

.NET Micro Framework STM32F4 Discovery

Wojciech Duda

2016.4.21

Spis treści

1	Teoria	2
2	Instalacja	2
2.1	Narzędzia:	2
2.2	Konfiguracja	3
3	Przycisk	8
3.1	Klasa InterruptPort	8
3.1.1	Referencje	8
3.1.2	Konstruktor	8
3.1.3	Funkcje	8
3.2	Deklaracja	9
4	LED	9
4.1	Klasa OutputPort	9
4.1.1	Referencje	9
4.1.2	Konstruktor	9
4.1.3	Funkcje	9
4.2	Deklaracja	9
5	PWM	10
5.1	Klasa PWM	10
5.1.1	Referencje	10
5.1.2	Konstruktor	10
5.1.3	Atrybuty	10
5.1.4	Funkcje	10
5.2	Deklaracja	10
6	Zegar czasu rzeczywistego	11
6.1	Klasa DateTime	11
6.1.1	Atrybuty	11
7	SPI-Akcelerometr	11
7.1	Klasa SPI	11
7.1.1	Referencje	11
7.1.2	Konstruktor	11
7.1.3	Funkcje	11
7.2	Klasa SPI.Configuration	12
7.2.1	Referencje	12
7.2.2	Konstruktor	12
7.3	Deklaracja	12
7.4	Timer	13
7.5	Klasa Timer	13
7.5.1	Konstruktor	13
7.6	Deklaracja	14

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

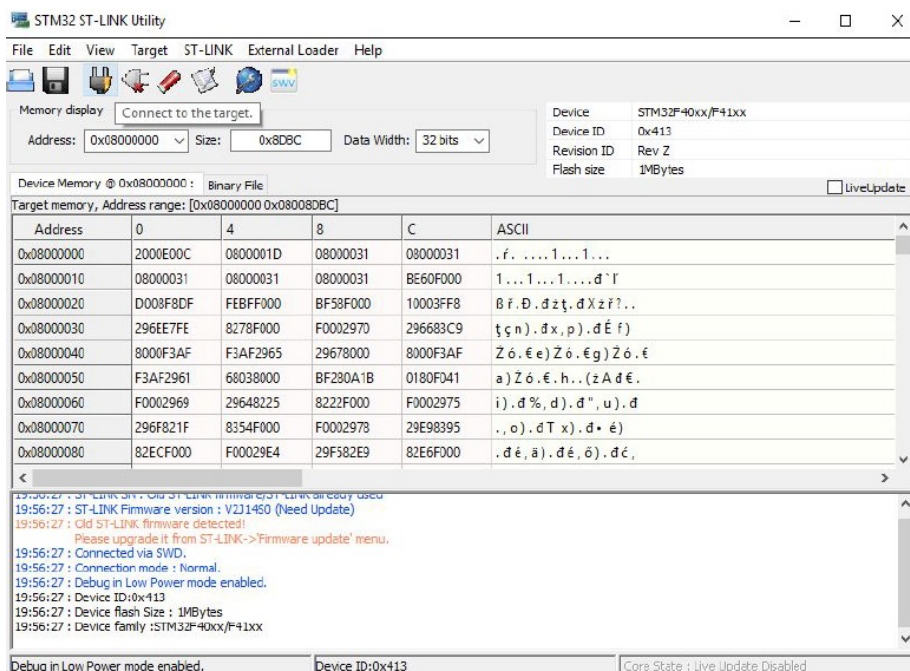
2 Instalacja

2.1 Narzędzia:

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

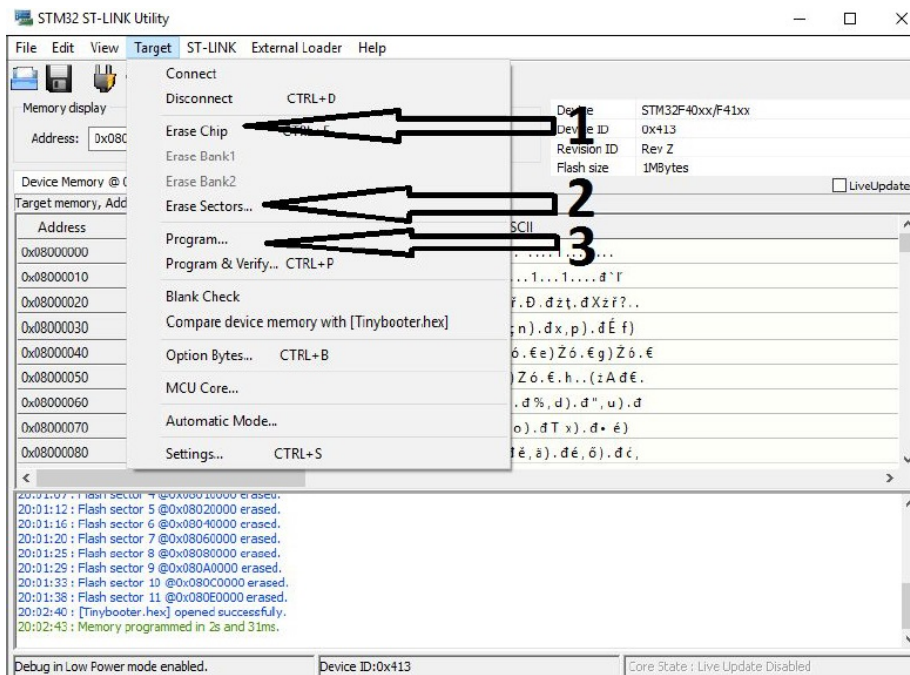
2.2 Konfiguracja

1. Zainstaluj STLINK, oraz SDK, resztę plików rozpakuj.
2. Podłącz kabel USB Mini (do wejścia oznaczonego jako “Złącze USB” na zdjęciu powyżej.)
3. Włącz STLINK Utility , a następnie połącz się z stm32f4 poprzez przycisk: “Connect to the = target”



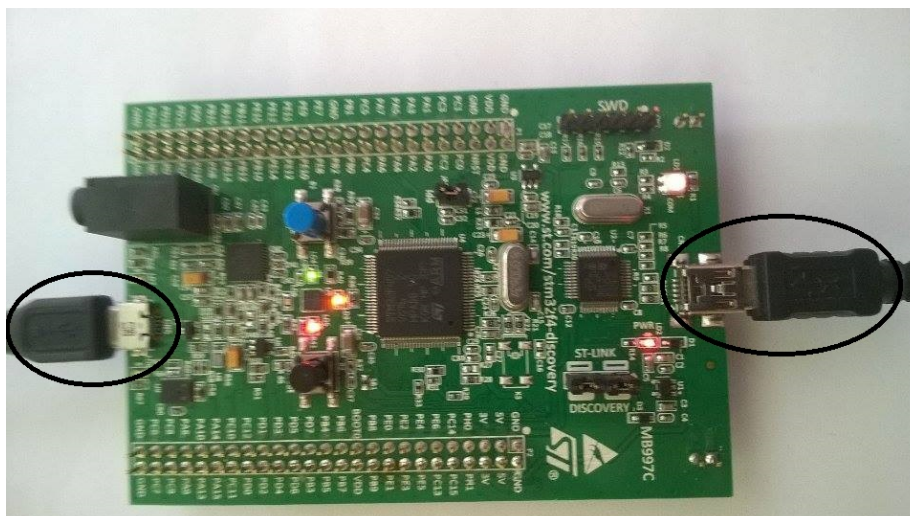
Rysunek 2: STLINK Utility

4. 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.



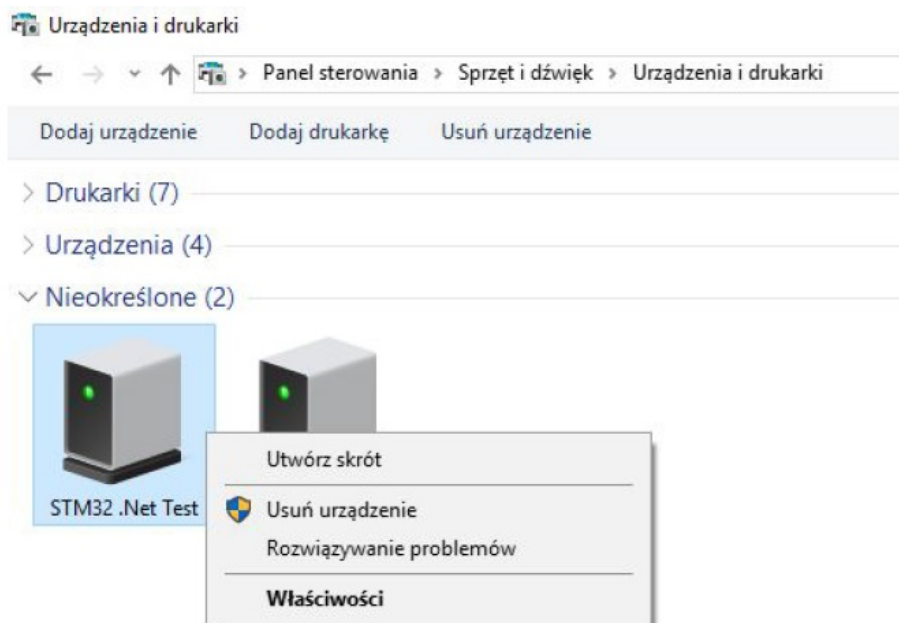
Rysunek 3: Programowanie debuggera

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



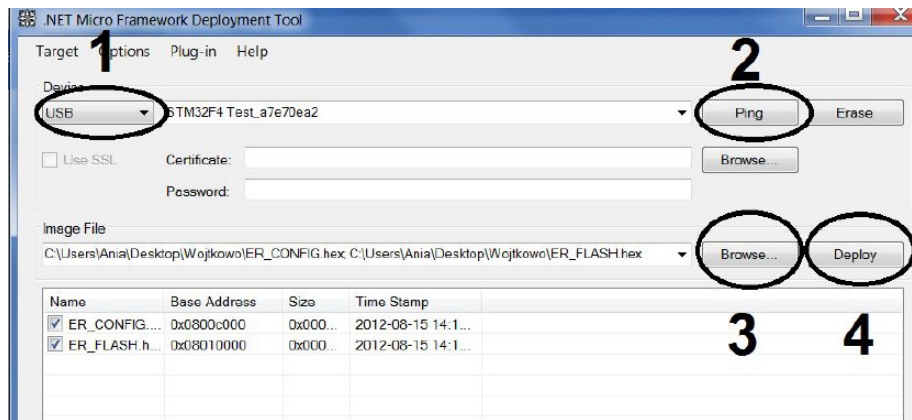
Rysunek 4: Podłączony STM32F4 kablami mikro i mini USB

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



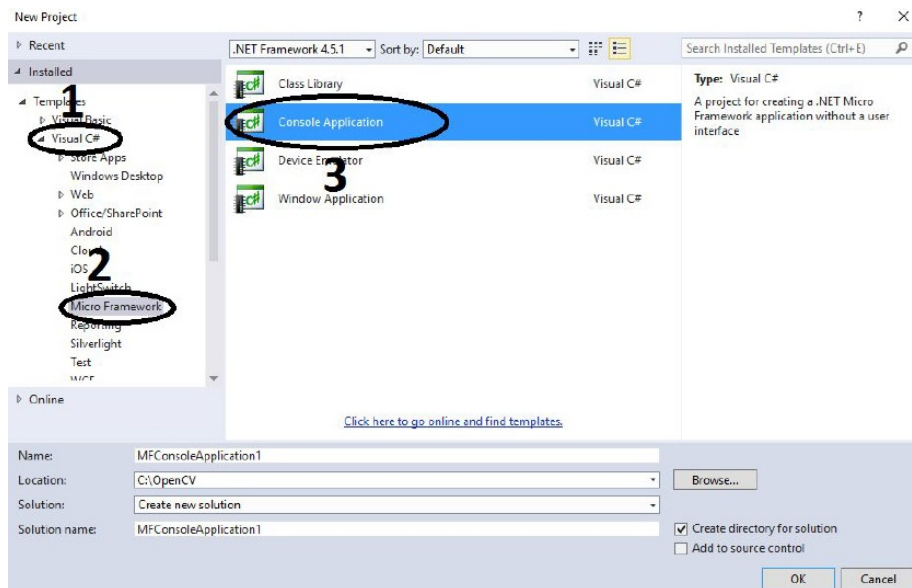
Rysunek 5: Urządzenia i drukarki

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



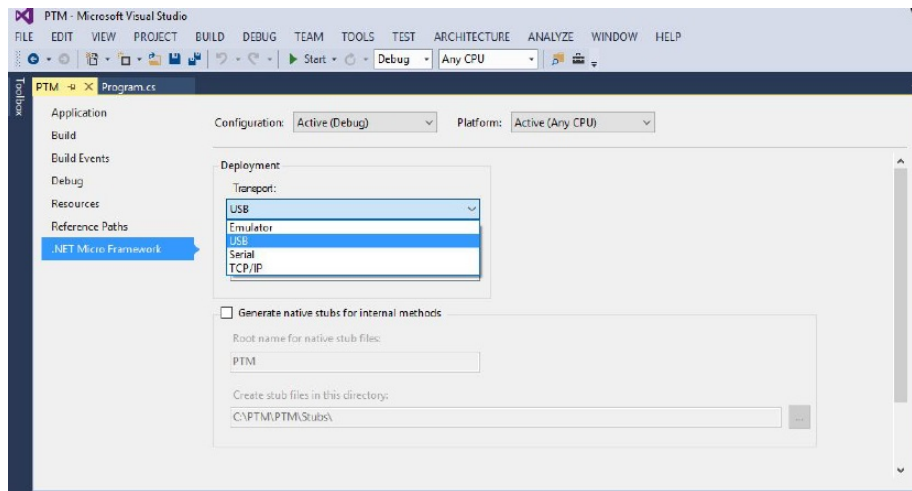
Rysunek 8: MF Deploy

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



Rysunek 9: Tworzenie projektu

11. 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 Deklaracja

`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 $16*0+0=0$.
- `false` - wyłączona obsługa filtru błędów
- `Port.ResistorMode.PullDown` - tryb pracy rezystora ustawiający na pulldown
- `Port.InterruptMode.InterruptEdgeLevelHigh` - włącza przerwanie kiedy wartość portu jest wysoka.

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 Deklaracja

`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 $16*3+15=63$.
- `false` - stan początkowy diod - wyłączony

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` - Cykl pracy impulsów jako ułamek jedności.
- `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 Deklaracja

```
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.
- `300` - Za niska częstotliwość może spowodować że jasność diod może nie zdążyć się zmienić.
- `0` - przyjmuje 100% czasu cyklu pracy.
- `false` - wyjście ustawione jako nie odwrócone.

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.

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 block 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. Jeśli prawda, sygnał zegara SPI zostanie ustawiony na wysoki, gdy urządzenie jest w stanie spoczynku. Jeśli fałsz, sygnał zegara SPI zostanie ustawiony na niski, gdy urządzenie jest w stanie bezczynności.
- **Clock_Edge** - Jeśli prawda, dane są próbkowane na zboczu wznoszącym zegara SPI. Jeśli fałsz, dane są próbkowane na zboczu opadającym zegara SPI.
- **Clock_Rate** - Częstotliwość zegara SPI w Hz.
- **SPI_mod** - Magistrala SPI używana do transakcji.

7.3 Deklaracja

-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 Hz
- **SPI.SPI_module.SPI1** - Magistrala SPI 1.

-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, 1100 oznacza szybkość danych wyjściowych równe 12.5 Hz, 0 oznacza że rejestr nie będzie się zmieniał dopóki nie przeczyta MSB oraz LSB, 111 oznacza włączenie X,Y,Z.

-(ReadRegister(**0x2D**))

- **0x2D** - Rejestr z wartością X.

-(ReadRegister(**0x29**))

- **0x29** - Rejestr z wartością z.

-(ReadRegister(**0x2B**))

- **0x2B** - Rejestr z wartością Y.

7.4 Timer

7.5 Klasa Timer

Klasa zdefiniowana w przestrzeni nazw System.Threading.

7.5.1 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.

7.6 Deklaracja

```
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)