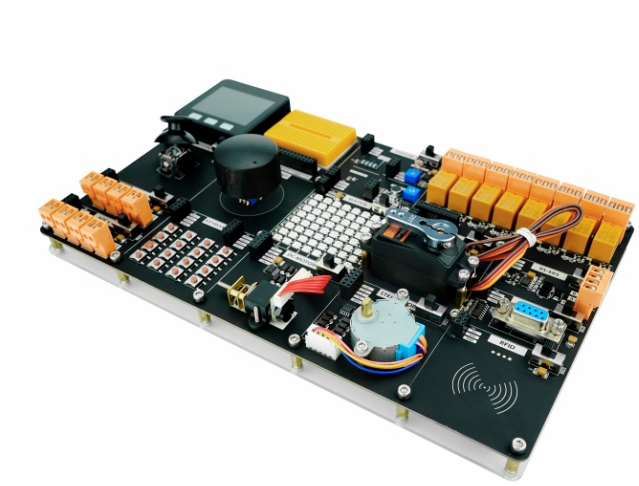
Programowanie w Elektrotechnice
Ćwiczenie Nr. 1

dr inż. Michał Łanczont

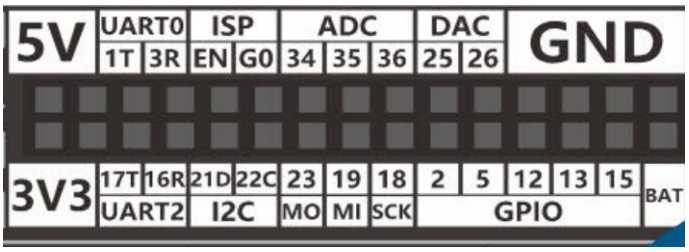
1) Wstęp

Celem postawionym przed przedmiotem jest zapoznanie studentów z możliwościami zastosowania systemów opartych o mikrokotroler w układach kontrolno-pomiarowych elementów elektrotechnicznych. Program ćwiczeń projektowych oparty został o układ **Demo Board** firmy **M5STACK** i mikrokontroler **M5Core**, na Rys. 1.1 pokazano zdjęcie platformy programistycznej.



Rys. 1.1 Platforma programistyczny DEMO BOARD z mikrokontrolerem M5Core

Na płycie Demo Board wyprowadzono porty mikrokontrolera do złącza stykowego. Opis dostępnych portów pokazano an Rys. 1.2.



Rys. 1.2 Złącza mikrokontrolera M5Core na płycie Demo Board

Układ może być programowany za pomocą dwóch podstawowych modeli programowania:

1. programowania graficznego w środowisku UIFlow Coding IDE, system oparty jest o język `PYTHON`
2. programowanie w języku ogólnego przeznaczenia `C/C++` w środowisku Arduino IDE w wersji 1.8.19 lub 2.*.

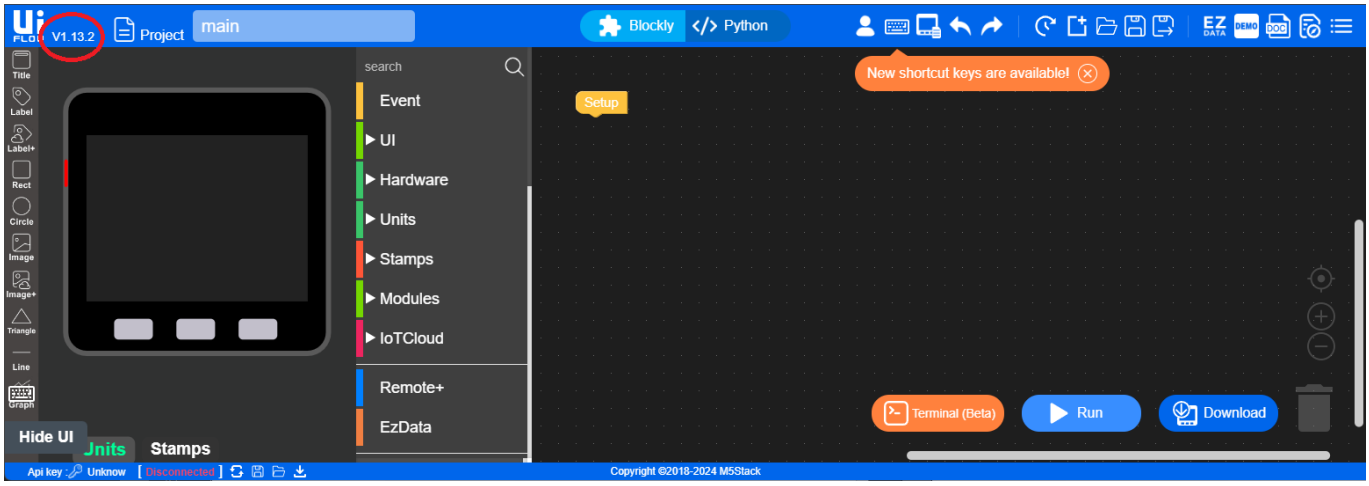
W realizacji zadań projektowych stosowane będą oba podejścia.

1.1 UIFlow Coding IDE

Programowanie graficzne polega na zapisaniu algorytmu w postaci diagramu blokowego zgodnego z specyfikacją środowiska programowania. Zapisany algorytm jest konwertowany na kod w języku PYTHON i wysyłany do pamięci mikrokontolera gdzie jest wykonywany przez zainstalowany na nim interpreter. Podejście takie wymaga aby na mikrokontroler zawsze wgrana była aktualna wersja interpretera. Wynika to z faktu że wraz z nim instalowana są aktualne biblioteki umożliwiające korzystanie z z aktualnych wersji narzędzi w trakcie tworzenia kodu graficznego. Aktualizacja oprogramowania mikrokontrolera realizowana jest przez dedykowane narzędzie M5Burner v3.0. Informację o zainstalowanej wersji interpretera można znaleźć na ekranie mikrokontrolera, jak pokazano na Rys. 1.3.

Rys. 1.3 Interfejs graficzny mikrokontrolera M5Core, numer wersji interpretera

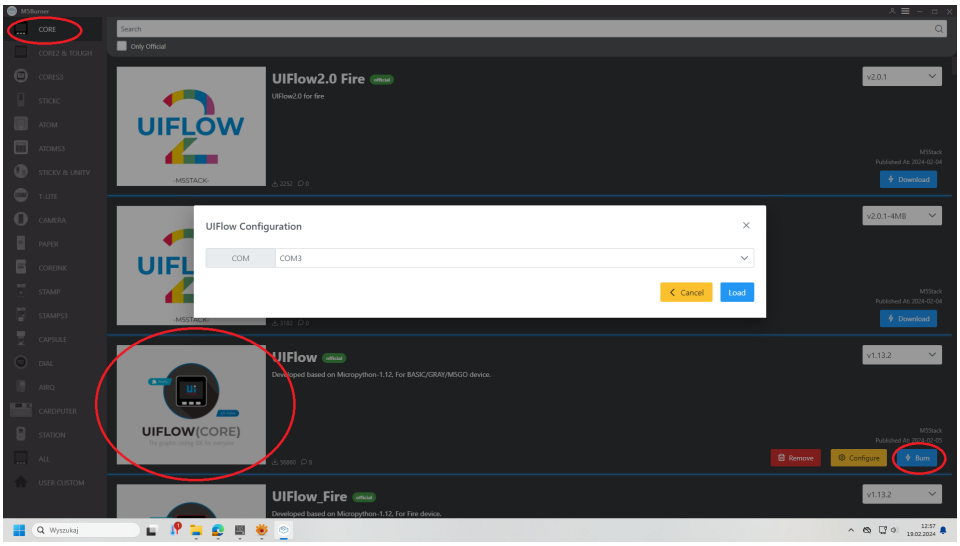
Weryfikację aktualnej wersji interpretera można dokonać w środowisku programowania graficznego, jak pokazano na Rys 1.4.



Rys. 1.4 Interfejs środowiska programistycznego UIFlow IDE, numer wersji interpretera

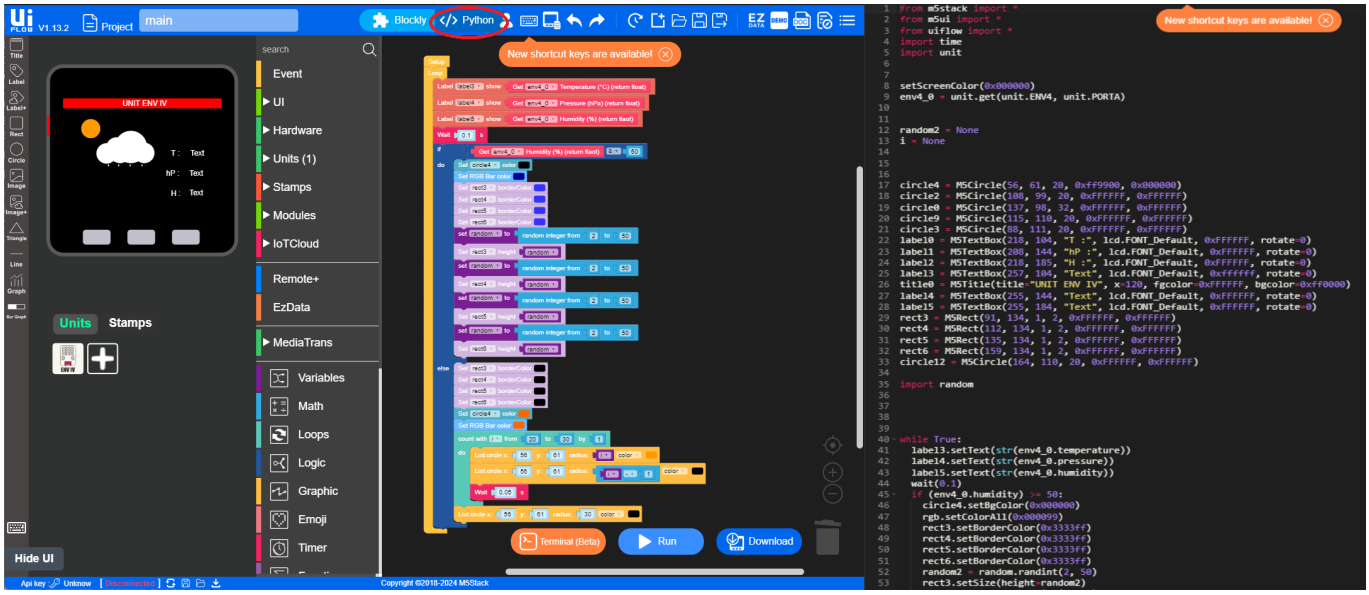
Po stwierdzeniu dostępności aktualizacji interpretera za pośrednictwem **M5Burner Tool** można zaktualizować oprogramowanie układowe mikrokontrolera, jak pokazano na Rys. 1.5. Po wybraniu właściwego modelu, wskazaniu portu USB do którego podpięte jest urządzenie i wybraniu wersji można wgrać właściwe oprogramowanie. Ze względu na to że środowisko programowania **UIFlow IDE** dostępne jest w wersji 1.* i 2.*, wgrać można firmware w wersji dedykowanej do wybranej wersji środowiska. Ze względu

na status "rozwojowy"wersji 2.8, a co zatem idzie ograniczoną dostępność narzędzi, zalecane jest stosowanie oprogramowania i środowiska dla rodziny 1.*.



Rys. 1.5 Interfejs M5Burner Tool

W środowisku **UIFlow IDE** projektuje się graficzną postać algorytmu za pomocą dostępnych bloków wykonawczych, jak pokazano na Rys. 1.6 , a kod wygenerowany na podstawie algorytmu jest wysyłany na mikrokontroler do wykonania.



Rys. 1.6 Kod graficzny przykładowej aplikacji

1.2 Arduino IDE

Środowisko **Arduino IDE** pozwala na zapisanie algorytmu programu w postaci kodu w języku opartym na C\C++. Jest ono wyspecjalizowanym tekstowym edytorem kodu wspierającym proces pisania programu, zintegrowanym z potężna bazą bibliotek i kompilatorów. Dzięki temu możliwe jest tworzenie kodu aplikacji dla mikrokontrolerów firm trzecich (nie tylko **Arduino**). W przypadku braku biblioteki lub kompilatory dla stosowanego mikrokontrolera lub elementu możliwe jest ręczne dodanie wymaganych komponentów z archiwów dostępnych na stronie producenta lub innych źródłach (np. ***GitHub***).

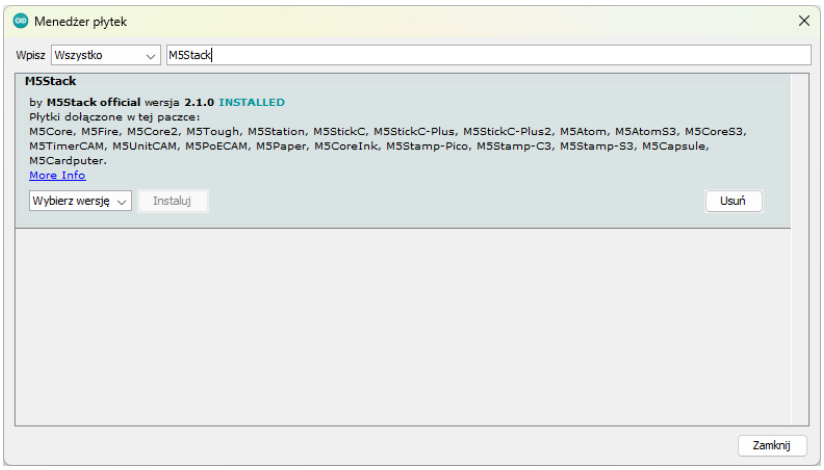
W przypadku mikrokontrolerów **M5Stack** producent dostarcza bazę bibliotek na swoim serwerze którą można dodać do linkowania w środowisku **Arduino IDE**. Szczegółowa dokumentacja dodania re-

pozytoriów **M5Stack** do **Arduino IDE** jest dostępna na stronie producenta i wymaga dodania w oknie konfiguracyjnym (**Preferencje**) środowiska Arduino IDE dodatkowego adresu URL źródeł:

```
https://m5stack.oss-cn-shenzhen.aliyuncs.com/resource/arduino
                                        /package_m5stack_index.json
```

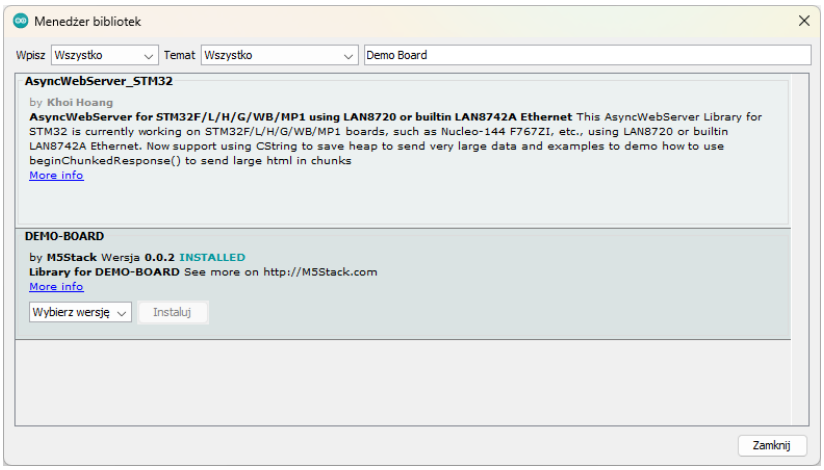
Po dodaniu w bazie mikrokontrolerów i bibliotek elementów dostępne będą nowe obiekty powiązane z urządzeniami firmy **M5Stack**. Dla potrzeb ćwiczeń projektowych realizowanych w ramach przedmiotu konieczne jest dodanie:

- z poziomu ***Menadżera Płytek*** kompilatorów dedykowanych dla rodziny mikrokontrolerów **M5Stack**, jak pokazano na Rys. 1.7.



Rys. 1.7 Okno menadżera płytek

- z poziomu ***Bazy Bibliotek*** (CTRL+SHIFT+l) zestaw powiązany z **Demo Board**, jak pokazano na Rys. 1.8.



Rys. 1.8 Okno menadżera bibliotek

Należny pamiętać, że do poprawnej pracy **Arduino IDE** wymaga zainstalowania interpretera Java SE Runtime Environment. Przy świeżo zainstalowanej wersji środowiska **Arduino IDE** wystarczy dodać biblioteki dla **Demo Board**, a kompilatory dla mikrokontrolerów **M5Stack** zostaną zainstalowane automatycznie. W przypadku korzystania z środowiska **Arduino IDE** w wersji 2.* procedura przebiega analogicznie. Jeżeli na komputerze zainstalowane są obydwie wersje rozwojowe środowiska należy pamiętać że biblioteki są przez nie współdzielone, wystarczy więc zainstalować wymagane komponenty tylko w jednej z wersji programu. Podstawową zaletą nowej rodziny środowiska **Arduino IDE** jest wbudowany

IntelliSense, czyli narzędzie podpowiadające składnię, np. listę metod dla danego obiektu czy listę i typy atrybutów). Jednakże wersja 1.* zapewnia lepszą kompatybilność z rozszerzeniami (np. instalacja bibliotek).

2) Podstawy programowania w środowisku Arduino IDE

Kod każdej aplikacji pisanej w Arduino IDE tworzony jest w oparciu o szablon pokazany poniżej, a składający się z dwóch funkcji `setup()` i `loop()`.

Listing 1: Szablon kodu Arduino IDE

```
1 void setup() {
2   // put your setup code here, to run once:
3
4 }
5
6 void loop() {
7   // put your main code here, to run repeatedly:
8
9 }
```

Pierwsza z nich wykonywana jest automatycznie w momencie uruchomienia programu na mikrokontrolerze, natomiast kod drugiej wykonywany jest w pętli nieskończonej po wykonaniu kodu z funkcji `setup()`.

Podstawowe składnia kodu jest zgodna z specyfikacją języków `C\C++`:

- deklaracje zmiennych, tablice, struktur, klas,
- dynamiczna alokacja pamięci,
- pętle (iteracyjne i warunkowe),
- warunki,

a także zaawansowane technologie, jak np. wątki.

2.1 Zmienne

W środowisku Arduino IDE, bazującym na `C\C++`, dostępne typy danych można podzielić na trzy podstawowe grupy: typowe, złożone i specyficzne. Deklarując i korzystając z zmiennych należy pamiętać o specyfice mikrokontrolerów, które w odróżnieniu od maszyn typu PC dysponują bardzo ograniczoną wielkością pamięci. Kontrola jej użycia staje się więc bardzo istotna przy projektowaniu kodu aplikacji. Dobrym rozwiązaniem może być dynamiczna alokacja pamięci, pozwalająca na efektywniejsze zarządzanie wielkością tablicami.

- Podstawowe typy danych
  - **boolean**: Przechowuje wartość `true` lub `false`.
  - **char**: Przechowuje pojedynczy **znak**(np. **literę**).
  - ***unsigned* char** lub **byte**: Liczba całkowita bez znaku w zakresie od `0` do `255`.
  - **int** lub **short**: Liczba całkowita ze znakiem w zakresie od `-32 768` do `32 767`.
  - ***unsigned* int** lub **word**: Liczba całkowita bez znaku w zakresie od `0` do `65 535`.
  - **long**: Liczba całkowita ze znakiem w zakresie od `-2 147 483 648` do `2 147 483 647`.
  - ***unsigned* long**: Liczba całkowita bez znaku w zakresie od `0` do `4 294 967 295`.

  - **float**: Liczba zmiennoprzecinkowa o pojedynczej precyzji. Przechowuje liczby zmiennoprzecinkowe z około 6-7 cyframi dziesiętnymi.
  - **double**: W Arduino na AVR to samo co float. Na innych platformach, jak ESP32, może oferować większą precyzję niż **float**.

- Złożone typy danych
  - *Tablice*: Umożliwiają przechowywanie wielu wartości tego samego typu w jednej zmiennej.
  - *Struktury* (**struct**): Pozwalają na grupowanie zmiennych różnych typów pod jedną nazwą.
  - *Unie* (**union**): Podobnie jak struktury, ale wszystkie pola zajmują ten sam obszar pamięci.
  - *Klasy* (**class**): Podstawowa jednostka programowania obiektowego w `C++`, umożliwiająca tworzenie złożonych typów danych z metodami i właściwościami.

- Typy specyficzne dla Arduino
  - **String**: Klasa umożliwiająca operacje na łańcuchach znaków.
  - **Stream**: Klasa bazowa dla wielu klas wejścia/wyjścia, np. `Serial`.
  - **Printable**: Interfejs umożliwiający klasom implementację metody `printTo(Print& p)` do obsługi wydruku.

Ze względu na ograniczoną dostępność pamięci zaleca się także stosowanie dostępnych w środowisko predefiniowanych typów danych opartych na typach podstawowych o ograniczonym rozmiarze. Typy te dostarczane są przez bibliotekę `<stdint.h>`:

- Typy całkowite ze znakiem
  - **int8_t**: 8-bitowa liczba całkowita ze znakiem.
    Zakres: od `-128` do `127`.
  - **int16_t**: 16-bitowa liczba całkowita ze znakiem.
    Zakres: od `-32 768` do `32 767`.
  - **int32_t**: 32-bitowa liczba całkowita ze znakiem.
    Zakres: od `-2 147 483 648` do `2 147 483 647`.
  - **int64_t**: 64-bitowa liczba całkowita ze znakiem.
    Zakres: od `-9 223 372 036 854 775 808` do `9 223 372 036 854 775 807`.

- Typy całkowite bez znaku
  - **uint8_t**: 8-bitowa liczba całkowita bez znaku.
    Zakres: od `0` do `255`.
  - **uint16_t**: 16-bitowa liczba całkowita bez znaku.
    Zakres: od `0` do `65 535`.
  - **uint32_t**: 32-bitowa liczba całkowita bez znaku.
    Zakres: od `0` do `4 294 967 295`.
  - **uint64_t**: 64-bitowa liczba całkowita bez znaku.
    Zakres: od `0` do `18 446 744 073 709 551 615`.

Procedura deklaracji typów złożonych i dynamicznych realizowana jest analogicznie jak w czystym kodzie `C\C++`. W przypadku klas zalecane jest przeniesienie ich kodu do zewnętrznych plików, nagłówkowego (*.h) i źródłowego (*.cpp). Poniżej zapisano kod klasy realizującej cykliczna zmianę stanu na wskazanym porcie mikrokontrolera w podanej sekwencji czasowej. Biblioteka `Arduino.h` jest podstawową biblioteką używaną w projektach Arduino i innych mikrokontrolerów. Zawiera definicje funkcji i

makra potrzebne do komunikacji z hardwarem mikrokontrolera, takie jak operacje wejścia/wyjścia na pinach, obsługa czasu (np. `delay()`), konfiguracja pinów itp. Dzięki niej można łatwo sterować peryferiami bez konieczności zagłębiania się w szczegóły mikrokontrolera.

Listing 2: Plik nagłówkowy klasy *.h

```
#ifndef Blinker_h
#define Blinker_h

#include "Arduino.h"

class Blinker
{
  public:
    Blinker(int pin);
    void blink(int delayTime);
  private:
    int _pin;
};
#endif
```

Listing 3: Plik źródłowy klasy *.cpp

```
#include "Blinker.h"
#include "Arduino.h"

Blinker::Blinker(int pin)
{
  pinMode(pin, OUTPUT);
  _pin = pin;
}

void Blinker::blink(int
   delayTime)
{
  digitalWrite(_pin, HIGH);
  delay(delayTime);
  digitalWrite(_pin, LOW);
  delay(delayTime);
}
```

W pisanym projekcie aplikacji wystarczy za pomocą dyrektywy `#include" "` podpiąć bibliotekę klasy, utworzyć w niej obiekt i można korzystac z jej właściwości, jak pokazano w przykładzie poniżej.

Listing 4: Kod programu z zaimportowaną klasą

```
#include "Blinker.h"

Blinker ledBlinker(13);

void setup() {
  // nic nie jest potrzebne w setup
}

void loop() {
  ledBlinker.blink(1000); // Miga co 1000ms (1 sekunda)
}
```

Należy zwrócić uwagę na funkcję `delay(czas)`, która wstrzymuje działanie aplikacji na `czas` podany w *ms*.

2.2 Dynamiczna alokacja pamięci

Dynamiczna alokacja tablic może być realizowana analogicznie analogicznie jak w `C` (za pomocą funkcji `malloc()`, `calloc()`, `realloc()` i `free()`) i `C++` (za pomocą operatorów `new` i `delete`). Należy pamiętać że w przypadku dynamicznej alokacji pamięci dla tablicy obiektowej zalecane jest korzystanie z operatora `new`. Poniżej zapisano kod aplikacji obiektowej z dynamiczną alokacją tablicy obiektowej zdefiniowanej w kodzie programu klasą.

Listing 5: Dynamiczna alokacja tablicy obiektowej

```
class klasa{
  public:
    klasa(){
```

```
4         Serial.println("Konstruktor klasy.");
5       }
6       ~klasa(){
7         Serial.println("Destruktor klasy.");
8       }
9       void toDo(int i){
10         Serial.print("Wywolanie metody dla obiektu: ");
11         Serial.println((unsigned long)this, HEX);
12      }
13  };
14  void setup(){
15    Serial.begin(9600);
16    klasa *obiekt;
17    int n = 5;
18    Serial.println("Utworzonie "+String(n)+ " obiektow.");
19    obiekt = new klasa[n];
20    for(int i=0;i<n;i++)
21       obiekt[i].toDo(i);
22    delete [] obiekt;
23  }
24  void loop(){}
```

3) Interfejs GPIO

Podstawowym interfejsem każdego mikrokontrolera są porty GPIO obsługujące komunikację urządzenia z elementami zewnętrznymi. Wyszczególnić tu można:

- porty cyfrowe
- port PWM
- Porty cyfrowo-analogowe i analogowo-cyfrowe
- porty szeregowe:
  - USB
  - UART
  - SPI
  - I^2C

Ich obsługa realizowana jest przez dedykowane narzędzia dostępne w środowisku.

3.1 Port cyfrowy

Obsługę każdego portu należny rozpatrywać w dwóch trybach pracy, jako portu wejściowego i wyjściowego. Wybór trybu realizowany jest na etapie inicjacji portu, zazwyczaj w funkcji `setup()`. Porty przyjmują wartości (`HIGH` lub `LOW`), co jest tożsame z wartościami logicznymi ((`TRUE`/`FALSE`) i binarnymi ((`1`/`0`).

```
pinMode(port, tryb);
```

Parametr `port` aktywuje port o podanym numerze, natomiast atrybut `tryb` ustawia wybrany tryb pracy:

- `INPUT` - odczytywanie stanu portu (`HIGH`/`LOW`)
- `OUTPUT` - ustawianie stanu portu (`HIGH`/`LOW`)
- `INPUT_PULLUP` - odczytywanie stanu portu z podciągnięciem stanu portu do napięcia `Vcc`

Odczytanie stanu portu realizowane jest za pomocą instrukcji `digitalRead(port)`. Funkcja zwraca odczytany stan portu ((`HIGH` lub `LOW`)). Natomiast zapisywanie funkcja:

```
digitalWrite(port,wartosc);
```

, gdzie atrybut `wartość` może przyjmować jedną z dwóch wartości logicznych (`HIGH`/`LOW`). Poniżej zapisano kod przykładowego programu operującego na portach cyfrowych.

Listing 6: Porty cyfrowe

```
1  void setup() {
2    pinMode(13, OUTPUT);
3    pinMode(2, INPUT);
4  }
5
6  void loop() {
7    if (digitalRead(2) == HIGH) {
8      digitalWrite(13, HIGH);
9    } else {
10     digitalWrite(13, LOW);
11   }
12 }
```

3.2 Port PWM

PWM (**P**ulse **W**idth **M**odulation, modulacja szerokości impulsów) to technika wykorzystywana w elektronice do sterowania mocą dostarczaną do urządzeń, takich jak silniki, serwomechanizmy, diody LED i inne. W platformach mikrokontrolerowych, porty **PWM** pozwalają na symulowanie analogowego sygnału wyjściowego za pomocą cyfrowej metody sterowania, która szybko przełącza stan pinu między `HIGH` a `LOW`, z różnym stosunkiem wypełnienia.

Stosunek czasu, przez który sygnał jest w stanie wysokim, do całkowitego czasu cyklu nazywany jest "stosunkiem wypełnienia"i jest wyrażony w procentach. Zmieniając stosunek wypełnienia impulsu, można kontrolować ilość energii dostarczanej do urządzenia. Na przykład, jeśli sygnał **PWM** ma stosunek wypełnienia 50%, średnia moc dostarczana do urządzenia będzie wynosić połowę maksymalnej mocy.

Porty cyfrowe zdolne do generowania sygnału **PWM** są zazwyczaj oznaczone na płytce symbolem . Aby sterować wyjściem **PWM**, używa się funkcji `analogWrite(port, wartość)`, gdzie pin to numer pinu obsługującego **PWM**, a `wartość` to wartość od 0 do 255, która odpowiada różnym stopniom wypełnienia impulsu (od 0% do 100%). Poniżej zapisano kod przykładu z zastosowaniem portu **PWM** do sterowania jasnością świecenia diody LED.

Listing 7: Port PWM

```
1  int ledPin = 9;
2
3  void setup() {
4    pinMode(ledPin, OUTPUT);
5  }
6
7  void loop() {
8    for (int i = 0; i <= 255; i++) {
```

```
9       analogWrite(ledPin, i);
10      delay(10);
11    }
12    for (int i = 255; i >= 0; i--) {
13      analogWrite(ledPin, i);
14      delay(10);
15    }
16  }
```

Przedstawiona powyżej procedura nie jest obsługiwana przez API M5Core. W przypadku tego mikrokontrolera i innych opartych na ESP32. Realizowane jest to przez bibliotekę `<ledc>`, dostarczająca wyspecjalizowanych funkcji do tego zadania.

- konfiguracja kanału PWM
  `ledcSetup(kanał,częstotliwość,rozdzielczość);`
  , gdzie:
  - kanał - w zależności od modelu mikrokontrolera z rodziny M5Stack od 1 do 16 kanałów
  - częstotliwość - od 1 HZ do około 40 MHz, zazwyczaj kilka kHz
  - rozdzielczość - precyzja sterowania stopniem wypełnienia PWM z zakresu 0 do 100%, w rozdzielczości określonej długością słowa bitowego od 1 do 16 bit (1 bit –> dwie wartości 0% i 100%, 8bit –> 256 wartości, 16bit –> 65536 wartości)
- konfiguracje portu PWM
  `ledcAttachPin(kanał,port);`
  podpięcie do `kanał` wskazanego `port` mikrokontrolera
- ustawienie wartości dla kanału PWM
  `ledcWrite(kanał,wartość);`

W oparciu o powyższe informacje można zmodyfikować kod z listingu 7 dla mikrokontrolera M5Core, jak pokazano poniżej.. Przy projektowaniu układu elektrycznego z mikrokontrolerem M5Core nalezy pamiętać, że jego porty pracują na zakresie napięciowym stanów logicznych `FALSE` –> 0V, `TRUE` –> 3.3V.

Listing 8: Port PWM dla M5Core

```
1  #include <M5Stack.h>
2  #include "driver/ledc.h"
3
4  #define kanal 0
5  #define rozdz 8
6  #define czest 5000
7  #define port 5
8
9  void setup() {
10   ledcSetup(kanal, czest, rozdz);
11   ledcAttachPin(port, kanal);
12 }
13
14 void loop() {
15   for (int i = 0; i <= 255; i++) {
16     ledcWrite(kanal, i);
17     delay(15);
18   }
19   for (int i = 255; i >= 0; i--) {
```

```
20      ledcWrite(kanal, i);
21      delay(15);
22    }
23  }
```

3.3 Port analogowy

Zintegrowany na płytce mikrokontrolera przetwornik ADC pozwala odczytanie skwantyfikowanej wartości napięcia z danego portu. Odczytanie wartości realizowane jest przez funkcję `analogRead(port);`. Dostępne narzędzia pozwalają na określenie rozdzielczości z jaką wartość będzie odczytywana, co jest realizowane za pomocą funkcji:

```
analogReadResolution(rozdzielczosc);
```

, gdzie `rozdzielczosc` w przypadku mikrokontrolera M5Core może przyjmować wartość od 9 do 12 bit.

9 –> 512

10 –> 1024

11 –> 2048

12 –> 4096

Wartość napięcia można obliczyć odnosząc się do napięcia odniesienia `Vcc=3.3V` dla maksymalnej wartości przy danej rozdzielczości: $U = \frac{Vcc \cdot VAL}{rozdzielczosc}$, gdzie `VAL` jest wartością odczytaną z portu analogowego funkcją `analogRead();`.

3.4 Port szeregowy

Mikrokontroler M5Core pozwala na korzystanie z trzech portów szeregowych UART, przy czym port UART0 jest domyślnie używany do komunikacji po porcie USB. Obsługa szeregowego portu USB jest wykorzystywana zazwyczaj na etapie projektowania aplikacji do komunikacji z mikrokontrolerem w celu kontroli poprawności działania kodu poprzez wymianę informacji. Otwarcie komunikacji pop porcie szeregowym USB realizowane jest przez instrukcję:

```
Serial.begin(szybkosc, param);
```

, gdzie:

- `szybkosc` określa szybkość transmisji (baud rate), M5Core pozwala na ustawienie jej w zakresie od około 300 baud'ów do ponad 4 Mbps, zazwyczaj ustawia się jedną ze wartości z standardowego ciągu prędkości transmisyjnych: 9600, 19200, 38400, 57600, 115200, 230400, 460800, 921600 bps.
- opcjonalny parametr określający parametry transmisji szeregowej liczba bitów danych, bitów stopu i parzystości(domyślnie `SERIAL_8N1`), pozostałe można znaleźć w dokumentacji języka na stronie WWW.

Inicjacja portu szeregowego umiejscawiana jest zazwyczaj w funkcji `setup()`, dzięki czemu w dalszej części kodu można realizować komunikację dwukierunkową z konsola szeregową w środowisku programowania Arduino IDE. Dane wysyłane z mikrokontrolera można w środowisku programowania wyświetlać na dwa sposoby, w konsoli w postaci ciągu tekstowego, oraz o przesyłane są dane liczbowe (`int`, `float`) w kreślarce w postaci wykresu danych w funkcji czasu.

Otwarty port szeregowy jest obiektem dostarczającym szeregu metod opisanych w API języka. Do podstawowych i najczęściej stosowanych metod należą:

- `Serial.begin();` - otwarcie portu szeregowego (domyślnego)
- `while(!Serial){}` - pętla wstrzymująca wykonanie kodu do czasu poprawnego otwarcia portu szeregowego, stosowana gdy użycie w aplikacji portu szeregowego jest warunkiem krytycznym
- `Serial.print();` i `Serial.println();` - wysłanie na port szeregowy ciągu tekstowego z lub bez znaku przejścia do nowego wiersza `"\n"`.
  Metoda przyjmuje dwa atrybuty, przy czym drugi jest opcjonalny.

  `Serial.print(dane,format);`

  `dane` mogą być ciagiem tekstowym, liczbą, wartością zwracaną przez funkcję, zmienną, itd. `format` pozwala na określenie systemu liczbowego za pomocą którego liczby całkowite będą przesłane (BIN,OCT, HEX, DEC) lub liczbę znaków po przecinku dla liczb rzeczywistych.
- `Serial.read();` odczytanie pojedynczego bajtu (znaku) z portu szeregowego
- `Serial.parseInt();` odczytanie znaków z portu szeregowego i konwersja, o ile to możliwe, do liczby całkowitej
- `Serial.parseFloat();` odczytanie znaków z portu szeregowego i konwersja, o ile to możliwe, do liczby rzeczywistej
- `Serial.parseString();` i `Serial.parseStringUntil(znak);` odczytanie znaków z portu szeregowego i konwersja do ciągu tekstowego

Za poniższym przykładzie pokazano kod programu demonstrujący wykorzystanie przytoczonych powyżej metod do wymiany danych na porcie szeregowym.

Listing 9: Komunikacja po porcie szeregowym USB

```
1  int x = 0;
2  void setup() {
3    Serial.begin(115200);
4    while(!Serial)
5    {}
6    Serial.println("Komunikacja na porcie szeregowym USB zostala
       nawiazana.");
7  }
8
9  void loop() {
10   Serial.print("Podaj wartosc (int) x:");
11   while(Serial.available()<=0){
12     delay(100);
13   }
14   x = Serial.parseInt();
15   Serial.println("Wprowadzono wartosc x=");
16   Serial.println(x,DEC);
17   Serial.print("(HEX)x=");
18   Serial.println(x,HEX);
19   if(x>0){
20     for(int i=0;i<=x;i++)
21       Serial.println(i,DEC);
22   }
23   delay(1000);
24 }
```

4) Przerwania

Przerwania są specjalną metodą programowania umożliwiając zakodowanie reakcji programu na zdarzenie zewnętrzne związane z zmianą stanu na porcie. Przerwania programowane w kodzie dla M5Core pozwalają na reakcję w wyniku jednej z zmian stanu portu:

- `RISING` - przerwanie jest generowane przy przejściu stanu z niskiego na wysoki.
- `FALLING` - przerwanie jest generowane przy przejściu stanu z wysokiego na niski.
- `CHANGE` - przerwanie jest generowane przy każdej zmianie stanu.
- `LOW` - przerwanie jest generowane, gdy pin jest w stanie niskim.
- `HIGH` (nie zawsze obsługiwane) - przerwanie jest generowane, gdy pin jest w stanie wysokim.

Procedura konfiguracyjna przerwania realizowana jest dla zdefiniowanego portu wejściowego z funkcją i wyborem typu przerwania. Ze względu na specyfikę istotne jest aby funkcja realizowana w wyniku uruchomienia przerwania była możliwie szybka. Dlatego zaleca się aby jej kod nie był zbyt rozbudowany, a sama funkcja zainicjowana z atrybutem `IRAM_ATTR`. Atrybut `IRAM_ATTR` w kodzie dla M5Core (i podobnych mikrokontrolerów) jest używany do oznaczania funkcji lub zmiennych, które powinny być umieszczone w wewnętrznej pamięci RAM instrukcji (IRAM). Dostęp do IRAM jest szybszy niż do innych rodzajów pamięci, co jest krytyczne dla obsługi przerwań, które wymagają szybkiego reagowania.

Składnię instrukcji inicjującej zapisano poniżej.

```
attachInterrupt(digitalPinToInterrupt(port), funkcja, tryb);
```

,gdzie:

- `digitalPinToInterrupt(port)` - funkcja wiążąca przerwanie z portem,
- `funkcja` - nazwa funkcji wywoływanej przez przerwanie,
- `tryb` - wybór stanu portu inicjującego przerwanie.

W pewnych sytuacjach konieczne może być deaktywacja przerwania, może być ona zrealizowana przez funkcję `detachInterrupt(digitalPinToInterrupt(port));`. Poniżej zapisano przykład kodu z przerwaniem reagującym na zmianę stanu portu z `HIGH` na `LOW` zmieniając stan wybranego portu na przeciwny.

Listing 10: Przykład kodu z przerwaniem

```
1  volatile byte stan = LOW;
2
3  void IRAM_ATTR zmiana() {
4    stan = !stan;
5  }
6
7  void setup() {
8    pinMode(13, OUTPUT);
9    pinMode(2, INPUT_PULLUP);
10   attachInterrupt(digitalPinToInterrupt(2), zmiana, FALLING);
11 }
12
13 void loop() {
14   digitalWrite(13, stan);
15   delay(500);
16 }
```

Modyfikator `volatile` jest stosowany dla zmiennych które mogą być zmieniane w dowolnym momencie, bez bezpośredniego działania kodu, np. przez przerwanie. Dlatego zalecane jest aby zmienne które będą podlegały zmianie w wyniku działanie przerwań były deklarowane z dodatkowym modyfikatorem `volatile`.

Specyficznym typem przerwania jest zdarzenie zegarowe, którego wyzwalaczem jest zegar systemowy. Przerwanie jest inicjowane z określonym interwałem czasowym.

W kodzie pisanym dla mikrokontrolera M5Core korzystanie z timera wymaga zaprogramowania poniższych elementów:

- podpięcie biblioteki `Arduino.h`
- zdefiniowanie wskaźnika na obiekt klasy timer
  `hw_timer_t * zegar = NULL;`
- utworzenie funkcji z modyfikatorem `IRAM_ATTR`
- w funkcji `setup()`:
  - inicjacja timera (0-3)
    `zegar = timerBegin(numer, dzielnik, true);`
    , gdzie
    `numer` - numer timera
    `dzielnik` - ustawienie taktowania zegara, dobierane do czasu wyzwolenia zegara i taktowania mikrokontrolera. M5Core (240 MHz). Dzielnik powinien być dobrany na częstotliwość będącą całkowitą częścią zakładanego interwału zegara. Dla timera 16bit dzielnik może przyjmować wartość w zakresie od 2 do 65536. Wartość dzielnik dobiera się na podstawie zależności:
    $dzielnik = \frac{CzstotliwozegaraCPU}{Porzdanaczstotliwozegara}$
    Zakładając że zegar będzie wykonywany co 1 s, można przyjąć oczekiwaną częstotliwość na 100 kHz, wtedy $dzielnik = \frac{240 \cdot 10^6}{100 \cdot 10^3} = 2400$
  - przypisanie funkcji do zegara
    `timerAttachInterrupt(zegar, &funkcja, true);`
  - określenie czasu wyzwolenia funkcji zegara, podawana w cyklach zegara, ustalonych przez dzielnik. Zakładając ustaloną częstotliwość zegara przy dzielniku 2400 na 100 kHz, oznacza że 1 s odpowiada 100 000 cyklom zegara. Jeżeli chcemy aby zdarzenie było wywoływane 5 razy w ciągu sekundy, to należy ustawić czas wyzwolenia zdarzenia zegarowego na 20 000.
    `timerAlarmWrite(zegar, czas, true);`
  - uruchomienie zegara
    `timerAlarmEnable(timer);`

Czas interwału zdarzenia zegarowego może zostać zmieniony w trakcie działania programu. Procedura jest trójstopniowa:

- zatrzymanie zegara
  `timerAlarmDisable(zegar);`
- ustawienie nowej wartości interwału
  `timerAlarmWrite(zegar, czas, true);`
- uruchomienie przerwania zegarowego
  `timerAlarmEnable(timer);`

5) M5 API

API języka programowania stosowanego w **Arduino IDE** przy pisaniu kodu aplikacji dla mikrokontrolera **M5Core** jest w znacznym stopniu zgodna z ta stosowaną dla mikrkokontrolerów z rodziny **Arduino**. Jest także szereg bibliotek, klas i funkcji które są odmienne i dostępne są dedykowane dla rodziny mikrokontrolerów **M5Stack**. Szczegółowych informacji należy szukać w dokumentacji dostępnej na stronie producenta, np. na stronie odnoszącej się do API języka. W ramach poszczególnych instrukcji do ćwiczeń będą wprowadzane kolejne elementy API. Oprogramowanie elementów zintegrowanych na płytce mikrokontrolera **M5Core** wymaga podpięcia biblioteki `<M5Stack.h>` w kodzie programu.

5.1 Wyświetlacz

Wyświetlacz LCD w mikrokontrolerze M5Core ma rozdzielczośc 320x240 kolorowych pikseli, pozycja lewego górnego rogu indeksowana jest wartościami (0,0), a prawego dolnego (319,239). Wyświetlanie elementów na LCD jest realizowane w formie warstwowej, nowy element umieszczany na ekranie przykrywa to co jest pod nim.

Użycie ekranu wymaga jego inicjalizacji, co jest realizowane poprzez obiekt `M5` za pomocą metody `M5.begin();`. Obiekt jest automatycznie inicjowany w momencie podpięcia biblioteki `<M5Stack.h`. Możliwe jest także uśpienie ekranu w celu minimalizowanie zużycia energii przez mikrokontroler i wybudzenia go gdy jest to potrzebne. Zadanie te realizowane są odpowiednio przez metody `M5.sleep();` i `M5.wakeup();`.

W oparciu o dostępne narzędzia, opisane na stronie API M5Core, możliwe jest między innymi:

- wyświetlanie tekstu
- rysowanie (grafika wektorowa: piksel, linia, prostokąt, koło, elipsę,...)
- rysowanie tekstu, liczb
- wyświetlanie bitmap

Poniżej pokazano kod przykładowego programu demonstrującego obsługę ekranu LCD mikrokontrolera **M5Core**.

Listing 11: Test ekranu LCD

```
1  #include <M5Stack.h>
2  void rysujRamkeA()
3  {
4    M5.Lcd.fillRect(5,110,155,90,GREEN);
5    M5.Lcd.drawRect(5,110,155,90,RED);
6  }
7  void rysujRamkeB()
8  {
9    M5.Lcd.fillRect(165,110,155,125,GREEN);
10   M5.Lcd.drawRect(165,110,155,125,RED);
11 }
12 void animacja(){
13   int r=10;
14   for(int i=0;i<=45;i++){
15     rysujRamkeB();
16     M5.Lcd.fillCircle(242,172,r+i,BLACK);
17     delay(80);
18   }
19   for(int i=45;i>=0;i--){
```

```
20      rysujRamkeB();
21      M5.Lcd.fillCircle(242,172,r+i,BLACK);
22      delay(80);
23    }
24    rysujRamkeB();
25  }
26  void showTekst()
27  {
28    float x = random(10000000);
29    x=x/1000000.0;
30    rysujRamkeA();
31    M5.Lcd.setTextColor(BLACK,GREEN);
32    M5.Lcd.setTextSize(2);
33    M5.Lcd.setCursor(20,115);
34    for(int i=0;i<5;i++){
35      int y = M5.Lcd.getCursorY();
36      M5.Lcd.setCursor(20,y);
37      M5.Lcd.println((1+random(10))*x,6);
38      delay(500);
39    }
40  }
41  void setup() {
42    M5.begin();
43    M5.Lcd.fillScreen(BLACK);
44    M5.Lcd.setTextColor(RED,BLACK);
45    M5.Lcd.drawString("Test ekranu LCD",10,20,4);
46    M5.Lcd.drawString("Okno tekstu.",10,80,2);
47    M5.Lcd.drawString("Okno animacji.",170,80,2);
48    randomSeed(analogRead(0));
49    rysujRamkeA();
50    rysujRamkeB();
51  }
52
53  void loop() {
54    showTekst();
55    animacja();
56  }
```

5.2 Przyciski

Mikrokontroler **M5Core** posiada trzy zintegrowane przyciski fizyczne. Biblioteka `<M5Stack.h>` dostarcza metod detekcji zdarzeń związanych z naciśnięciem i zwolnieniem przycisku. Poszczególne przyciski zdefiniowane są jako `BtnA`, `BtnB` i `BtnC`. W dokumentacji API M5Core opisane są szczegółowo poszczególne metody. Są to w większości funkcje logiczne zwracające wartość `TRUE` lub `FALSE` w zależności od wystąpienia wybranego typu zdarzenia od czasu poprzedniej weryfikacji zdarzeń na mikrokontrolerze za pomocą metody `M5.update()`; Dostępne funkcje testowe pozna podzielić na kategorie:

- naciśnięcie
- zwolnienie
- czas naciśnięcia
- czas od zwolnienia przycisku

- zwolnienia po przytrzymaniu

Poniżej zapisano kod demonstrujący detekcję zdarzeń związanych z przyciskami fizycznymi mikrokontrolera **M5Core**.

Listing 12: Test zdarzeń z przyciskami fizycznymi mikrokontrolera M5Core

```
#include <M5Stack.h>

void setup() {
  M5.begin();
  M5.Lcd.print("Przykladowy program");
}

void loop() {
  M5.update();

  if (M5.BtnA.wasPressed()) {
    M5.Lcd.fillScreen(BLACK);
    M5.Lcd.setCursor(0, 0);
    M5.Lcd.print("Nacisnieto A");
  }
  if (M5.BtnB.wasPressed()) {
    M5.Lcd.fillScreen(BLACK);
    M5.Lcd.setCursor(0, 0);
    M5.Lcd.print("Nacisnieto B");
  }
  if (M5.BtnC.wasPressed()) {
    M5.Lcd.fillScreen(BLACK);
    M5.Lcd.setCursor(0, 0);
    M5.Lcd.print("Nacisnieto C");
  }
  delay(100);
}
```

Przyciski podpięte są do portów cyfrowych mikrokontrolera **M5Core** i mogą być także oprogramowane w oparciu o podstawowe narzędzie omówione w rozdziale 3 instrukcji. Poszczególne przyciski są podpięte do portów:

- Przycisk **A**: do *GPIO39*
- Przycisk **B**: do *GPIO38*
- Przycisk **C**: do *GPIO37*

6) Zadania

W oparciu o materiały przytoczone w instrukcji, API **M5Core** i inne materiały dostępne w internecie napisać programy realizujące poniższe zadania. Zadania zrealizować w środowisku **Arduino IDE***.

*Pisane kody programów oprzeć na klasach, funkcjach użytkownika. W programach dla układów z elementami zewnętrznymi (joystick, enkoder, wyświetlacz segmentowy, klawiatura), ich obsługę oprogramować samodzielnie bez korzystania z zewnętrznych bibliotek dedykowanych dla tych elementów.

1. Opracować układ symulujący drogowy sygnalizator świetlny, wykorzystać diody (czerwoną, żółtą i zieloną).

Implementując w układzie z mikrokontrolerem diodę LED należny pamiętać o ograniczeniu prądowym portu mikrokontrolera (około 20 mA) oraz bezpiecznym napięciu diody (2.0V dla czerwonej diody LED, 2.1V dla zielonej, 2.2V dla żółtej). Konieczne jest zastosowanie rezystora ograniczającego napięcie w obwodzie portu $R = \frac{Vcc - U_d}{I_{ogr}}$, co po podstawieniu wartości liczbowych daje wartość Rezystancji rezystora około 65Ω.

2. Opracować klasę "DIODA"umożliwiającą sterowanie diodą LED na wskazanym porcie. Klasa dostarcza metod pozwalających na:

   - zapalenie diody LED
   - zgaszenie diody LED
   - ustawienie jasności świecenia diody LED

3. Opracować układ umożliwiający sterowanie przez mikrokontroler cykliczne zapalanie diody LED zgodnie z kodem podanym przez użytkownika za pośrednictwem konsoli szeregowej. Cykl zapisany jest w postaci ciągu wartości liczbowych (1-9) naprzemienne oznaczających czas w sekundach zapalenia i zgaszenia diody LED. Kod programu oprzeć na klasie opracowanej w punkcie 2.
   Przykładowo dla przekazanego ciągu tekstowego: **241537912**, cykl pracy diody wygląda zgodnie z poniższym opisem:

   - 2s - zapalona
   - 4s - zgaszona
   - 1s - zapalona
   - 5s - zgaszona
   - 3s - zapalona
   - 7s - zgaszona
   - 9s - zapalona
   - 1s - zgaszona
   - 2s - zapalona

   Po zakończeniu cyklu dioda zostaje automatycznie wygaszona, a program przechodzi w tryb oczekiwania na nowy zestaw danych.

4. Opracować układ w którym sterowanie wychyleniem serwomechanizmu realizowane jest za pomocą joysticka.

5. Wykorzystując system zdarzeń związanych z przyciskami mikrokontrolera (A,B,C) przy opracowaniu grę "Refleks" polegającej na jak najszybszym naciśnięciu przycisku po zapaleniu losowo wybranej diody LED. Poszczególne przyciski odpowiadają diodom (żółtej, czerwonej i zielonej). Program liczy czas reakcji i wyświetla go na ekranie. Opracować system punktacji, który po wykonaniu określonej serii prób wyświetli ocenę w skali od 0 do 10.

6. Zaprojektować klasę do obsługi wyświetlacza segmentowego. Klasa powinna zawierać:

   - konstruktor ustawiający liczbę segmentów i konfigurujący porty sterujące działaniem wyświetlacza
   - metodę wyświetlającą wybraną cyfrę na wskazanym segmencie wyświetlacza
   - metodę wyświetlającą dowolną liczbę całkowitą 2,3 lub 4 cyfrową
   - metodę wyświetlającą liczbę rzeczywistą

7. Opracować program timera odliczającego czas do zera.

- czas podawany jest przez użytkownika, poprzez ustawienie minut i sekund (XX:XX),
- interfejs oprzeć na przyciskach mikrokontrolera (można wykorzystać także enkoder i joystick),
- odliczanie oprzeć na przerwaniu zegarowym,
- zaimplementować w kodzie możliwość zatrzymania i wznowienia odliczania, zresetowania jego wartości (00:00), ponownego ustawienia czasu odliczenia,
- w trakcie odliczania wyświetlany jest aktualny czas do końca, dwukropek pomiędzy cyframi minut i sekund miga w cyklu sekundowym,
- cyfry minut i sekund wyświetlane są w postaci dwucyfrowej (2 jako 02),
- po zakończeniu odliczania wyświetlany zaprogramowana animacja,
- opracować interfejs graficzny zegara z wykorzystanie funkcji graficznych biblioteki `<M5Stack.h>`,

8. Zmodyfikować kod z projektu nr. 6, aby czas zegara wyświetlany był za pomocą wyświetlacza segmentowego w oparciu o zaprojektowaną w zadaniu 5 klasę. Do rozdzielenia liczby godzin od sekund zastosować znak kropki. Po odliczeniu czasu wartości 00.00 mrugają w cyklu sekundowym.

9. Napisać program kontroli dostępu z wykorzystaniem dostępnych na płytce demo-board wybranych elementów:

   - - klawiatura - do wprowadzenia wybranych cyfr lub znaków
   - - enkoder - symulacja zamka szyfrowego
   - - RFID - karta lub token do weryfikacji
   - - wybrany element mechaniczny (silnik DC, serwomechanizm lub silnik krokowy) - do symulacji otwierania zamka

   Interfejs komunikacji z użytkownikiem oprzeć na interfejsie szeregowym lub ekranie LCD mikrokontrolera

Rozliczeniem ćwiczenia projektowego są:

A) zaprezentowanie na zajęciach działających programów realizujących poszczególne zadania od 1 do 6

B) przygotowanie indywidualnego sprawozdania z zadania 7,8 lub 9 w postaci dokumentu pdf zawierającego omówienie algorytmu (schemat blokowy), kodu (omówienie najważniejszych funkcji, klas, metod) i wniosków z realizacji zadania. Do sprawozdania załączyć kod programu.