

.NET Micro Framework STM32F4 Discovery

Wojciech Duda

2016.4.21

Spis treści

1	Teoria	3
2	Instalacja	3
2.1	Narzędzia:	3
2.2	Konfiguracja	4
3	Przycisk	9
3.1	Klasa InterruptPort	9
3.1.1	Referencje	9
3.1.2	Konstruktor	9
3.1.3	Funkcje	9
3.2	Program	10
4	LED	10
4.1	Klasa OutputPort	10
4.1.1	Referencje	10
4.1.2	Konstruktor	11
4.1.3	Funkcje	11
4.2	Program	11
5	PWM	12
5.1	Klasa PWM	12
5.1.1	Referencje	12
5.1.2	Konstruktor	12
5.1.3	Atrybuty	12
5.1.4	Funkcje	12
5.2	Program	12
6	Zegar czasu rzeczywistego	13
6.1	Klasa DateTime	13
6.1.1	Atrybuty	13
6.2	Program	13
7	SPI-Akcelerometr	14
7.1	Klasa SPI	14
7.1.1	Referencje	14
7.1.2	Konstruktor	14
7.1.3	Funkcje	14
7.2	Klasa SPI.Configuration	15
7.2.1	Referencje	15
7.2.2	Konstruktor	15
7.3	Program	16

8	Timer	18
8.1	Klasa Timer	18
8.1.1	Konstruktor	18
8.2	Program	18

1 Teoria

Rdzeń CortexM4F wykorzystuje architekturę ARMv7M. Pod względem organizacji pamięci jest to architektura harwardzka, tzn. pamięć zawierająca kod programu (Flash) i pamięć danych (SRAM) są rozdzielone i dostęp do nich odbywa się poprzez osobne magistrale.



Rysunek 1: Opis urządzenia

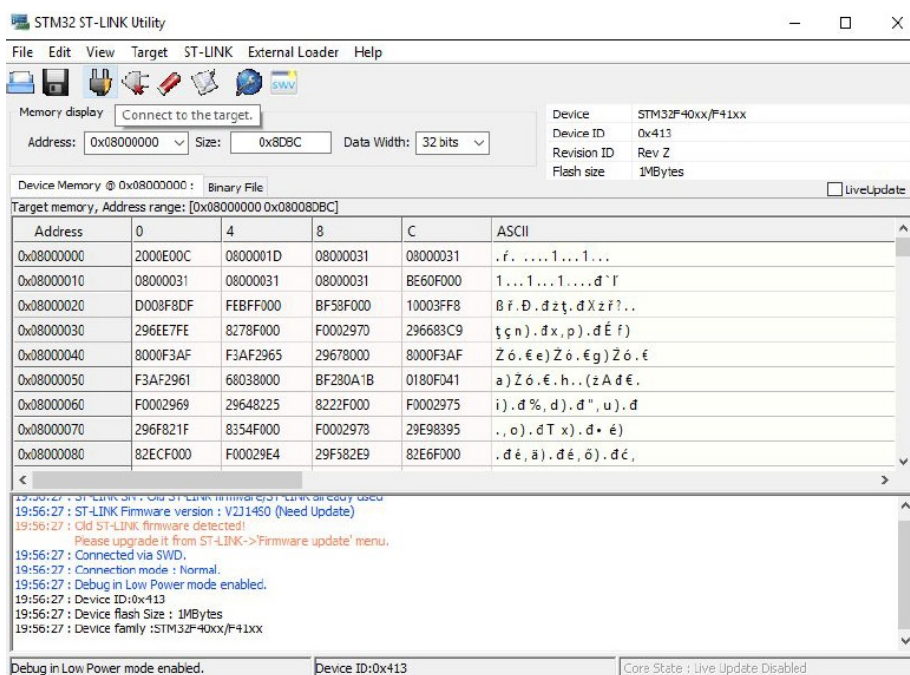
2 Instalacja

2.1 Narzędzia:

- mikrokontroler STM32F4 Discovery
- kable USB Micro oraz USB Mini
- Visual studio
- STM32 ST-LINK Utility (kliknij aby pobrać)
- sterownik USB (kliknij aby pobrać)
- bootloader oraz pliki hex (kliknij aby pobrać)
- .NET MicroFramework SDK (kliknij aby pobrać)

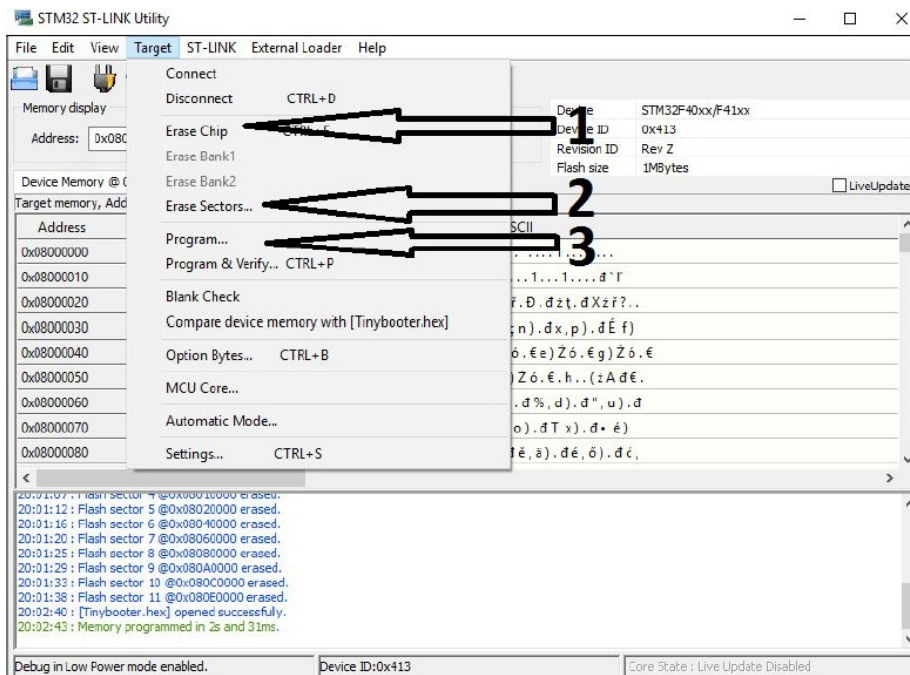
2.2 Konfiguracja

1. Pobierz pliki z punktu 2.1.
2. Zainstaluj STM32 ST-LINK Utility, oraz SDK, resztę plików rozpakuj.
3. Podłącz kabel USB Mini (do wejścia oznaczonego jako “Złącze USB” na Rysunku 1.)
4. Włącz STLINK Utility , a następnie połącz się z stm32f4 poprzez przycisk: “Connect to the = target”



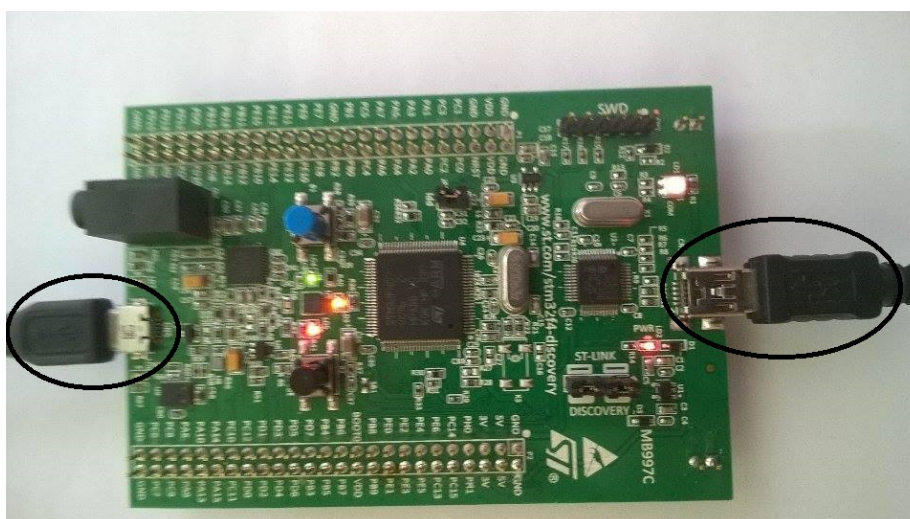
Rysunek 2: STLINK Utility

5. Następnie wybierz Target >Erase Chip oraz Target>Erase Sectors, wybierz wszystkie i potwierdź. Wybierz Target >Program... , wybierz ścieżkę Tinybooter.hex a następnie wybierz start. Zresetuj mikrokontroler poprzez przycisk zerujący.



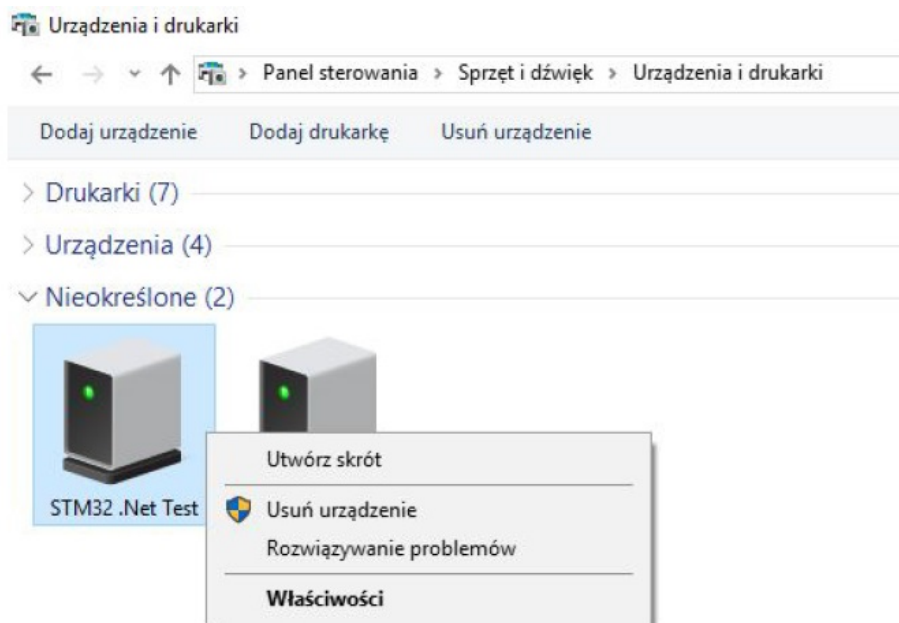
Rysunek 3: Programowanie debuggera

6. Jeżeli wszystko przebiegło prawidłowo powinny zapalić się 3 diody użytkowe. Podłącz kabel micro USB (jak na rysunku 4).



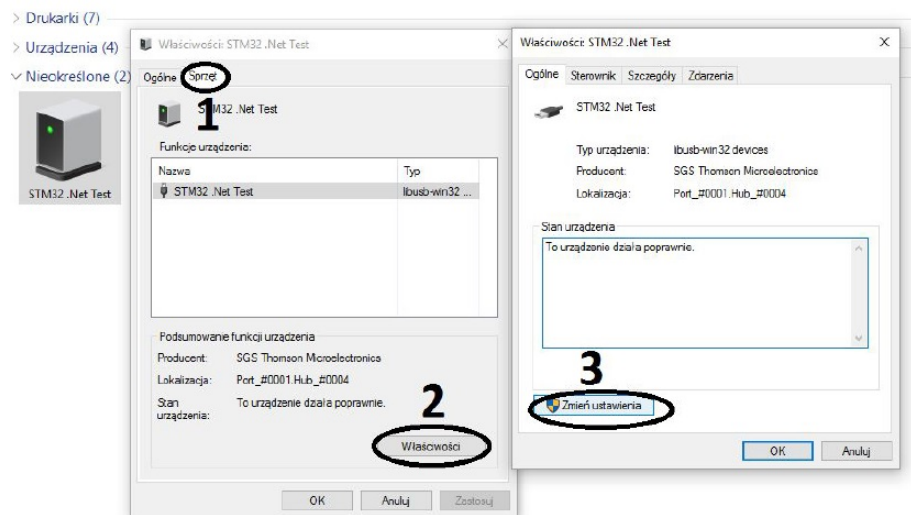
Rysunek 4: Podłączony STM32F4 kablami mikro i mini USB

7. Przejdź do “urządzenia i drukarki”. Tam w obszarze “nieokreślone” kliknij prawym przyciskiem myszy w “STM .Net Test” i wybierz właściwości.

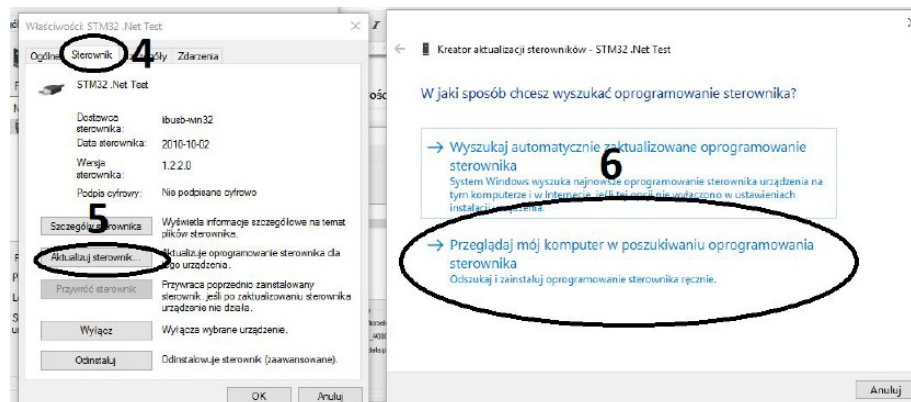


Rysunek 5: Urządzenia i drukarki

8. Wejdź w sprzęt >właściwości >zmień ustawienia >sterownik >Aktualizuj sterownik...

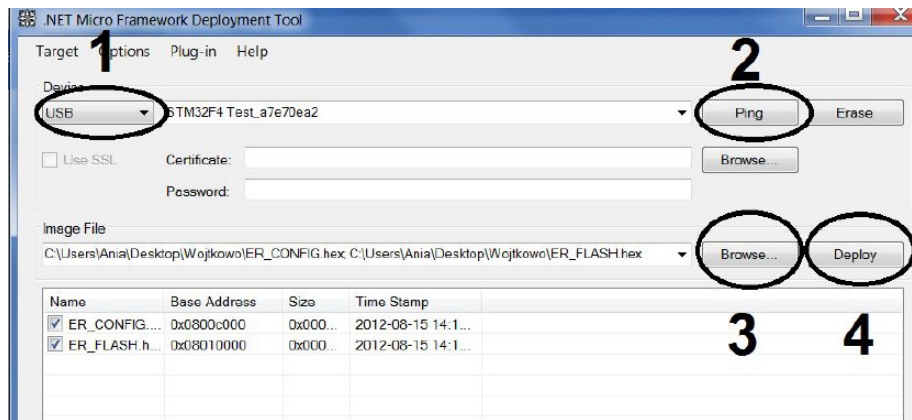


Rysunek 6: Instalacja sterownika krok 1



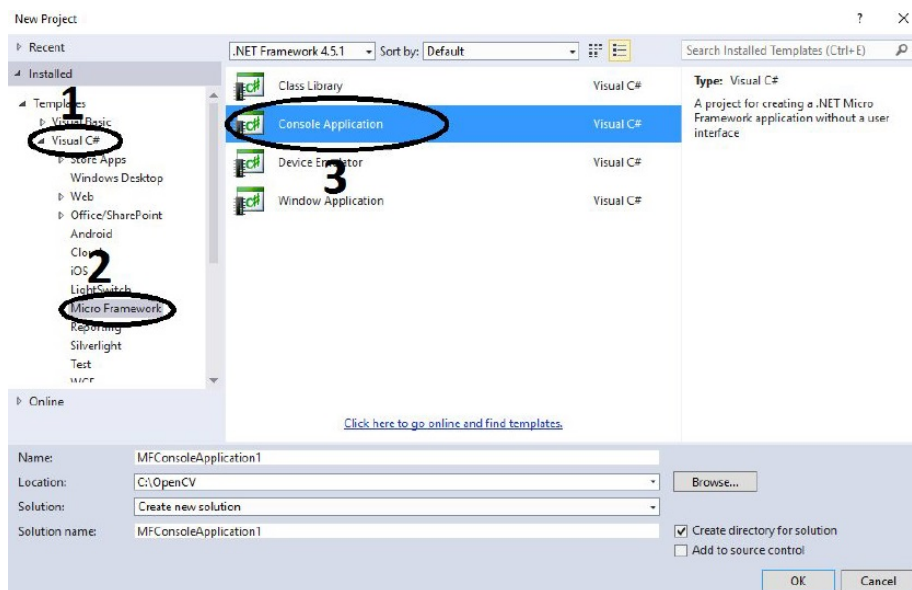
Rysunek 7: Instalacja sterownika krok 2

9. Wybierz "Przeglądaj mój komputer w poszukiwaniu oprogramowania sterownika" i wybierz ścieżkę gdzie rozpakowałeś na początku sterownik. Podczas instalacji ignoruj ostrzeżenia.
10. Uruchom MFDeploy. Wybierz Device: USB. Naciśnij przycisk Ping. Następnie drugie od góry Browse... , wybierz ścieżkę pozostałych dwóch plików hex: ER_CONFIG.hex, ER_FLASH.hex oraz wybierz Deploy.



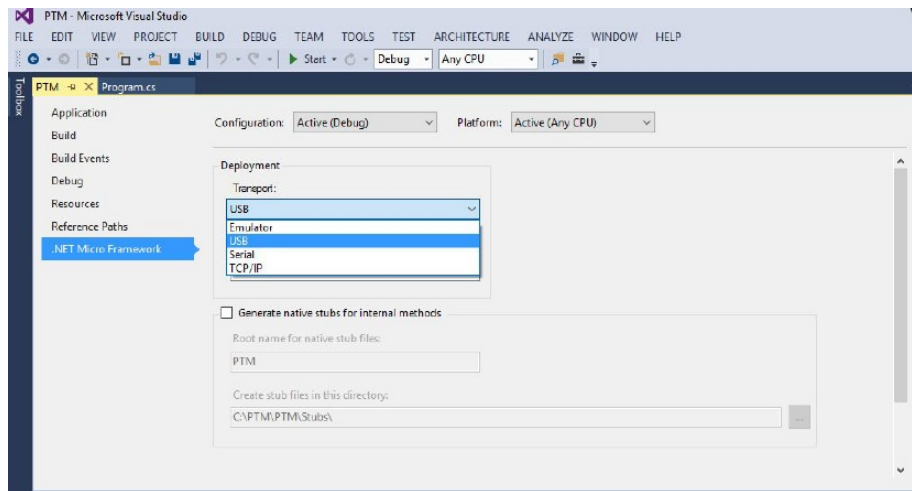
Rysunek 8: MF Deploy

11. Włącz Visual studio utwórz nowy projekt i wybierz C# >Micro Framework >Console Application.



Rysunek 9: Tworzenie projektu

12. W utworzonym projekcie, w Solution Explorer kliknij prawym przyciskiem myszy na projekt i wybierz "Properties". Tam wybierz .NET Micro Framework i Transport ustaw na USB.



Rysunek 10: Konfigurowanie Visual Studio

3 Przycisk

3.1 Klasa InterruptPort

Klasa zdefiniowana w przestrzeni nazw Microsoft.SPOT.Hardware.

3.1.1 Referencje

- Microsoft.SPOT.Hardware

3.1.2 Konstruktor

InterruptPort (Pin **portId**, bool **glitchFilter**, ResistorMode **resistor**, InterruptMode **interrupt**)

- **portId** - identyfikator portu.
- **glitchFilter**, - obsługa filtra błędów: true -włączony, false-wyłączony
- **resistor** - tryb rezystora, który określa stan domyślny dla portu.
- **interrupt** - tryb przerwania, który określa warunki wymagane do generowania przerwania.

3.1.3 Funkcje

bool Read () - zwraca aktualną wartość portu.

3.2 Program

Aby napisać program z użyciem przycisku trzeba najpierw utworzyć dla niego obiekt:

```
InterruptPort button = new InterruptPort( (Cpu.Pin)0, false, Port.ResistorMode.PullDown,
Port.InterruptMode.InterruptEdgeLevelHigh);
```

- `(Cpu.Pin)0` - Przycisk znajduje się na zerowym pinie portu A, każdy port ma 16 pinów. Port A jest pierwszym portem, więc $16 \cdot 0 + 0 = 0$.
- `false` - wyłączona obsługa filtru błędów
- `Port.ResistorMode.PullDown` - rezystor ustawiony na pulldown (Kiedy przycisk nie jest aktywny, zwracana jest wartość logiczna 0)
- `Port.InterruptMode.InterruptEdgeLevelHigh` - włącza przerwanie kiedy wartość portu jest wysoka.

Po utworzeniu obiektu dla przycisku można też zadeklarować inne obiekty potrzebne do programu (np. led opisane w punkcie 4). Po tych czynnościach można odczytać wartość portu przycisku (czy przycisk jest wduszony) za pomocą:

```
while(true)
if (button.Read() == true)
{kod programu(np. włączanie diód) }
else
{kod programu(np. wyłączenie diód) }
```

- `while(true)` - Pętla sterująca, dzięki niej jest ciągle sprawdzana wartość portu przycisku.
- `button.Read()` - Zwraca aktualną wartość portu przycisku.

4 LED

4.1 Klasa OutputPort

Klasa zdefiniowana w przestrzeni nazw Microsoft.SPOT.Hardware.

4.1.1 Referencje

- Microsoft.SPOT.Hardware

4.1.2 Konstruktor

OutputPort (Pin `portId`, bool `initialState`)

- `portId` - identyfikator portu.
- `initialState` - stan początkowy na porcie po aktywacji.

4.1.3 Funkcje

void Write(bool `state`) - wpisuje wartość do portu.

- `state` - wartość wpisywana do portu.

4.2 Program

Aby napisać program z użyciem diod trzeba dla każdej używanej diody stworzyć obiekt:

```
OutputPort led = new OutputPort( (Cpu.Pin)x, false)
```

- `(Cpu.Pin)x` - x- przyjmuje wartości od 60-63 diody znajdują się na końcowych pinach portu D, każdy port ma 16 pinów, więc $16*3+12=60$ oraz $16*3+15=63$. (niebieska-63, czerwona-62, pomarańczowa-61, zielona-60)
- `false` - stan początkowy diod - wyłączony

Po utworzeniu obiektów dla diod, można zająć się ich obsługą:

```
while(true)
{
    led.Write(true);
    for (int i = 0; i < 100000; i++)
    led.Write(false);
    for (int i = 0; i < 100000; i++)
}
```

- `while(true)` - Pętla sterująca, dzięki niej diody będą ciągle zmieniały swój stan.
- `led.Write(true);` - Włączenie diody.
- `led.Write(false);` - Wyłączenie diody.
- `for (int i = 0; i < 100000; i++)` - pętle opóźniające, dzięki nim diody wolniej zmieniają stan.

5 PWM

5.1 Klasa PWM

Klasa zdefiniowana w przestrzeni nazw Microsoft.SPOT.Hardware.

5.1.1 Referencje

- Microsoft.SPOT.Hardware.PWM
- Microsoft.SPOT.Hardware

5.1.2 Konstruktor

PWM (PWMChannel **channel**, Double **frequency_Hz**, Double **dutyCycle**, bool **invert**)

- **channel** - kanał PWM
- **frequency_Hz** - Częstotliwość impulsów w Hz.
- **dutyCycle** - Określa ile całkowitego czasu jest przeznaczanego na pracę jako wartość od 0.0 do 1.0(0-100%).
- **invert** - Wartość, która wskazuje, czy wyjście jest odwrócone.

5.1.3 Atrybuty

double DutyCycle - Pobiera lub ustawia cykl pracy impulsu jako wartość od 0.0 do 1.0.

5.1.4 Funkcje

void Start () - Uruchamia port PWM na nieokreślony czas.

5.2 Program

Aby napisać program obsługujący diody za pomocą PWM trzeba dla każdej używanej diody stworzyć obiekt:

```
var led = new PWM( Cpu.PWMChannel.PWM_x , 300, 0, false);
```

- **Cpu.PWMChannel.PWM_x** - x- przyjmuje wartości od 0 do 3. Oznaczając kanały PWM od 0 do 3(0-zielona, 1-pomarańczowa, 2-Czerwona, 3-Niebieska).
- **300** - Za niska częstotliwość może spowodować, że jasność diod nie zdąży się zmienić.

- 0 - przyjmuje 0% czasu cyklu pracy.
- false - wyjście ustawione jako nieodwrócone.

Oraz dla każdej trzeba wywołać funkcję Start:

led.Start(); Następnie trzeba zdefiniować kod programu(przykładowe zastosowanie):

```
while (true)
for (int i = 0; i < 10; i++)
{
led.DutyCycle = 1 - ((double)i / 10);
Thread.Sleep(1000);
}
```

- while (true) - Pętla sterująca, dzięki niej diody będą ciągle zmieniały swój stan.
- led.DutyCycle = 1 - ((double)i / 10); - zmiana mocy świecenia diody.
- Thread.Sleep(1000); - uśpienie wątku na sekundę(1000 milisekund). Dzięki czemu diody co sekundę zmieniają moc świecenia

6 Zegar czasu rzeczywistego

6.1 Klasa DateTime

Klasa zdefiniowana w przestrzeni nazw Microsoft.SPOT.

6.1.1 Atrybuty

- DateTime.Now.Second - zwraca sekundy z aktualnego czasu. Przyjmuje wartości od 0 do 59.
- DateTime.Now.Ticks - zwraca aktualną ilość przeskoków zegara.

6.2 Program

Przy samej obsłudze czasu nie trzeba tworzyć obiektów. Przykładowy kod w funkcji main:

```
while (true)
{
if (DateTime.Now.Second% 2==0)
{Kod programu}
}
```

- `while (true)` - Pętla sterująca, dzięki niej program się nie zakończy.
- `DateTime.Now.Second% 2==0` - Jeżeli sekundy z aktualnego czasu nie są nieparzyste, wykonaj Kod programu.

7 SPI-Akcelerometr

7.1 Klasa SPI

Klasa zdefiniowana w przestrzeni nazw Microsoft.SPOT.Hardware.

7.1.1 Referencje

- Microsoft.SPOT.Hardware

7.1.2 Konstruktor

SPI (`Config`)

- `Config` - Konfiguracja interfejsu SPI

7.1.3 Funkcje

-void Write (byte[] `writeBuffer`) - wpisuje blok danych do interfejsu.

- `writeBuffer` - buffor, który zostanie zapisany do interfejsu.

-void WriteRead (byte[] `writeBuffer`,ref byte[] `readBuffer`)

- `writeBuffer` - buffor, który zostanie zapisany do interfejsu.
- `readBuffer` - buffor do którego zostaną zapisane dane odczytane z interfejsu.

7.2 Klasa SPI.Configuration

Klasa zdefiniowana w przestrzeni nazw Microsoft.SPOT.Hardware.

7.2.1 Referencje

- Microsoft.SPOT.Hardware

7.2.2 Konstruktor

SPI.Configuration (Pin **ChipSelect_Port**, bool **ChipSelect_ActiveState**, UInt16 **ChipSelect_SetupTime**, UInt16 **ChipSelect_HoldTime**, bool **Clock_IdleState**, bool **Clock_Edge**, UInt16 **Clock_Rate**, SPI.module **SPI_mod**)

- **ChipSelect_Port** - Port wybranego czipu.
- **ChipSelect_ActiveState** - Stan aktywny dla portu wybranego czipu. Jeżeli prawda- port będzie ustawiany na wysoki w momencie dostępu do czipu, jeżeli fałsz- port będzie ustawiany na niski w momencie dostępu do czipu.
- **ChipSelect_SetupTime** - Czas pomiędzy wybraniem urządzenia a momentem kiedy zegar rozpocznie transakcję.
- **ChipSelect_HoldTime** - Określa, jak długo port czipu musi zostać w stanie aktywnym po zakończeniu transakcji czytania lub pisania.
- **Clock_IdleState** - Stan bezczynności zegara. Jeżeli prawda- sygnał zegara SPI zostanie ustawiony na wysoki, gdy urządzenie jest w stanie spoczynku. Jeżeli fałsz- sygnał zegara SPI zostanie ustawiony na niski, gdy urządzenie jest w stanie bezczynności.
- **Clock_Edge** - Jeżeli prawda- dane są próbkowane na zboczu wznoszącym zegara SPI. Jeżeli fałsz- dane są próbkowane na zboczu opadającym zegara SPI.
- **Clock_Rate** - Częstotliwość zegara SPI w kHz.
- **SPI_mod** - Magistrala SPI używana do transakcji.

7.3 Program

W klasie programu trzeba stworzyć obiekt:

```
static SPI MySPI = null;
```

Trzeba zdefiniować jedną metodę:

```
public static void WriteRegister( byte register, byte data)
{
    byte[] tx_data = new byte[2];
    tx_data[0] = (byte)(register — 0x00);
    tx_data[1] = data;
    MySPI.Write(tx_data);
}
```

- `byte register` - Adres rejestru.
- `byte data` - Wartość rejestru.
- `byte[] tx_data = new byte[2];` - Tablica, która będzie wpisana do SPI. W 0 elemencie przechowuje adres rejestru w 1 elemencie przechowuje wartość rejestru.
- `MySPI.Write(tx_data);` - wpisuje blok danych do akcelerometru.

Oraz jedną funkcję, zwracającą wartość z rejestru:

```
public static byte ReadRegister(byte register)
{
    byte[] tx_data = new byte[2];
    byte[] rx_data = new byte[2];
    tx_data[0] = (byte)(register — 0x80);
    tx_data[1] = 0;
    MySPI.WriteRead(tx_data, rx_data);
    return rx_data[1];
}
```

- `byte register` - Adres rejestru.
- `byte[] tx_data = new byte[2];` - Tablica, która będzie wpisana do SPI. W 0 elemencie przechowuje adres rejestru w 1 elemencie przechowuje wartość rejestru.
- `byte[] rx_data = new byte[2];` - Tablica, która będzie przechowywać wartości odczytane z rejestru.
- `MySPI.WriteRead(tx_data, rx_data);` - wpisuje i odczytuje bloki danych z akcelerometru.

W funkcji Main trzeba stworzyć obiekt SPI.Configuration:

```
SPI.Configuration MyConfig = new SPI.Configuration( (Cpu.Pin)67, false, 0,  
0, true, true, 1000, SPI.SPI_module.SPI1)
```

- (Cpu.Pin)67 - SPI znajduje się na trzecim pinie portu E, czyli $16 \cdot 4 + 3 = 67$.
- false - Port będzie ustawiany na niski w momencie dostępu do czipu.
- 0 - Natychmiastowe rozpoczęcie transakcji w momencie wybrania urządzenia
- 0 - Brak stanu aktywności po zakończeniu transakcji czytania lub pisania.
- true - Sygnał zegara SPI zostanie ustawiony na wysoki, gdy urządzenie jest w stanie spoczynku.
- true - Dane są próbkowane na zboczu wznoszącym zegara SPI.
- 1000 - Częstotliwość zegara SPI jest równa 1000 kHz
- SPI.SPI_module.SPI1 - Magistrala SPI 1.

Trzeba zdefiniować globalny obiekt MySPI:

```
MySPI = new SPI(MyConfig);
```

Uaktywnić akcelerometr:

```
WriteRegister( 0x20, 0xC7)
```

- 0x20 - W kodzie binarnym jest równe 0010 0000, pierwsze 00 oznacza tryb pracy zapisu, a reszta jest adresem rejestru.
- 0xC7 - W kodzie binarnym jest równe 11000111:
 - 1 - szybkość danych wyjściowych 400Hz(zero oznacza -100Hz)
 - 1 - ustawienie urządzenia w trybie aktywnym
 - 0 - wartości muszą być zero aby określone były zakresy X,Y,Z.
 - 00 - normlany tryb.
 - 111 - oznacza włączenie kolejno Z,Y,X.

Aby odczytać wartości akcelerometru, trzeba posłużyć się funkcją ReadRegister(X), gdzie X może przyjąć jedną z wartości:

- 0x2D - Rejestr z wartością Z.
- 0x29 - Rejestr z wartością X.
- 0x2B - Rejestr z wartością Y.

8 Timer

8.1 Klasa Timer

Klasa zdefiniowana w przestrzeni nazw System.Threading.

8.1.1 Konstruktor

Timer(TimerCallback **callback**, object **state**, uint **dueTime**, uint **period**)

- **callback** - nazwa metody, która ma być wykonywana.
- **state** - obiekt z informacjami wykorzystywanych w metodzie callback lub null.
- **dueTime** - opóźnienie, z jakim będzie wywoływać się metoda callback, podane w milisekundach.
- **period** - czas między wywołaniami metody callback, podany w milisekundach.

8.2 Program

Aby uzyskać timer trzeba zadeklarować specjalną metodę, która będzie wywoływana:

```
public static void nazwa(object state) { Kod programu }
```

- **nazwa** - zdefiniowana przez programistę nazwa wywoływanej metody przez Timer.
- **object state** - dodatkowy obiekt z informacjami wykorzystywanymi w metodzie.

Pozostało utworzyć obiekt timer:

```
Timer timer = new System.Threading.Timer( funTimer, null, 0, 1000);
```

- **funTimer** - Metoda, która ma być wykonywana.
- **null** - Brak obiektu z informacjami wykorzystywanymi w metodzie FunTimer.
- **0** - Brak opóźnienia wywołania metody FunTimer.
- **1000** - Czas, co ile będzie wywoływać się metoda Funtimer(1 sekunda)