KATEDRA AUTOMATYKI, BIOMECHANIKI
I MECHATRONIKI

Laboratorium
Automatyki

TEMAT:

Sterowanie sygnalizacja świetlną na skrzyżowaniu – automat skończony (Finite State Machine)

Opracował: dr inż. Krystian Polczyński

wersja: 1.0

Cel ćwiczenia

Celem ćwiczenia jest opracowanie programu napisanego w języku C sterującego sygnalizacja świetlna na skrzyżowaniu według założonego scenariusza. Program sterujący opracowany zostanie przy wykorzystaniu teorii automatu skończonego.

1 Automat skończony

Automat skończony (inne nazwy: automat stanów, maszyna stanów) jest to obliczeniowy model matematyczny służący do reprezentowania systemów, które przechodzą między skończoną liczbą stanów w zależności od wejściowego ciągu symboli.

Matematycznie automat skończony *A* można zdefiniować w następujący sposób

$$A = (Q, \Sigma, \delta, q_0, F), \qquad (1)$$

gdzie:

Q – skończony zbiór stanów,

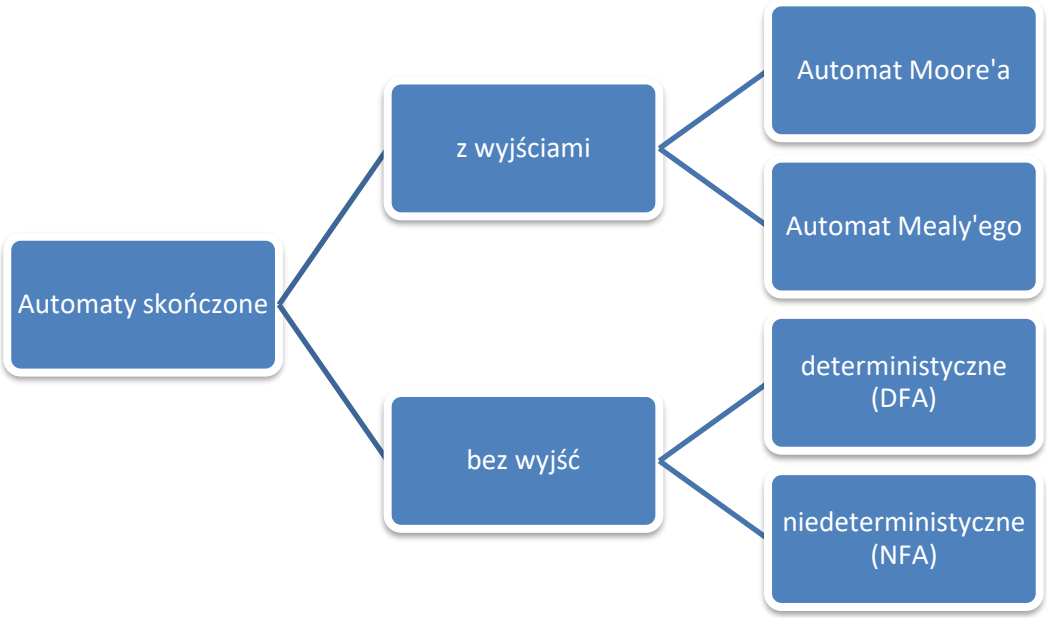
Σ – alfabet symboli wejściowych,

q_0 – stan startowy, $q_0 \in Q$,

F – zbiór stanów końcowych, $F \subseteq Q$,

δ – funkcja przejścia.

Automaty skończone możemy podzielić według diagramu na rys. 1.



Rys. 1. Podział automatów skończonych.

Automaty z wyjściami (przetwarzające) generują sygnał wyjściowy w odpowiedzi na wejście i aktualny stan. Używane są w systemach, które nie tylko rozpoznają ciągi symboli, ale także produkują wyniki w czasie działania. Automaty te są stosowane w układach sterowania oraz cyfrowych maszynach sekwencyjnych, gdzie ważne jest nie tylko rozpoznawanie ciągów wejściowych, ale także generowanie odpowiednich sygnałów wyjściowych.

Automaty bez wyjść (rozpoznające/akceptory) służą do rozpoznawania ciągów symboli wejściowych. Nie generują wyjścia w postaci ciągów sygnałów, lecz jedynie określają, czy dany ciąg wejściowy jest

akceptowany. Automat kończy pracę w jednym z możliwych stanów, z których pewne są akceptujące (oznaczają akceptację ciągu), a inne nie.

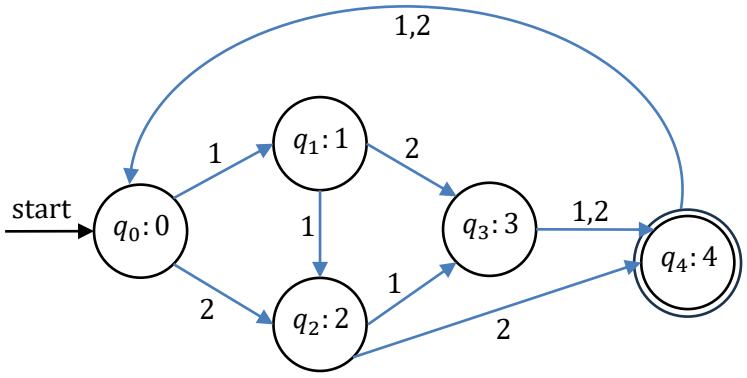
- Automat Moore'a
  Automat skończony, w którym sygnał wyjściowy zależy wyłącznie od aktualnego stanu automatu. Wyjście zmienia się tylko w momencie przejścia do innego stanu.
- Automat Mealy'ego
  Automat skończony, w którym sygnał wyjściowy zależy zarówno od aktualnego stanu, jak i symbolu wejściowego. Wyjście może zmieniać się przy każdej zmianie wejścia.

- Układ deterministyczny
  Automat skończony, w którym dla każdego stanu i symbolu wejściowego istnieje dokładnie jedno możliwe przejście do następnego stanu.
- Układ niedeterministyczny
  Automat skończony, w którym dla każdego stanu i symbolu wejściowego może istnieć wiele możliwych przejść do następnych stanów, w tym także brak przejścia. Automat niedeterministyczny jest tylko umownie ponieważ można go zawsze przekształcić w deterministyczny.

2 Przykład automatu skończonego – prosty automat do kupowania batonów

Rozważmy prostą w działaniu maszynę do wydawania biletów parkingowych, która akceptuje monety 1 zł, 2 zł i nie wydaje reszty. Załóżmy, że pobyt na parkingu kosztuje 4 zł, a to znaczy, że automat ma rozpoznawać (rozumieć) te ciągi, których suma równa jest 4 złotym i wtedy wyda bilet. Stany wejściowe automatu zależą od kombinacji wrzuconych monet z tego wynika, że $q_0 = 0$ (*stan początkowy*), $q_1 = 1, q_2 = 2, q_3 = 3, q_4 = 4$ (*stan końcowy*).

Opracujmy graf działania automatu pokazujący zmianę stanów układu, patrz rys. 2. Rozpoczynam od narysowania kółka ze stanem początkowym q_0, stan ten opatrzymy strzałką '*start*'. Następnie na grafie umieścimy wszystkie możliwe stany q_1, q_2, q_3, q_4 wraz ze stanem końcowym q_4, który oznaczymy kółkiem z podwójną obwiednią.



Rys. 2. Graf działania prostego automatu parkingowego.

Ponieważ maszyna przyjmuje tylko 1 zł i 2 zł, w każdym stanie q_i tylko te wartości mogą pojawić się na wejściu i tylko dla tych przypadków musimy określić do jakiego kolejnego stanu przejdzie automat. Ciągi wrzuconych monet (słowa), które maszyna rozpozna jako te wydające bilet (sukces) są następujące: (1,2,1), (1,2,2), (1,1,1,1), (1,1,1,2), (1,1,2), (2,1,1), (2,1,2), (2,2).

Chcąc opisać nasz automat matematycznie zgodnie ze wzorem (1) należy przyjąć, że:

- Skończony zbiór stanów $Q = \{q_0, q_1, q_2, q_3, q_4\}$
- Alfabet symboli wejściowych $\Sigma = \{e_1, e_2\} = \{1,2\}$

- Stan startowy q_0
- Zbiór stanów końcowych $F = q_4$
- Funkcja przejścia δ określana jest przez poniższą tabelę 1 (stan startowy poprzedzony jest symbolem →, natomiast stan końcowy poprzedzony jest symbolem *)

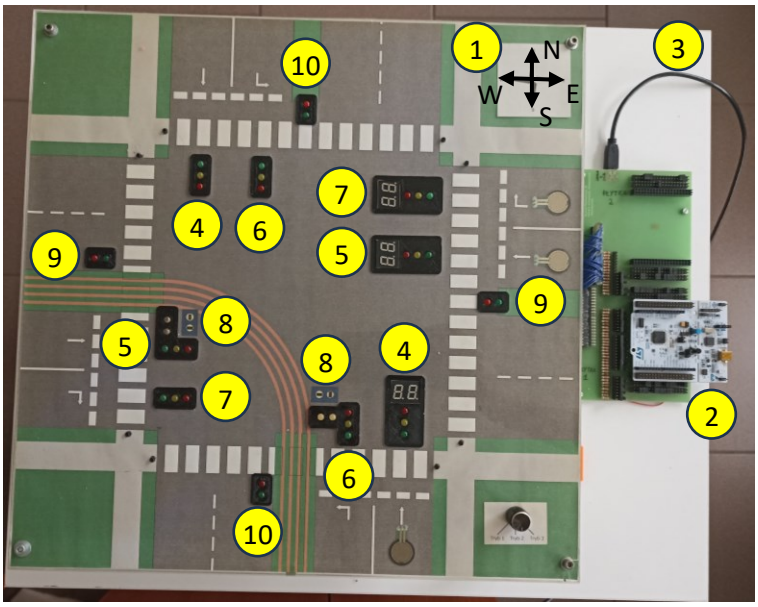
Tabela 1. Stabelaryzowana funkcja przejść δ automatu skończonego.

	Symbole Σ	
Stan obecny układu q^t	1	2
$\rightarrow q_0$	q_1	q_2
q_1	q_2	q_3
q_2	q_3	q_4
q_3	q_4	q_4
$* q_4$	q_1	q_1
	Następny stan układu q^{t+1}	

Funkcja przejść określa następny stan q^{t+1} jaki układ ma osiągnąć po podaniu na jego wejście symbolu ze zbioru $\Sigma = \{e_1, e_2\}$, jeśli symbol ten został podany w stanie q^t. Zapisać to można np. w następujący sposób $q^{t+1} = \delta(e_1, e_2, q^t)$.

3 Budowa stanowiska laboratoryjnego

Stanowisko laboratoryjne powstało w ramach pracy inżynierskiej Pana Marcina Laufera w 2020 roku pod promotorstwem dr. hab. inż. Pawła Olejnika [1]. Zdjęcie stanowiska pokazano na rys. 3. Numerem (1) oznaczono rysunek z kierunkami świata. Numerem (2) oznaczono mikrokontroler STM32 Nucleo L152RE, który steruje sygnalizacją świetlna makiety. Mikrokontroler na makiecie podpisany jest jako *PŁYTKA 1.* Numerem (3) oznaczono przewód USB zasilający mikrokontroler oraz sygnalizatory.



Rys. 3. Stanowisko laboratoryjne. Znaczenie numerów znajduje się w tekście.

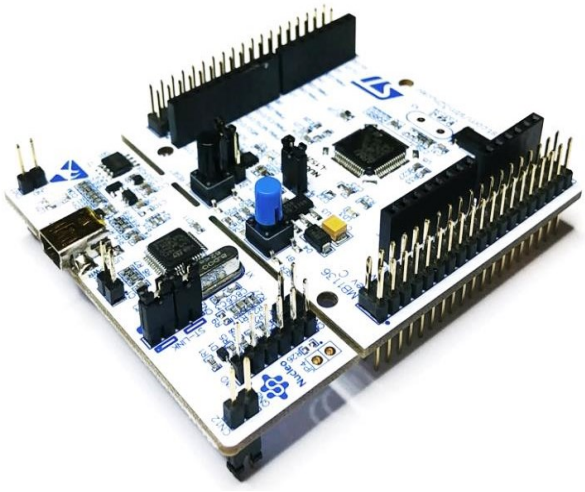
Sygnalizatory dla samochodów oraz pieszych wykonane są z diod elektroluminescencyjnych (LED) o różnych kolorach – zielonym, pomarańczowym, czerwonym. Sygnalizatory dla tramwajów wykonane są z diod LED o kolorze białym. Samochody mogą jechać w dwóch kierunkach świata N-S, W-E. Białe strzałki na makiecie pokazują możliwości jazdy samochodów. Tramwaj może tylko skręcać w lewo. Warto zwrócić uwagę, że gdy na wszystkich sygnalizatorach do skrętu pojawi się światło zielone, skręcające samochody i tramwaj nie będą sobie przeszkadzać. Numerami (4) oznaczono sygnalizatory kierujące ruchem samochodów na wprost zgodnie z kierunkiem S-N świata, a numerami (5) zgodnie z kierunkiem W-E świata. Numerem (6) oznaczono sygnalizatory kierujące pojazdami skręcającymi w lewo, a numerem (7) samochodami skręcającymi w prawo. Numerem (8) oznaczono sygnalizatory zezwalające na skręt lewo/prawo tramwaju. Numerami (9) oznaczono sygnalizatory zezwalające na ruch pieszych w kierunku S-N, a numerem (10) w kierunku W-E.

Poszczególne diody sygnalizatorów sterowane są przy pomocy wyjść mikrokontrolera. W celu ograniczenia liczby potrzebnych złącz, diody o tych samych barwach sygnalizatorów dla kierunków, po których pojazdy mogą poruszać się jednocześnie zostały połączone w pary. W ten sposób 32 diody mogą być sterowane poprzez 18 wyjść cyfrowych mikrokontrolera. Schemat połączeń można znaleźć w Załączniku 1.

Płytka ewaluacyjna STM32 Nucleo

STM32 Nucleo to rodzina płytek ewaluacyjnych opartych na rdzeniu ARM Cortex produkowanych przez firmę STMicroelectronics. Wszystkie płytki z serii Nucleo-64 są wyposażone w złącza zgodne ze standardem Arduino Uno oraz złącze ST Morpho – wspólne dla wielu płytek tego producenta. Dzięki takiemu rozwiązaniu możliwe jest stosowanie nie tylko płytek rozszerzających (z ang. *shield*), stworzonych dla płytek Nucleo lecz także wielu rozszerzeń dla znacznie bardziej popularnej płytki Arduino Uno. Platforma jest również wyposażona w interfejsy komunikacji USART, I2C, oraz SPI. Dodatkowo, z pomocą wirtualnego portu COM oraz programu ST Link Utility możliwe jest monitorowanie stanów portów płytki w czasie rzeczywistym. Oprócz podstawowej serii Nucleo-64 producent oferuje również mniejsze płytki Nucleo-32 (zgodne ze standardem Arduino Nano) oraz większe i wydajniejsze płytki Nucleo-144.

W każdej z serii producent umożliwia wybór płytki spośród trzech kategorii:



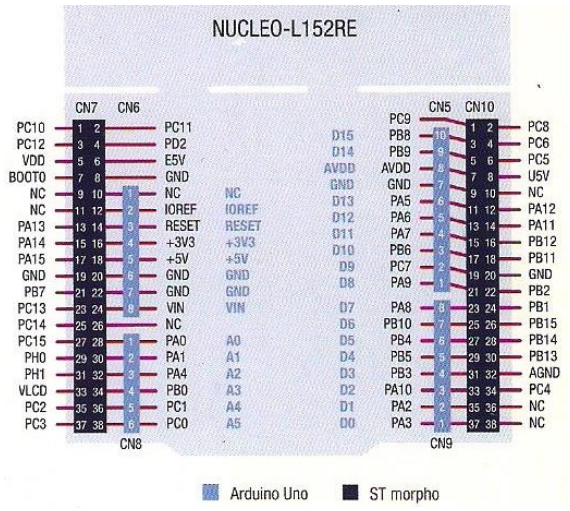
- standardowa (*mainstream*);
- bardzo niskie zużycie energii (*ultra-low-power*);
- wysoka wydajność (*high-performance*).

Do obsługi makiety skrzyżowania ulicznego wybrano model Nucleo L152RE (

rys. 4), należący do kategorii płytek z bardzo niskim zużyciem energii.

Rys. 4. Płytka ewaluacyjna STM32 Nucleo L152RE

Zaletą płytek z rodziny Nucleo jest liczba wejść-wyjść ogólnego przeznaczenia (GPIO – z ang. *general-purpose input/output*) oraz przyjazne środowisko programistyczne STM32CubeIDE obsługujące jezyk C. Na rys. 5 pokazano schemat wyprowadzeń płytki.



Rys. 5. Schemat wyprowadzeń płytki STM32 Nucleo L152RE

Podczas ćwiczenia opracują Państwo program na mikrokontroler STM32 sterujący sygnalizacja świetlną skrzyżowania. Poszczególne fazy sterowania, w których będzie znajdował się nasz automat skończony przedstawione są w tabeli 2.

Tabela 2. Konfiguracja pinów mikrokontrolera oraz tablica prawdy dla poszczególnych faz cyklu działania sygnalizacji świetlnej.

DIODY	PIN	FAZA CYKLU											
		0	1	2	3	4	5	6	7	8	9	10	11
SN_Front_R	A0	1	1	0	0	1	1	1	1	1	1	1	1
SN_Front_Y	A1	0	1	0	1	0	0	0	0	0	0	0	0
SN_Front_G	B2	0	0	1	0	0	0	0	0	0	0	0	0
SN_Left_R	B3	1	1	1	1	1	1	1	1	1	1	0	0
SN_Left_Y	A4	0	0	0	0	0	0	0	0	0	1	0	1
SN_Left_G	B5	0	0	0	0	0	0	0	0	0	0	1	0
WE_Front_R	A6	1	1	1	1	1	1	0	0	1	1	1	1
WE_Front_Y	A7	0	0	0	0	0	1	0	1	0	0	0	0
WE_Front_G	A8	0	0	0	0	0	0	1	0	0	0	0	0
WE_Right_R	A9	1	1	1	1	1	1	1	1	1	1	0	0
WE_Right_Y	A10	0	0	0	0	0	0	0	0	0	1	0	1
WE_Right_G	A11	0	0	0	0	0	0	0	0	0	0	1	0
T_LeftRight_Drive	A12	0	0	0	0	0	0	0	0	0	1	1	1
T_LeftRight_Stop	A13	1	1	1	1	1	1	1	1	1	0	0	0
P_NS_R	A14	1	0	0	0	1	1	1	1	1	1	1	1
P_NS_G	A15	0	1	1	1	0	0	0	0	0	0	0	0
P_WE_R	B0	1	1	1	1	1	0	0	0	1	1	1	1
P_WE_G	B1	0	0	0	0	0	1	1	1	0	0	0	0

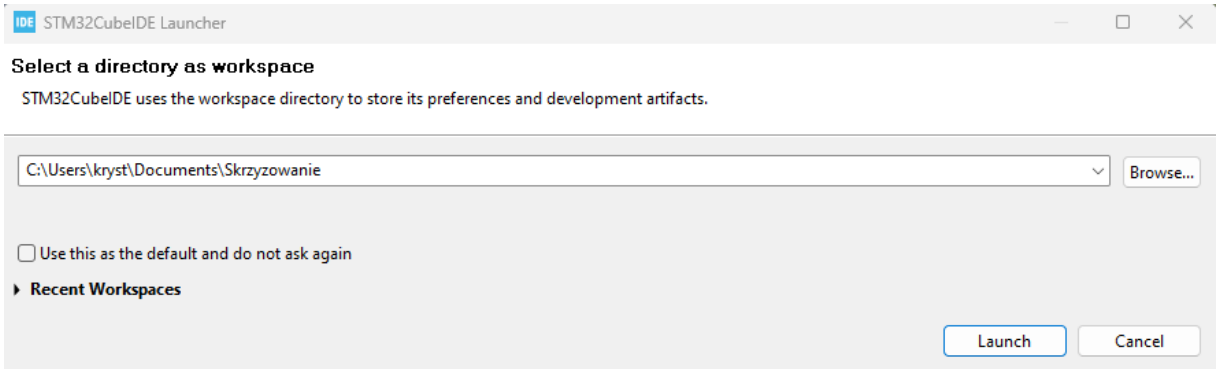
Legenda:

- tylko światła czerwone
- tylko światła w kierunku SN
- tylko światła w kierunku WE
- tylko światła przy skrętach

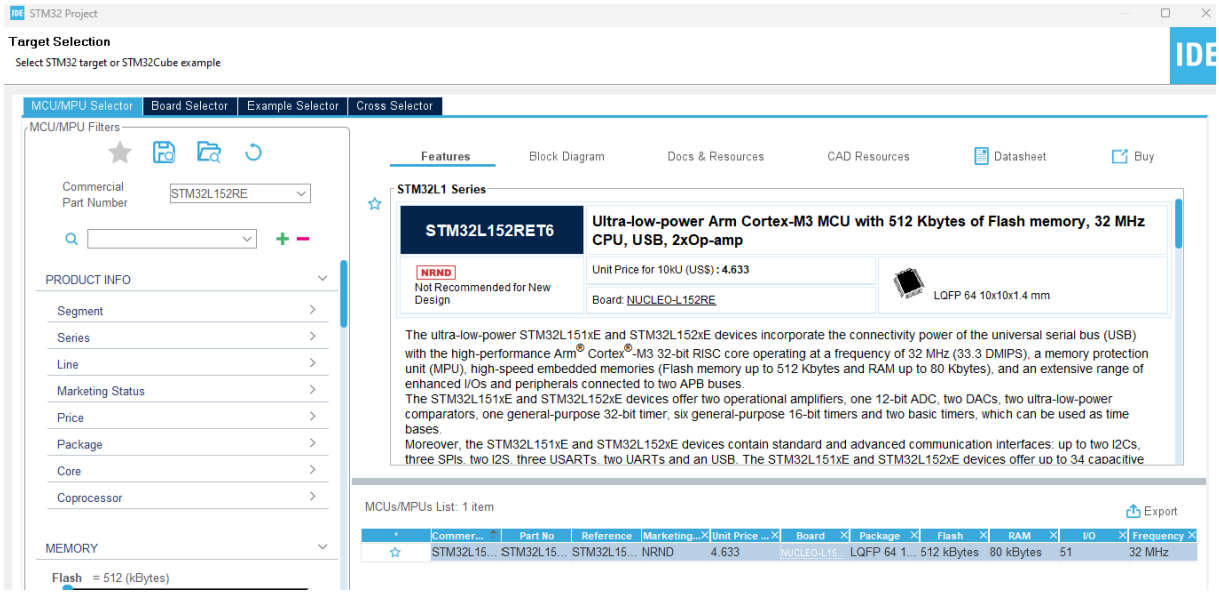
4 Przebieg ćwiczenia

Przebieg ćwiczenia zostanie podany w formie kroków postępowania, które należy wykonać, żeby opracować program sterujący sygnalizacją świetlną.

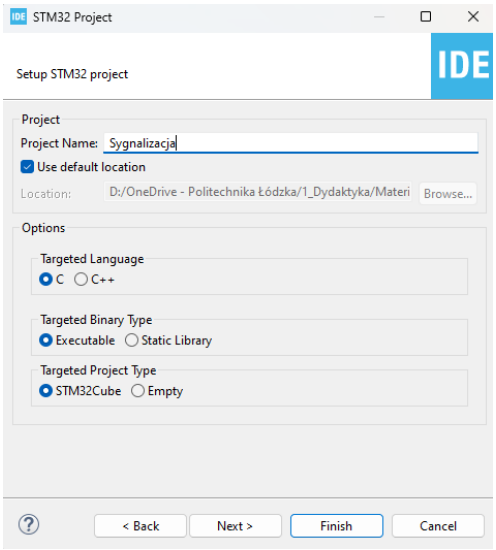
1) Włączyć komputer, a następnie utworzyć w *Moje Dokumenty* folder o nazwie np. *Skrzyzowanie*. UWAGA: PODCZAS NADAWANIA NAZW FOLDERÓW I PLIKÓW NIE MOŻA UŻYWAĆ POLSKICH ZNAKÓW ORAZ ZNAKÓW INTERPUNKCJYJNYCH! OPROGRAMOWANIE STM32CubeIDE GENERUJE BŁĘDY PODCZAS KOMPILACJI W RAZIE UŻYCIA TYCH ZNAKÓW. RASUMUJĄC, CAŁA ŚCIEŻKA DOSTĘPU DO FOLDERU, W KTÓRYM ZNAJDUJĄ SIĘ PLIKI Z KODEM, JAKI I SAME PLIKI NIE MOGĄ MIEĆ POLSKICH ZNAKÓW W NAZWACH.
2) Uruchomić oprogramowaniem *STM32CubeIDE* i wybrać powyższy folder jako *workspace* naszego projektu.

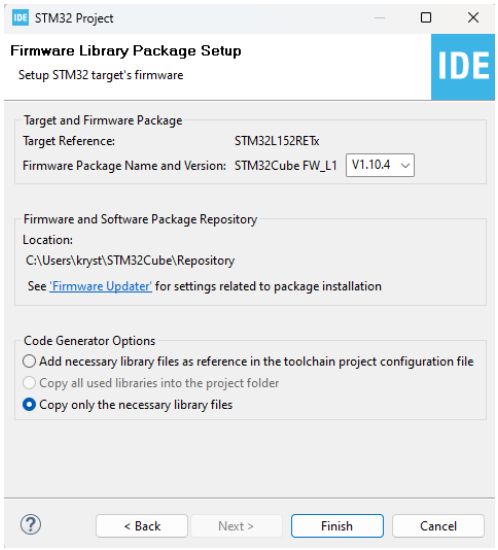


3) W lewym górnym rogu programu wybieramy zakładkę *File →New→SMT32 Project*
4) W tym kroku należy wybrać rodzaj mikrokontrolera STM, który używamy w ćwiczeniu. W okienku *Commercial Part Number* wpisujemy *STM32L152RE* i wybieramy go z listy, która wyświetla się w dole okna. Następnie klikamy *Next >*

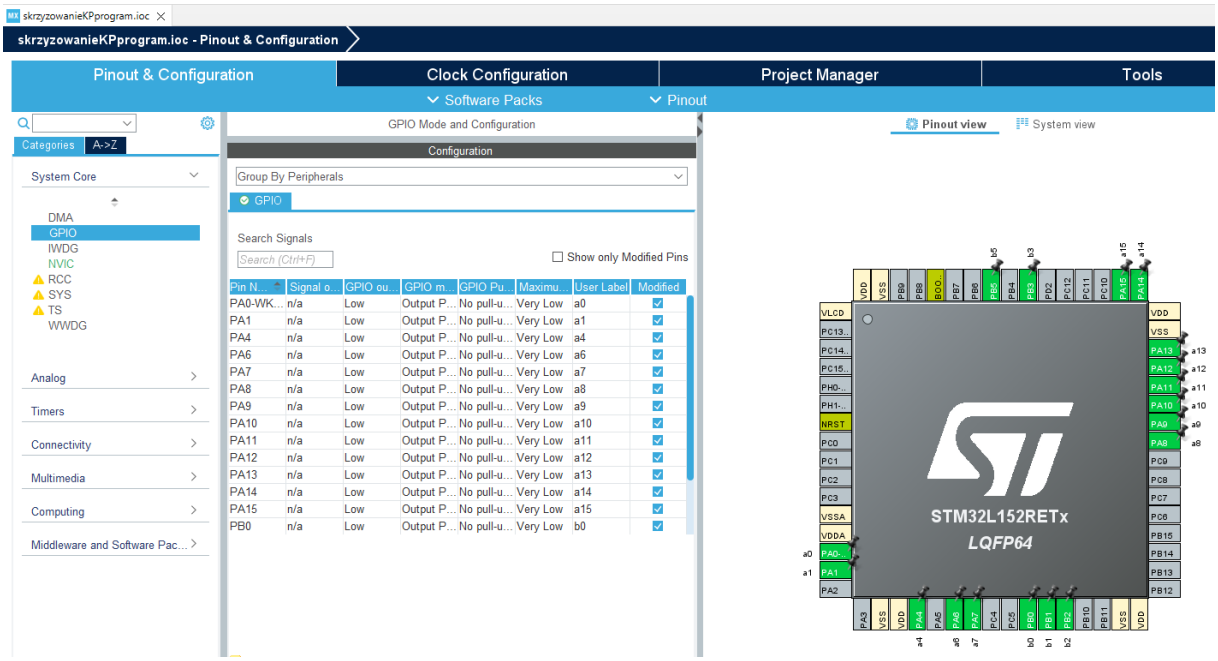


5) W okienku *Project Name* wpisujemy nazwę naszego projektu np. *Sygnalizacja*. Resztę wartości pozostawiamy w ustawieniach domyślnych i klikamy *Next>*. W następnym okienku klikamy *Finish.*

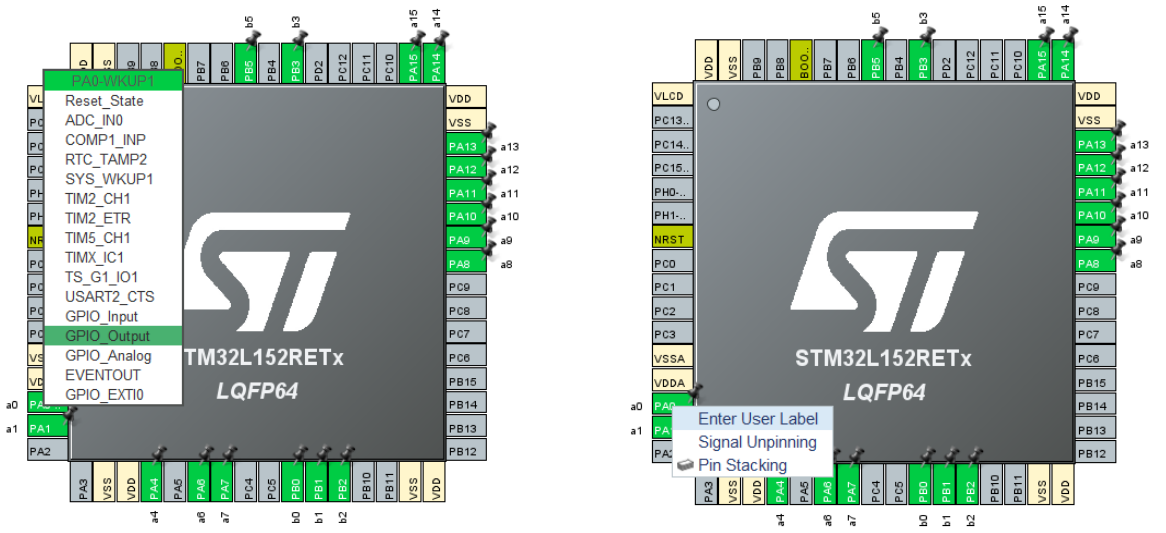




6) W tym kroku zdefiniujemy wyjścia mikrokontrolera. Zgodnie z tabelą 2 przypiszemy odpowiednim pinom kontrolera, czy mają pracować jako wejścia, czy jako wyjścia oraz nadamy im nazwy, którymi później w programie będziemy się posługiwać. Piny mikrokontrolera podzielone są na trzy porty: Port A (PA0, PA1,…), Port B (PB0, PB1,…) i Port C (PC0, PC1,…).
W wyświetlonym w programie oknie *Pinout&Configuration* widoczne jest zdjęcie naszego mikrokontrolera wraz z pinami, a po jego lewej stronie znajduje się lista z zakładkami *System Core*, *Analog*, *Timers*, itd. Rozwijamy listę *System Core → GPIO*, w której będziemy mieli podgląd wszystkich pinów i ich opcji.

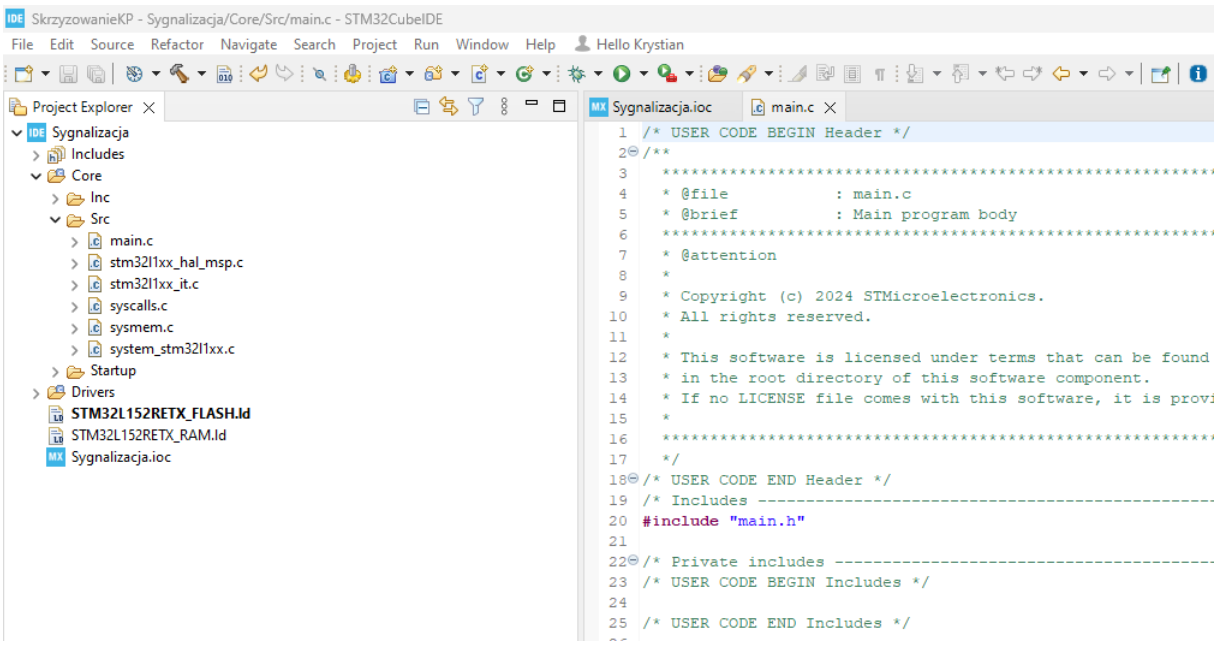


Przykładowo zgodnie z tabelą pin PA0 ma pracować jako wyjście, aby to ustawić należy kliknąć LPM na ten pin na zdjęciu mikrokontrolera i wybrać opcję *GPIO_Output.* Następnie, żeby nadać pinowi nazwę *PPM→Enter User Label* i wpisujemy *„a1”*. Analogicznie robimy z resztą pinów zgodnie z tabelą 2, wszystkie piny pracować będą jako wyjścia.



Po ustawieniu wszystkich opcji klikamy *File → Save.* Następnie wyskoczy okno z zapytaniem, czy wygenerować kod? Zgadzamy się klikając *Yes.* W następnym oknie potwierdzamy (*Yes*), że godzimy się na pracę z kodem napisanym w języku *C/C++.*

7) Przechodzimy teraz do struktury kodu. Kod programu znajduje się w pliku o nazwie *main.c*. Ścieżka dostępu do tego pliku znajduje się po lewej stronie okna w tzw. „drzewku" i jest następująca *Sygnalizacja→Core→Src*. Z pozycji „drzewka" możemy też zmienić przypisania pinów z pkt 6 wchodząc w plik *Sygnalizacja.ioc*, po wprowadzeniu zmian w tym pliku należy pamiętać o ich zapisaniu (*save*).

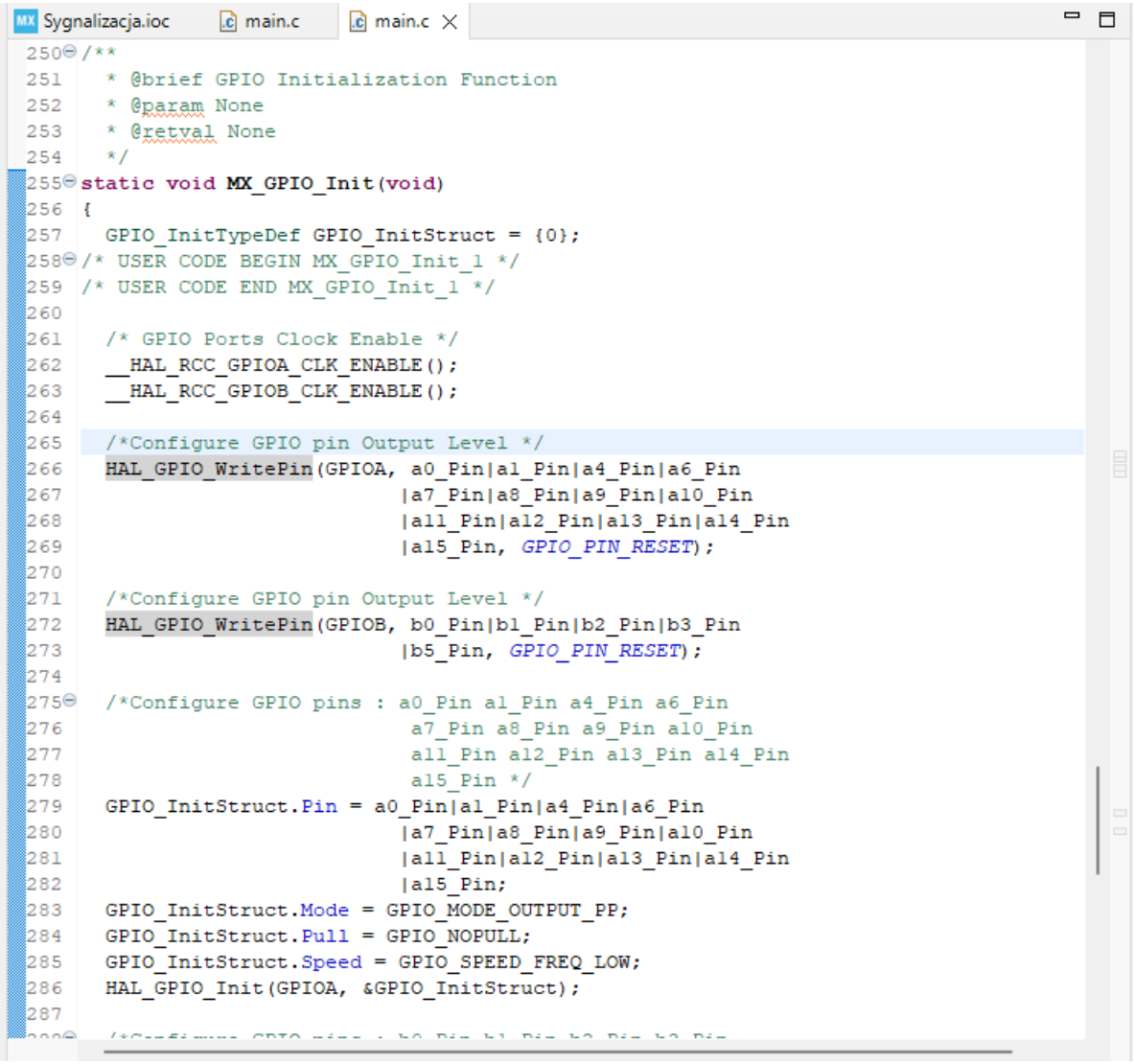


W środkowej części okna wygenerowany został „szkielet" naszego programu. W szkielecie tym naszą uwagę przyciąga duża liczba linijek zielonych komentarzy.

Komentarze te dzielą nam nasz szkielet kodu na poszczególne sekcje. Każda z sekcji ma swoje zadanie i może zawierać określone dla tej sekcji dane, dlatego podczas pisania kodu należy w

odpowiednim miejscu szkieletu wpisać odpowiednie linijki kodu. NIE NALEŻY USUWAĆ TYCH KOMENTARZY PONIEWAŻ PODCZAS KOMPILACJI PROGRAMU KOMPILATOR SZUKA ODPOWIEDNICH LINIJEK KODU WŁAŚNIE DZIĘKI TYM KOMENTARZOM. USUNIĘCIE TYCH KOMENTARZY BĘDZIE SKUTKOWAĆ BŁĘDAMI PODCZAS KOMPILACJI I NIEDZIAŁĄNIEM PROGRAMU!

Jako przykład możemy zobaczyć, że na końcu naszego szkieletu znajdują się sekcje, które na podstawie działań z pkt 6 utworzyły linijki kodu definiujące jakie działanie mają mieć piny:




```
250 /**
251   * @brief GPIO Initialization Function
252   * @param None
253   * @retval None
254   */
255 static void MX_GPIO_Init(void)
256 {
257   GPIO_InitTypeDef GPIO_InitStruct = {0};
258 /* USER CODE BEGIN MX_GPIO_Init_1 */
259 /* USER CODE END MX_GPIO_Init_1 */
260
261   /* GPIO Ports Clock Enable */
262   __HAL_RCC_GPIOA_CLK_ENABLE();
263   __HAL_RCC_GPIOB_CLK_ENABLE();
264
265   /*Configure GPIO pin Output Level */
266   HAL_GPIO_WritePin(GPIOA, a0_Pin|a1_Pin|a4_Pin|a6_Pin
267                           |a7_Pin|a8_Pin|a9_Pin|a10_Pin
268                           |a11_Pin|a12_Pin|a13_Pin|a14_Pin
269                           |a15_Pin, GPIO_PIN_RESET);
270
271   /*Configure GPIO pin Output Level */
272   HAL_GPIO_WritePin(GPIOB, b0_Pin|b1_Pin|b2_Pin|b3_Pin
273                           |b5_Pin, GPIO_PIN_RESET);
274
275   /*Configure GPIO pins : a0_Pin a1_Pin a4_Pin a6_Pin
276                            a7_Pin a8_Pin a9_Pin a10_Pin
277                            a11_Pin a12_Pin a13_Pin a14_Pin
278                            a15_Pin */
279   GPIO_InitStruct.Pin = a0_Pin|a1_Pin|a4_Pin|a6_Pin
280                           |a7_Pin|a8_Pin|a9_Pin|a10_Pin
281                           |a11_Pin|a12_Pin|a13_Pin|a14_Pin
282                           |a15_Pin;
283   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
284   GPIO_InitStruct.Pull = GPIO_NOPULL;
285   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
286   HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
287
```

8) Zacznijmy więc pisać kod sterujący naszym układem bazując na automacie skończonym. W tabeli 2 przyjęliśmy, że działanie naszego układu (cykl) możemy podzielić na 12 różnych faz od 0 do 11. W każdej fazie układ zapala odpowiednie światła na danych sygnalizatorach, np. w fazie 0 na wszystkich sygnalizatorach palą się tylko światła czerwone, tzn. że na piny mikrokontrolera odpowiadające tym światłom wysyłane jest 5V, czyli stan wysoki (logiczna 1), natomiast na inne piny wysyłane jest 0 V (logiczne 0). Następnie po upływie ustalonego przez nas czasu, np. 5 sekund, układ ma przejść do fazy 1 i zapalić lub zgasić odpowiadające tej fazie piny.

Zacznijmy nasz kod od zdefiniowania struktury, która będzie przechowywała informację o pojedynczej fazie działania systemu świateł. Odszukujemy w szkielecie naszego programu komentarz:

```
/* Private typedef -----------------------------------------------------
------*/
/* USER CODE BEGIN PTD */
```

i pod nim zamieszczamy następujący fragment kodu:

```
/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

typedef struct {
    uint32_t duration;       // czas trwania fazy w milisekundach
    uint16_t lights_on_A;    // bity reprezentujące piny świateł do włączenia w GPIOA
    uint16_t lights_off_A;   // bity reprezentujące piny świateł do wyłączenia w GPIOA
    uint16_t lights_on_B;    // bity reprezentujące piny świateł do włączenia w GPIOB
    uint16_t lights_off_B;   // bity reprezentujące piny świateł do wyłączenia w GPIOB
} Phase;
```

Ten fragment kodu definiuje strukturę danych o nazwie `Phase`, która służy do przechowywania informacji o pojedynczej fazie działania systemu świateł. Struktura jest używana do definiowania zestawu poleceń dla każdej fazy systemu:

- ✓ Jakie piny włączyć/wyłączyć w dwóch portach (GPIOA i GPIOB).
- ✓ Jak długo faza ma trwać.

Dzięki temu, zamiast pisać osobny kod dla każdej fazy, można łatwo przechowywać dane w tablicy struktur i przetwarzać je w pętli.

Struktura zawiera pięć pól, a ich znaczenie to:

1. **`uint32_t duration;`**
   - **Opis:** Przechowuje czas trwania danej fazy w milisekundach.
   - **Znaczenie:** Określa, jak długo dana faza będzie aktywna, zanim program przejdzie do kolejnej.
2. **`uint16_t lights_on_A;`**
   - **Opis**: Przechowuje informacje o pinach portu GPIOA, które mają zostać włączone (ustawione na stan wysoki).
   - **Znaczenie**: Bity w tej zmiennej reprezentują konkretne piny portu GPIOA. Gdy dany bit jest ustawiony na 1, odpowiadający mu pin zostanie aktywowany.
3. **`uint16_t lights_off_A;`**
   - **Opis**: Przechowuje informacje o pinach portu GPIOA, które mają zostać wyłączone (ustawione na stan niski).
   - **Znaczenie**: Bity w tej zmiennej reprezentują konkretne piny portu GPIOA. Gdy dany bit jest ustawiony na 1, odpowiadający mu pin zostanie dezaktywowany.
4. **`uint16_t lights_on_B;`**
   - **Opis**: Przechowuje informacje o pinach portu GPIOB, które mają zostać włączone (ustawione na stan wysoki).
   - **Znaczenie**: Działa analogicznie do `lights_on_A`, ale dla portu GPIOB.
5. **`uint16_t lights_off_B;`**
   - **Opis**: Przechowuje informacje o pinach portu GPIOB, które mają zostać wyłączone (ustawione na stan niski).
   - **Znaczenie**: Działa analogicznie do `lights_off_A`, ale dla portu GPIOB.

Typy danych `uint16_t` i `uint32_t` są całkowitymi typami bez znaku używanymi w języku C. Typ `uint16_t` ma zakres wartości: od 0 do 65 535 i używany jest do przechowywania małych liczb całkowitych bez znaku, np. wartości bitów w rejestrze, flag sterujących lub pinów GPIO. Typ danych `uint32_t` ma zakres wartości: od 0 do 4 294 967 295 i używany jest do przechowywania większych liczb całkowitych bez znaku, np. liczników czasu, wskaźników pamięci lub danych konfiguracyjnych.

9) Teraz w celu skrócenia i uproszczenia zapisów w dalszej części programu oraz poprawieniu czytelność kodu, zamiast pełnych nazw pinów (a0_Pin) opracujemy definicję dzięki, którym można będzie używać krótszych nazw (a0). Szukamy w szkielecie kodu następującego komentarza:

```
/* Private define ------------------------------------------------------
------*/
/* USER CODE BEGIN PD */
```

i wprowadzamy następujące linijki kodu:

```
42 /* Private define ------------------------------------------------------------*/
43 /* USER CODE BEGIN PD */
44
45 #define a0 a0_Pin
46 #define a1 a1_Pin
47 #define a4 a4_Pin
48 #define a6 a6_Pin
49 #define a7 a7_Pin
50 #define a8 a8_Pin
51 #define a9 a9_Pin
52 #define a10 a10_Pin
53 #define a11 a11_Pin
54 #define a12 a12_Pin
55 #define a13 a13_Pin
56 #define a14 a14_Pin
57 #define a15 a15_Pin
58
59 #define b0 b0_Pin
60 #define b1 b1_Pin
61 #define b2 b2_Pin
62 #define b3 b3_Pin
63 #define b5 b5_Pin
64
65 #define NUM_PHASES 12
66
67 /* USER CODE END PD */
68
```

1. Makra zdefiniowane za pomocą `#define`
   - Makra `#define` są używane do tworzenia stałych tekstowych (zamienników) w kodzie, które są później zastępowane przez preprocesor podczas kompilacji.
   - Dzięki nim można używać bardziej intuicyjnych nazw zamiast pełnych identyfikatorów.
2. Definicje pinów GPIO (`a0`, `a1`, ..., `b0`, `b1`, ...)
   - Definicje w postaci `#define a0 a0_Pin` i podobne przypisują bardziej czytelną nazwę (`a0`) do identyfikatora wygenerowanego automatycznie przez CubeMX (np. `a0_Pin`).
   - Każda nazwa odpowiada konkretnemu pinowi GPIO mikrokontrolera.
3. NUM_PHASES
   - Definicja `#define NUM_PHASES 12` wprowadza stałą `NUM_PHASES`, która określa liczbę faz w systemie. Jest używana w innych częściach kodu, np. do deklaracji tablicy faz. Stała ta zapewnia elastyczność — w przypadku zmiany liczby faz wystarczy zmodyfikować jedną wartość, a nie całą logikę programu.

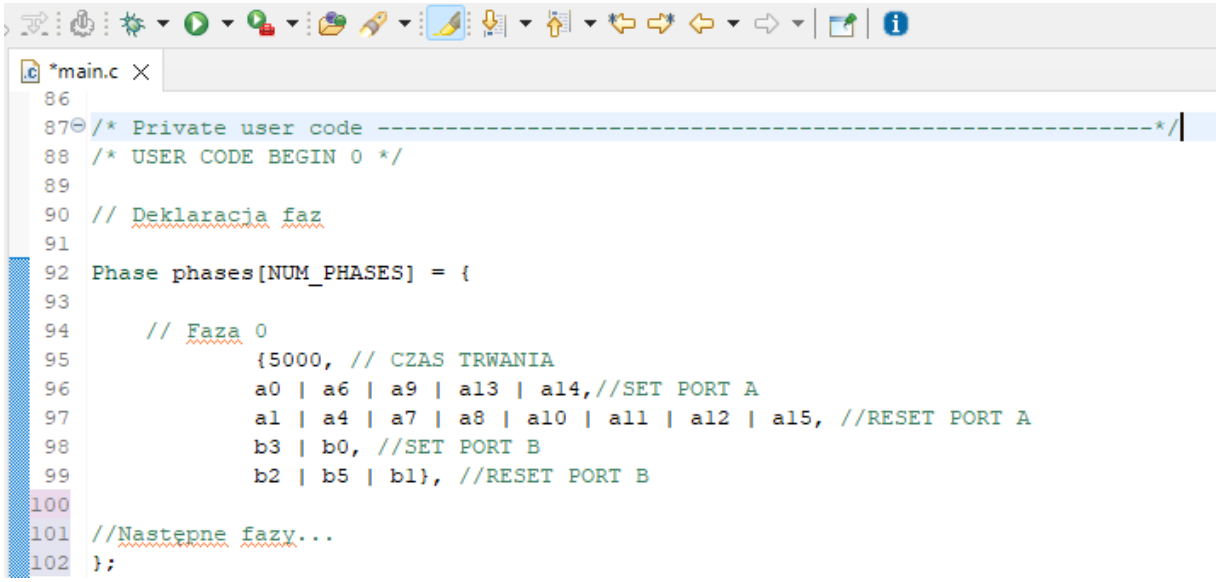
10) Przejdziemy teraz do napisania fragmentu kodu definiującego tablicę struktur `phases` typu `Phase`, która przechowuje informację o konfiguracji świateł dla każdej z 12 faz (zdefiniowanych przez stałą `NUM_PHASES`). Wykorzystamy do tego dane z tabeli 2.
    1. Tablica `phases`
       - Tablica `phases` zawiera obiekty typu `Phase` (wcześniej zdefiniowana struktura danych, patrz pkt 8).
       - Każdy element tablicy odpowiada jednej fazie sygnalizacji świetlnej i przechowuje:
         ✓ czas trwania fazy w milisekundach,

- ✓ które piny GPIO mają być włączone na portach A i B (`lights_on_A`, `lights_on_B`),
- ✓ które piny GPIO mają być wyłączone na portach A i B (`lights_off_A`, `lights_off_B`).

Definicje poszczególnych faz:

- Każda faza w tablicy jest opisana w nawiasach klamrowych `{}` w formie listy wartości przypisanych do pól struktury `Phase`.

Przykład analizy jednej fazy – Faza 0:



- **5000**: Faza trwa 5000 ms (5 sekund).
- **a0|a6|a9|a13|a14**: Piny a0, a6, a9, a13 i a14 na porcie A zostaną ustawione na logiczną 1.
- **a1|a4|a7|a8|a10|a11|a12|a15**: Piny a1, a4… na porcie A zostaną ustawione na logiczne 0.
- **b3 | b0**: Piny b3 i b0 na porcie B zostaną ustawione na logiczną 1.
- **b2|b5|b1**: Piny b2, b5… na porcie B zostaną ustawione na logiczne 0.

Każda faza ma podobną strukturę, ale inne konfiguracje świateł (np. inne piny do włączenia lub wyłączenia).

Na podstawie powyższego przykładu fragmentu kodu dla fazy 0, należy opracować resztę kodu dla następnych faz.

11) W tym kroku zdefiniujemy funkcję `execute_phase`, która wykonuje działania związane z włączeniem i wyłączeniem świateł (pinów GPIO) dla konkretnej fazy na portach GPIOA i GPIOB. Funkcja przyjmuje jeden parametr:
- `phase_index` (typ `uint8_t`): indeks fazy w tablicy `phases`, który wskazuje, jakie piny mają zostać włączone lub wyłączone.

Funkcję tę należy umieścić w linijce zaraz pod tablicą `phases` z pkt 10.

```
136 void execute_phase(uint8_t phase_index) {
137     // Włącz światła na porcie GPIOA dla bieżącej fazy
138     HAL_GPIO_WritePin(GPIOA, phases[phase_index].lights_on_A, GPIO_PIN_SET);
139     // Wyłącz światła na porcie GPIOA dla bieżącej fazy
140     HAL_GPIO_WritePin(GPIOA, phases[phase_index].lights_off_A, GPIO_PIN_RESET);
141
142     // Włącz światła na porcie GPIOB dla bieżącej fazy
143     HAL_GPIO_WritePin(GPIOB, phases[phase_index].lights_on_B, GPIO_PIN_SET);
144     // Wyłącz światła na porcie GPIOB dla bieżącej fazy
145     HAL_GPIO_WritePin(GPIOB, phases[phase_index].lights_off_B, GPIO_PIN_RESET);
146 }
147
148 /* USER CODE END 0 */
149
```

Działanie funkcji krok po kroku:

1. **Włączenie świateł na GPIOA:**
   `HAL_GPIO_WritePin(GPIOA, phases[phase_index].lights_on_A, GPIO_PIN_SET);`
   - Funkcja `HAL_GPIO_WritePin` ustawia wskazane piny na porcie `GPIOA` na stan **HIGH** (włączenie).
   - `phases[phase_index].lights_on_A` wskazuje bity (piny), które mają być ustawione na HIGH dla tej fazy.
2. **Wyłączenie świateł na GPIOA:**
   `HAL_GPIO_WritePin(GPIOA, phases[phase_index].lights_off_A, GPIO_PIN_RESET);`
   - Funkcja `HAL_GPIO_WritePin` ustawia wskazane piny na porcie `GPIOA` na stan **LOW** (wyłączenie).
   - `phases[phase_index].lights_off_A` wskazuje bity (piny), które mają być ustawione na LOW dla tej fazy.
3. **Włączenie i wyłączenie świateł na GPIOB ma podobną strukturę jak dla GPIOA.**

Funkcja `execute_phase` odpowiada za fizyczne ustawienie odpowiednich pinów GPIO zgodnie z konfiguracją bieżącej fazy w tablicy `phases`. Dzięki temu system steruje światłami zgodnie z zadanymi ustawieniami.

12) Przed przejściem do zasadniczego fragmentu kodu wykonującego w zapętleniu nasz cykl zmiany świateł, w szkielecie kodu odszukujemy sekcję opisaną komentarzem:

```
/* Initialize all configured peripherals */
MX_GPIO_Init();
/* USER CODE BEGIN 2 */
```

W tej sekcji wprowadzimy fragment kodu odpowiadający za inicjalizację systemu oraz przygotowanie zmiennych sterujących fazami.

```
226   /* Initialize all configured peripherals */
227   MX_GPIO_Init();
228   /* USER CODE BEGIN 2 */
229
230   uint8_t current_phase = 0;
231   uint32_t phase_start_time = HAL_GetTick();
232   execute_phase(0% NUM_PHASES);
233
234   /* USER CODE END 2 */
```

Wyjaśnijmy go szczegółowo:

1. **Inicjalizacja skonfigurowanych peryferiów:**

   `MX_GPIO_Init();`

- Funkcja `MX_GPIO_Init()` jest generowana automatycznie przez STM32CubeMX.
- Odpowiada za skonfigurowanie pinów GPIO zgodnie z ustawieniami projektu, np. jako wyjścia, wejścia, z włączonymi rezystorami pull-up/pull-down itp.
- W naszym przypadku konfiguruje piny GPIOA i GPIOB jako wyjścia, aby sterować diodami (światłami) w projekcie.

2. **Deklaracja zmiennej `current_phase`:**

```
uint8_t current_phase = 0;
```

- `uint8_t`: Typ zmiennej przechowującej liczby całkowite od 0 do 255 (8 bitów bez znaku).
- `current_phase`: Przechowuje numer aktualnej fazy sygnalizacji świetlnej.
- `= 0`:
  - ✓ Domyślnie inicjalizujesz tę zmienną wartością `0`.
- `execute_phase(current_phase % NUM_PHASES)`: mikrokontroler ustawia na wyjściach stany wysokie i niskie zgodnie z obecną fazą cyklu `current_phase` równą `0`.

3. **Deklaracja i inicjalizacja `phase_start_time`:**

```
uint32_t phase_start_time = HAL_GetTick();
```

- `uint32_t:` Typ zmiennej przechowujący liczby całkowite od 0 do 4 294 967 295 (32 bity bez znaku).
- `phase_start_time:`
  - ✓ Ta zmienna przechowuje czas w milisekundach, kiedy rozpoczęła się aktualna faza.
  - ✓ Funkcja `HAL_GetTick()` zwraca czas systemowy w milisekundach od momentu uruchomienia mikrokontrolera.
  - ✓ Dzięki przypisaniu wyniku `HAL_GetTick()` podczas inicjalizacji możesz rozpocząć odliczanie czasu dla pierwszej fazy.

Podsumowując w tym fragmencie kodu inicjalizowane są wszystkie skonfigurowane piny GPIO oraz zadeklarowane zostają zmienne:

- `current_phase`, aby śledzić aktualną fazę sygnalizacji.
- `phase_start_time`, aby znać moment rozpoczęcia aktualnej fazy i móc poprawnie kontrolować czas jej trwania.

13) Przejdziemy teraz do ostatniego i zasadniczego fragmentu kodu, który zawiera główną pętlę programu (`while 1)`). Pętla ta działa w nieskończoność i kontroluje cykl działania systemu, przełączając się pomiędzy fazami zgodnie z ustalonym harmonogramem.

```
191   /* Infinite loop */
192   /* USER CODE BEGIN WHILE */
193   while (1)
194   {
195     /* USER CODE END WHILE */
196
197       if (HAL_GetTick() - phase_start_time >= phases[current_phase].duration) {
198                 // Przejdź do następnej fazy
199                 current_phase = (current_phase + 1) % NUM_PHASES;
200                 execute_phase(current_phase);
201                 phase_start_time = HAL_GetTick();
202             }
203
204     /* USER CODE BEGIN 3 */
205   }
206   /* USER CODE END 3 */
207 }
208
```

Oto szczegółowe wyjaśnienie krok po kroku:

4. **Nieskończona pętla główna:**

```
while (1){
```

- Pętla `while (1)` działa w nieskończoność, co jest standardowym podejściem w systemach opartych na mikrokontrolerach. Główne zadania programu są wykonywane wewnątrz tej pętli.

5. **Sprawdzenie, czy upłynął czas trwania bieżącej fazy:**

```
if (HAL_GetTick() - phase_start_time >= phases[current_phase].duration) {
```

- `HAL_GetTick()` zwraca liczbę milisekund, które upłynęły od uruchomienia mikrokontrolera.
- `phase_start_time` przechowuje moment rozpoczęcia bieżącej fazy.
- Wyrażenie `HAL_GetTick() - phase_start_time` oblicza czas, który upłynął od początku bieżącej fazy.
- `phases[current_phase].duration` przechowuje czas trwania bieżącej fazy.
- Jeśli upłynęło wystarczająco dużo czasu (równy lub większy od `duration`), następuje przejście do kolejnej fazy.

6. **Przejście do kolejnej fazy:**

```
current_phase = (current_phase + 1) % NUM_PHASES;
```

- `current_phase` to zmienna przechowująca indeks bieżącej fazy.
- `(current_phase + 1)` zwiększa indeks o 1, co oznacza przejście do następnej fazy.
- `% NUM_PHASES` zapewnia, że indeks fazy wraca do 0 po osiągnięciu ostatniej fazy. Dzięki temu cykl faz działa w sposób ciągły.

7. **Wykonanie nowej fazy:**

```
execute_phase(current_phase);
```

- Wywołanie funkcji `execute_phase` włącza i wyłącza odpowiednie światła dla nowej fazy, zgodnie z jej konfiguracją w tablicy `phases`.

8. **Aktualizacja czasu rozpoczęcia fazy:**

```
phase_start_time = HAL_GetTick();
```

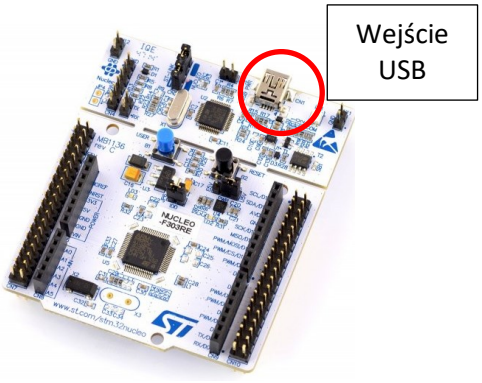
- Aktualizuje wartość `phase_start_time` na bieżący czas, aby rozpocząć odliczanie dla nowej fazy.

9. **Zamknięcie pętli i powrót do początku:**

```
}
```

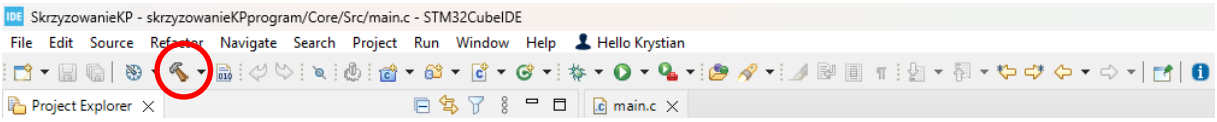
- Po zakończeniu bloku `if` program wraca do początku pętli `while (1)` i ponownie sprawdza, czy czas trwania bieżącej fazy został przekroczony.

Fragment ten jest kluczowym elementem programu, który steruje przełączaniem między fazami sygnalizacji świetlnej. Zapewnia, że każda faza trwa odpowiedni czas, a po zakończeniu ostatniej fazy cykl zaczyna się od nowa.

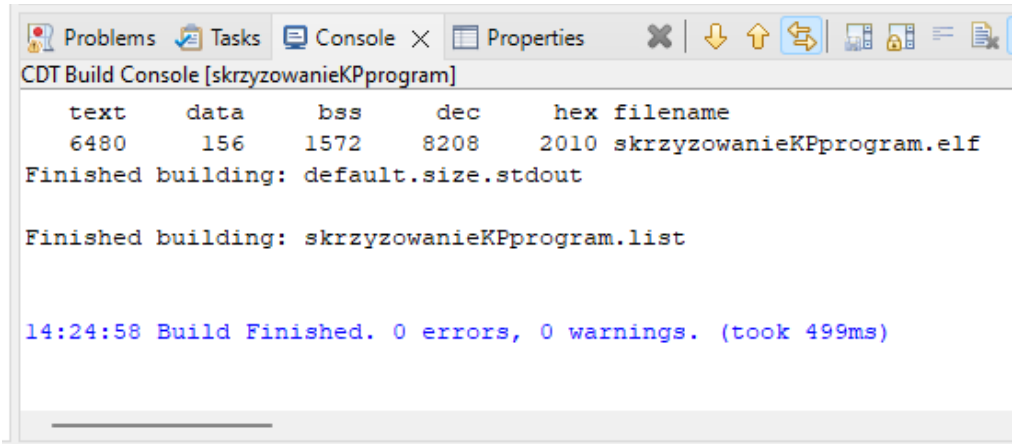


Rys. 6. Wejście USB mikrokontrolera.

14) Program mamy już napisany, teraz możemy przejść do jego zbudowania, skompilowania i wgrania na mikrokontroler. Zbudowanie programu wykonamy poprzez kliknięcie ikonki „młotka” znajdującej się na pasku głównym programu.



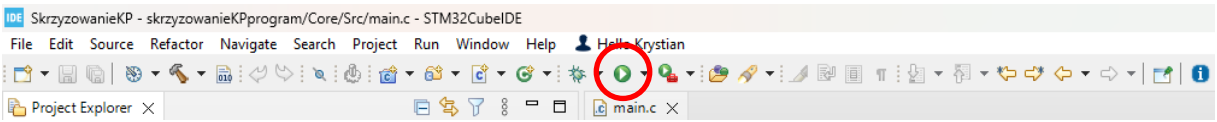
Jeśli dobrze napisaliśmy program po jego zbudowaniu w na dole programu, w oknie konsoli powinniśmy otrzymać potwierdzenie zbudowania programu i brak błędów.



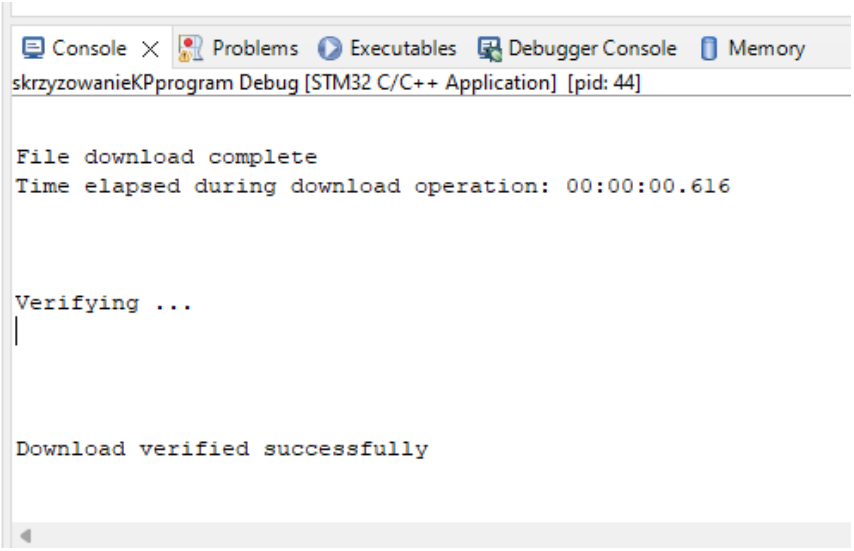
Następnie musimy podłączyć mikrokontroler do komputera. W tym celu posłuży nam kabel USB, którego jeden koniec podłączamy do wejścia USB komputera, a drugi do wejścia USB PŁYTKI 1 mikrokontrolera (

15) rys. 6). Po podłączeniu otworzy nam się folder z plikami zapisanymi w pamięci mikrokontrolera, folder ten możemy zamknąć.

Teraz, aby zdebugować i skompilować program należy nacisnąć ikonkę „PLAY” znajdującą się na pasku głównym programu *STM32CubeIDE.*



Po poprawnym zdebugowaniu i skompilowaniu kodu, w konsoli znajdującej się na dole głównego okna programu powinien wyświetlić się następujący komunikat:



Teraz należy odłączyć kabel USB od PŁYTKI 1 mikrokontrolera i podłączyć ten kabel do gniazda zasilania całego układu, tak jak pokazano to numerem (3) na rys. 3. W tym momencie rozpocznie się sterowanie sygnalizacją świetlną na skrzyżowaniu.

5 Bibliografia

[1] W. Traczyk, Układy cyfrowe automatyki, Wydawnictwo Naukowo-Techniczne, Warszawa 1974.
[2] J. Kalisz, Podstawy elektroniki cyfrowej, Wydawnictwo Komunikacji i Łączności, Warszawa 2002.
[3] T. Łuba, Synteza układów logicznych, Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa 2005.
[4] D. Schmid, Mechatronika, Warszawa: REA, 2008.
[5] M. Olszewski, Urządzenia i systemy mechatroniczne, Warszawa: REA, 2009.
[6] Microsoft, "Microsoft Learn Challenge: C++," Microsoft Learn, 2024. [Online]. Dostępny: https://learn.microsoft.com/en-us/training/topics/event-challenges. [Accessed: 25-Nov-2024].
[7] Forbot, "Kurs STM32 L4," Forbot, 2024. [Online]. Dostępny: https://forbot.pl/blog/kategoria/artykuly/stm32. [Accessed: 25-Nov-2024].

6 Sprawozdanie

W sprawozdaniu należy zamieścić:

1) Graf działania automatu sterowania sygnalizacją świetlną analizowanego skrzyżowania.
2) Zamieścić listę przyporządkowania pinów z zakładki *Pinout&Configuration* oraz screen mikrokontrolera obrazujący to przyporządkowanie.
3) Fragment kodu definiującego tablicę struktur `phases` typu `Phase`, przechowującą informację o konfiguracji świateł dla każdej z 12 faz.
4) Własne spostrzeżenia, obserwacje i wnioski.

Załącznik 1 – Schemat połączeń diod sygnalizatorów z wyjściami mikrokontrolera