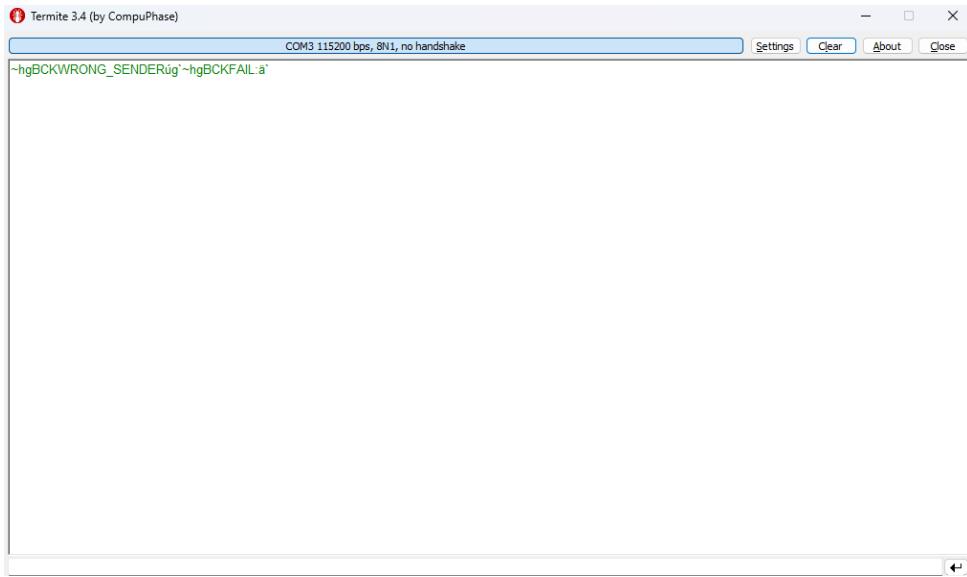
Rysunek 18: Poprawna ramka



Rysunek 19: Wartości nie rozpoznane - zapisane w kodzie ASCII



Rysunek 20: Niepoprawnie obliczone CRC przez użytkownika



Rysunek 21: Niepoprawny nadawca ramki

8.3 Zużycie zasobów STM32L476RG - ciekawostka

Projekt pochłania dosyć dużo zasobów, ponieważ tak jak wcześniej pisałem odnośnie buforowania. Z podwójnym buforem miałbym zapelnienie pamięci RAM w wysokości 90 %. Próbowałem zaciągnąć do pracy RAM2, który jest do dyspozycji, ale wsadzenie tam pomniejszych tablic inicjalizacyjnych dużo nie zmieniało, a nawet powodowało dziwne błędy.

MIKRO_PROJECT_FINAL.elf - /MIKRO_PROJECT_FINAL/Debug - Jan 24, 2025, 9:33:33 PM						
Memory Regions		Memory Details				
Region	Start address	End address	Size	Free	Used	Usage (%)
RAM	0x20000000	0x20017fff	96 KB	47,22 KB	48,78 KB	50.81%
RAM2	0x10000000	0x10007fff	32 KB	32 KB	0 B	0.00%
FLASH	0x08000000	0x080fffff	1024 KB	951,43 KB	72,57 KB	7.09%

Rysunek 22: Zużycie zasobów płytki STM32

9 Źródła

- Datasheet ST7735S
- User Manual LCD Module
- Strona producenta
- Komunikacja SPI
- Programy DEMO od producenta
- Biblioteka HAGL na licejnji MIT (link do nowszej, lecz ja wykorzystałem starszą wersję, dostosowaną pod moje wymogi)
- RGB565