Opis szablonu do ćwiczenia 4

1 Motywacja

Motywacją do napisania niniejszego dokumentu jest niemały poziom skomplikowania projektu na ćwiczenie wykorzystujące mikrokontroler STM32F746G. W tym dokumencie znajdą się informacje na temat dostępnych timerów, wykorzystanych przerwań oraz struktury projektu. Zostanie także krótko opisana kolejność wykonywania operacji, oraz możliwe problemy, które mogą wystąpić zarówno teraz jak i w przyszłych projektach.

2 Płytka rozwojowa

Wykorzystany mikrokontroler STM32F746G jest częścią płytki rozwojowej, na której zamontowany został z jednej strony wyświetlacz z ekranem dotykowym, a z drugiej takie elementy jak złącze do Arduino Uno,

port Ethernet, złącze do kamery, złącze karty µSD i wiele innych. Z naszego punktu widzenia najbardziej interesujące są: wyświetlacz, przycisk B1 (oznaczany również jako „User") oraz dioda LD1. W przyszłości wykorzystane zostanie dodatkowo złącze Arduino Uno, aby zamontować tam dodatkowe przetworniki cyfrowo analogowe.

2.1 Wyświetlacz

Wyświetlacz podłaczony do płytki rozwojowej jest wyświetlaczem kolorowym, który obsługiwany będzie przy użyciu biblioteki BSP (*Board Support Package*), pozwalającej na wygodną realizację podstawowych operacji. Do realizacji ćwiczenia nie jest wymagana znajomość modułu odpowiedzialnego za obsługę wyświetlacza, ani umiejętności jego konfiguracji, gdyż liczba parametrów jakie są dostępne w tym celu przekracza znacznie zakres tych ćwiczeń. Dodatkowo szablon dostępny jako punkt startowy ćwiczenia również nie został napisany „z palca", a wygenerowany z użyciem STM32CubeMX – jest to znacznie szybsza metoda, choć nie koniecznie najlepsza.

Podstawowymi funkcjami, które będą wykorzystane dalszych ćwiczeniach są:

- rysowanie pikseli,
- rysowanie kształtów,
- rysowanie tekstu,
- zmiana koloru tła,
- zmiana koloru pierwszoplanowego,
- pobranie koloru tła,
- pobranie koloru pierwszoplanowego.

Nazwy funkcji realizujących powyższe zadania są intuicyjne. Wszystkie funkcje związane z wyświetlaczem rozpoczynają się od `BSP_LCD_`. W tym momencie warto zawuażyć, że dopełnianie składni (skrót **CTRL+SPACJA**) znacznie upraszcza przeglądanie dostępnych funkcji bibliotecznych. Funkcja rysująca piksele ma następującą deklarację (fragment z pliku `stm32746g_discovery_lcd.c`):

```
/**
* @brief  Draws a pixel on LCD.
* @param  Xpos: X position
* @param  Ypos: Y position
* @param  RGB_Code: Pixel color in ARGB mode (8-8-8-8)
* @retval None
*/
void BSP_LCD_DrawPixel(uint16_t Xpos, uint16_t Ypos, uint32_t RGB_Code)
```

Jako pierwszy argument należy wstawić numer wiersza, jako drugi numer kolumny piksela, który ma zmienić kolor na podany w trzecim argumencie. Numeracja wierszy i kolumn rozpoczyna się od 0, gdzie punkt (0,0) znajduje się w lewym górnym rogu wyświetlacza. Wszelkie punkty między wierszami 0 i 271 włącznie oraz między kolumnami 0 i 459 włącznie są widoczne na wyświetlaczu, Istnieje jednak zestaw

punktów, które nie mają odzwierciedlenia w rzeczywistości i są wykorzystywane do innych celów – będą one ignorowane w trakcie ćwiczeń. Ostatnim argumentem funkcji jest kolor – wiele został już zdefiniowanych w pliku `stm32746g_discovery_lcd.h`:

```
#define LCD_COLOR_BLUE          ((uint32_t)0xFF0000FF)
#define LCD_COLOR_GREEN         ((uint32_t)0xFF00FF00)
#define LCD_COLOR_RED           ((uint32_t)0xFFFF0000)
#define LCD_COLOR_CYAN          ((uint32_t)0xFF00FFFF)
#define LCD_COLOR_MAGENTA       ((uint32_t)0xFFFF00FF)
#define LCD_COLOR_YELLOW        ((uint32_t)0xFFFFFF00)
#define LCD_COLOR_LIGHTBLUE     ((uint32_t)0xFF8080FF)
#define LCD_COLOR_LIGHTGREEN    ((uint32_t)0xFF80FF80)
#define LCD_COLOR_LIGHTRED      ((uint32_t)0xFFFF8080)
#define LCD_COLOR_LIGHTCYAN     ((uint32_t)0xFF80FFFF)
#define LCD_COLOR_LIGHTMAGENTA  ((uint32_t)0xFFFF80FF)
#define LCD_COLOR_LIGHTYELLOW   ((uint32_t)0xFFFFFF80)
#define LCD_COLOR_DARKBLUE      ((uint32_t)0xFF000080)
#define LCD_COLOR_DARKGREEN     ((uint32_t)0xFF008000)
#define LCD_COLOR_DARKRED       ((uint32_t)0xFF800000)
#define LCD_COLOR_DARKCYAN      ((uint32_t)0xFF008080)
#define LCD_COLOR_DARKMAGENTA   ((uint32_t)0xFF800080)
#define LCD_COLOR_DARKYELLOW    ((uint32_t)0xFF808000)
#define LCD_COLOR_WHITE         ((uint32_t)0xFFFFFFFF)
#define LCD_COLOR_LIGHTGRAY     ((uint32_t)0xFFD3D3D3)
#define LCD_COLOR_GRAY          ((uint32_t)0xFF808080)
#define LCD_COLOR_DARKGRAY      ((uint32_t)0xFF404040)
#define LCD_COLOR_BLACK         ((uint32_t)0xFF000000)
#define LCD_COLOR_BROWN         ((uint32_t)0xFFA52A2A)
#define LCD_COLOR_ORANGE        ((uint32_t)0xFFFFA500)
#define LCD_COLOR_TRANSPARENT   ((uint32_t)0xFF000000)
```

Jak widać kolory zostały zapisane jako zmienne `uint32_t` – tak należy również je przechowywać w swoim kodzie. Zapis ten wynika z faktu, iż kolory w ramach tego i następnych ćwiczeń reprezentowane będą dla ułatwienia jako liczby, gdzie najbardziej znaczące 8 bitów określa przeźroczystość (`0x00` – całkowicie przeźroczysty, `0xFF` – całkowicie widoczny), następne 8 bitów określa natężenie koloru czerwonego (`0x00` – zerowe, `0xFF` – maksymalne), kolejne 8 bitów określa natężenie koloru zielonego (wartości analogicznie jak w przypadku koloru czerwonego), a najmłodszych 8 bitów określa natężenie koloru niebieskiego (analogicznie jak dla czerwonego i zielonego). Przeznaczenie poszczególnych bajtów (8 bitów) jest wyraźnie widoczne w przypadku kolorów `LCD_COLOR_RED`, `LCD_COLOR_GREEN` oraz `LCD_COLOR_BLUE`. Mimo zastosowania zapisu wykorzystującego przeźroczystość, kolor „przeźroczysty" jest równoważny kolorowi czarnemu. Wynika to z faktu iż korzystamy wyłącznie z jednej warstwy – zakładamy więc, że pod warstwą roboczą znajduje się czarna odchłań. A więc narysowanie białego prostokąta o przeźroczystości na poziomie 50% spowoduje narysowaniem szarego prostokąta. W szczególności nie spowoduje to połowicznego przykrycia obecnie wytworzonego obrazu białym prostokątem! Wniosek z powyższego jest raczej oczywisty – nie ma potrzeby i sensu operować przeźroczystością w ramach tego i następnych ćwiczeń.

Skoro wiadomo już jak rysować pojedyncze piksele, warto rozważyć bardziej ambitne kształty. Konwencja nazewnicza jest następująca: początek `BSP_LCD_Draw` oznacza wyrysowanie wyłącznie krawędzi kształtu, natomiast `BSP_LCD_Fill` spowoduje wypełnienie również wnętrza. W obu przypadkach użyty

zostanie kolor pierwszoplanowy. Dostępne kształty to: `Circle` (koło/okrąg), `Ellipse` (elipsa), `Polygon` (wielokąt) oraz `Rect` (prostokąt). Argumenty dotyczące tych funkcji są oczywiste – w razie jednak problemów warto zajrzeć do komentarza nad definicją danej funkcji. Poza powyższymi kształtami, które mogą mieć wypełnienie, jest możliwość wyrysowania kształtów takie jak: `Line` (linia), `VLine` (linia pionowa), `HLine` (linia pozioma), `Pixel` (piksel/punkt), `Bitmap` (bitmapa). Jak wcześniej – argumenty powinny być zrozumiałe, w razie jednak problemów dokładny opis znajduje się tuż nad definicją danej funkcji. Warto zauważyć, że niektóre funkcje nie rysują dokładnie tego, czego możnaby się było spodziewać. Np. funkcja `BSP_LCD_DrawRect` rysuje prostokąt, lecz w jego prawym dolnym rogu brakuje narożnego piksela.

Zdefiniowany został dodatkowo zestaw funkcji do wyświetlania znaków. Funkcje te mają wspólny początek `BSP_LCD_Display`, natomiast końcówki mogą być następujące: `Char` (znak), `StringAt` (ciąg znaków w wybranym miejscu), `StringAtLine` (ciąg znaków w wybranej linii). Wyświetlanie znaku wymaga podania pozycji w pionie i poziomie oraz znaku do wyświetlenia. Warto zauważyć, że podana pozycja jest lewym górnym rogiem rysowanego znaku (wraz z jego tłem). Funkcja `StringAt` jako ostatni argumenty przyjmuje "tryb pracy". Zdefiniowane są one następująco:

- `LEFT_MODE` – lewy górny róg napisu umieszczany jest w podanym punkcie początkowym, punkt (0,0) znajduje się w lewym górnym rogu ekranu,
- `RIGHT_MODE` – prawy górny róg napisu umieszczany jest w podanym punkcie początkowym, punkt (0,0) znajduje się w prawym górnym rogu ekranu,
- `CENTER_MODE` – środek górnej krawędzi napisu umieszczany jest w podanym punkcie początkowym, punkt (0,0) znajduje się na środku górnej krawędzi ekranu.

Funkcja `StringAtLine` zawsze wypisuje tekst począwszy od lewej krawędzi ekranu (między krawędzią a tekstem jest 1 piksel odstępu). W zależności od wybranej czcionki ekran dzielony jest na linie, w których wypisywany jest tekst. Numer linii podawany jest jako pierwszy argument – linie numerowane są od zera.

Do przydatnych funkcji należą także te, które nie generują obrazu – modyfikują one jednak produkty następujących po nich operacji. Są to:

- `BSP_LCD_SetTextColor` – ustawia kolor pierwszoplanowego (zarówno dla tekstu jak i figur),
- `BSP_LCD_SetBackColor` – ustawia kolor tła dla tekstu,
- `BSP_LCD_GetTextColor` – zwraca kolor pierwszoplanowy,
- `BSP_LCD_GetBackColor` – zwraca kolor tła,
- `BSP_LCD_ClearStringLine` – czyści podaną linię tekstu kolorem tła,
- `BSP_LCD_Clear` – czyści cały ekran podanym kolorem,
- `BSP_LCD_GetFont` – zwraca wskaźnik na strukturę definiującą czcionkę (zawiera między innymi wysokość i szerokość znaku),
- `BSP_LCD_GetXSize` – zwraca wysokość ekranu,

- `BSP_LCD_GetYSize` – zwraca szerokość ekranu,
- `BSP_LCD_SetFont` – ustawia czcionkę (jako argument warto wykorzystać zdefiniowane w bibliotece struktury o nazwach `Font8`, `Font12`, `Font16`, `Font20` oraz `Font24`, gdzie liczba stojąca na końcu nazwy oznacza wysokość znaku w danej czcionce).

Pozostałe funkcje są raczej nieprzydatne z punktu widzenia ćwiczenia – najczęściej wykorzystywane są w ramach inicjalizacji i konfiguracji wyświetlacza.

2.2 Ekran dotykowy

Ekran dotykowy wykorzystany w tym i następnych ćwiczeniach zostanie jako główne narzędzie do interakcji z użytkownikiem. Do realizacji tego zadania wykorzystana zostanie funkcja `BSP_TS_GetState`. Jako argument przyjmuje ona wskaźnik na strukturę typu `TS_StateTypeDef`. Po wywołaniu funkcji `BSP_TS_GetState`, w podanej strukturze znajdą się informacje dotyczące obecnego stanu ekranu dotykowego. Najważniejsze i prawdopodobnie jedyne przydatne w ramach tych ćwiczeń pola tej struktury to:

- `touchDetected` – liczba wykrytych dotknięć,
- `touchX` – tablica współrzędnych `x` poszczególnych wykrytych dotknięć (indeksowana od 0 do `N`, gdzie `N` to liczba wykrytych dotknięć),
- `touchY` – tablica współrzędnych `y` poszczególnych wykrytych dotknięć (indeksowana od 0 do `N`, gdzie `N` to liczba wykrytych dotknięć).

Istnieje możliwość wyzwolenia przerwania w momencie wykrycia zmiany stanu ekranu dotykowego, lecz dla ułatwienia obsługi zostanie ona zastąpiona regularnym, częstym odpytywaniem (proponowane jest odpytywanie co 100ms).

2.3 Przycisk B1

Na płytce uruchomieniowej został zamontowany przycisk. Znajduje się on na spodniej stronie płytki – po przeciwnej stronie niż lewy górny róg ekranu (patrząc od spodu płytki jest to prawy górny róg). Odczyt stanu przycisku realizowany jest przy użyciu funkcji `BSP_PB_GetState(BUTTON_TAMPER)`. Rezultatem wywołania tej funkcji jest wartość określająca czy przycisk został wciśnięty (`GPIO_PIN_SET`) czy nie (`GPIO_PIN_RESET`). Dla ułatwienia korzystania z tego przycisku został zrealizowany na płycie rozwojowej filtr dolnoprzepustowy, a więc jeden z klasycznych mechanizmów niwelacji drgań styków. Szczegóły znajdują się na rysunku 21 w dokumencie UM1907.

2.4 Dioda LD1

W przeciwieństwie do płyty ZL27ARM, tutaj zamontowano wyłącznie jedną diodę do użytku programisty. Jest to zielona dioda LD1 – znajduje się ona pod czarnym przyciskiem Reset, który znajduje się bezpośrednio pod przyciskiem B1. Sterować nią można za pomocą funkcji `BSP_LED_Off(LED1)` – wyłączenie diody, `BSP_LED_On(LED1)` – włączenie diody oraz `BSP_LED_Toggle(LED1)` – przełączenie diody. Może być ona zarówno wygodnym mechanizmem do prostego debugowania programu, lub kontrolką informującą o „zajętości" systemu.

3 Kod szablonu

3.1 Timery

Do dyspozycji użytkownik tego mikrokontrolera ma wiele timerów: zaawansowane TIM1 i TIM8, ogólnego użytku TIM2-TIM5 oraz TIM9-TIM14, a także podstawowe TIM6 i TIM7. Aby nie zagłębiać się w szczegóły konfiguracji timerów – część z nich została przykładowo skonfigurowana. Zrealizowane to zostało dla timerów TIM2-TIM5. Ich konfiguracja najduje się w pliku `main.c` w funkcjach o nazwach `MX_TIMx_Init`, gdzie za `x` należy wstawić numer timera. Metoda konfiguracji różni się nieznacznie od tej stosowanej w ramach standardowej biblioteki peryferiali, lecz nazwy pól stuktury inicjalizującej w większości się pokrywają. W ramach ćwiczeń będzie konieczne skonfigurowanie okresu wyzwolenia przerwania związanego z przepełnieniem timera. Tak jak w SPL wymagało to odpowiedniego uzupełnienia wartości `Period` oraz `Prescaler`, tak samo i tutaj należy odpowiednio zmodyfikować te pola. Ich znaczenie jest identyczne jak w przypadku SPL. Należy zwrócić uwagę jednak na pewien szczegół – dla timerów TIM2 i TIM5 pole `Period` jest 32-bitowe, natomiast dla timerów TIM3 i TIM4 jest ono 16-bitowe.

Funkcje do obsługi przerwań również już są zdefiniowane (znajdują się w pliku `stm32f7xx_it.c`) i zgodnie z konwencją z SPL mają nazwy od `TIM2_IRQHandler` po `TIM5_IRQHandler`. Każda z nich zawiera wywołanie `HAL_TIM_IRQHandler(&htimx);`, gdzie za `x` należy podstawić numer timera. Służy ono do odpowiedniej obsługi przerwania – w szczególności wyczyszczenia odpowiednich bitów przerwania oczekującego i wywołania funkcji `Callback`, związanej z odpowiednim zdarzeniem. Dzięki takiemu schematowi programista może nie implementować już całego przerwania, a jedynie funkcję związaną z interesującym go zdarzeniem. Dla przykładu, obsługę przepełnienia rejestru timera TIM2 można zrealizować jako:

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
    if(htim->Instance == TIM2){
        // obsługa przerwania
    }
}
```

gdzie funkcja obsługi przerwania timera TIM2 wygląda następująco:

```
void TIM2_IRQHandler(void){
    HAL_TIM_IRQHandler(&htim2);
}
```

Alternatywnie można oczywiście zastosować podejście klasyczne, tj. sprawdzenie flagi i własnoręczne wyczyszczenie stosownego bitu oczekującego przerwania, lecz mocno zalecane jest zaimplementowanie funkcji `HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *tim)`, zgodnie z powyższym wzorem. Funkcję tę można zaimplementować również w pliku `main.c` co pozwoli na ograniczenie zamieszania związanego ze zmiennymi o modyfikatorze `extern`.

Priorytety przerwań ustawione są w funkcji `main` w pliku `main.c`:

```
    /* Timers initialisation */
    MX_TIM2_Init();
    MX_TIM3_Init();
    MX_TIM4_Init();
    MX_TIM5_Init();
```

```
    /* Interrupts configuration */
    HAL_NVIC_SetPriority(SysTick_IRQn, 0, 0); // priority of SysTick
    HAL_NVIC_SetPriority(TIM2_IRQn   , 1, 0); // priority of TIM2
    HAL_NVIC_SetPriority(TIM3_IRQn   , 0, 0); // priority of TIM3
    HAL_NVIC_SetPriority(TIM4_IRQn   , 0, 0); // priority of TIM4
    HAL_NVIC_SetPriority(TIM5_IRQn   , 0, 0); // priority of TIM5

    /* Interrupts enabling */
    HAL_NVIC_EnableIRQ(TIM2_IRQn); // enabling handling functions of TIM2
    HAL_NVIC_EnableIRQ(TIM3_IRQn); // enabling handling functions of TIM3
    HAL_NVIC_EnableIRQ(TIM4_IRQn); // enabling handling functions of TIM4
    HAL_NVIC_EnableIRQ(TIM5_IRQn); // enabling handling functions of TIM5
```

Kolejno zrealizowana jest inicjalizacja (konfiguracja) samych timerów, następnie ustawienie ich priorytetów (oraz SysTick'a) a na koniec włączenie samych funkcji do obsługi przerwań. O ile ta ostatnia różni się od jej wersji z SPL jedynie dodanym początkiem `HAL_`, o tyle funkcja do ustawienia priorytetów nie wykorzystuje już specjalnej struktury do konfiguracji. Argumentami funkcji `HAL_NVIC_SetPriority` są kolejno: nazwa przerwania, priorytet wywłaszczenia oraz podpriorytet. Ważne jest, aby pamiętać, że priorytety domyślnie ustawione są tak, że **wszystkie 4 bity opisują priorytet wywłaszczenia**, nie pozostawiając nic na podpriorytet.

3.2 Przetwornik analogowo cyfrowy

W ramach tego i następnych ćwiczeń wykorzystany został przetwornik analogowo cyfrowy powiązany z DMA (*Direct Memory Access*). Przekłada się to na kod programu w taki sposób, że student nie musi już obsługiwać przerwania związanego z końcem przetwarzania. Wynik przetwarzania jest nieustannie aktualizowany i zawiera się w zmiennej `adc_value`. Jest to wynik uśredniony ze 100 ostatnich pomiarów. Implementacja uśredniania została zrealizowana w funkcji (w pliku `main.c`)

```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* AdcHandle){
    static int i=0;
    static uint32_t tmpval= 0;
    for(i=0,tmpval=0;i<ADC_BUFFER_LENGTH; ++i)
        tmpval += uhADCxConvertedValue[i];
    adc_value = tmpval/ADC_BUFFER_LENGTH;
}
```

gdzie `ADC_BUFFER_LENGTH` jest zdefiniowany w pliku `main.c` i wynosi 100, natomiast tablica `uhADCxConvertedValue` zawiera zrealizowane pomiary – właśnie tutaj informacje z ADC przekazuje DMA. Funkcja `HAL_ADC_ConvCpltCallback` wywoływana jest w momencie, gdy DMA wypełni wszystkie 100 elementów wspomnianej tablicy. W tym momencie dokonywane jest powyższe uśrednianie i wynik zapisywany jest do `adc_value`.