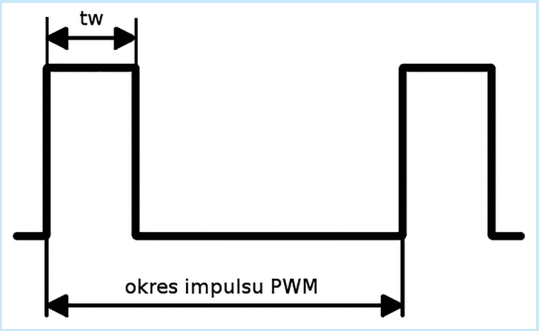
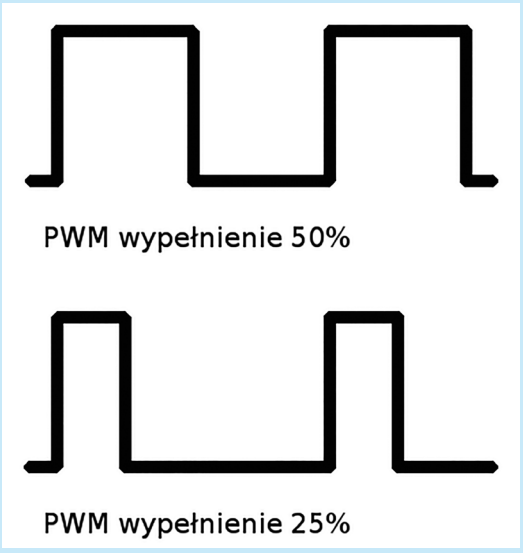
STM-owa układanka: generator PWM

Tematem artykułu jest odpowiedź na pytanie: jak zmusić kontroler STM32F411 do wygenerowania przebiegu PWM o pożądanych parametrach? Mam zamiar udowodnić, że za pomocą narzędzi wspomagających pisanie oprogramowania dla STM32, rozwiązanie problemu wcale nie musi być trudne.

Zgodnie z definicją PWM (*Pulse-Width Modulation*), jest metodą regulacji przebiegu o stałej amplitudzie i częstotliwości polegającą na zmianie wypełnienia. Modulacja PWM jest używana np. do sterowania silnikami elektrycznymi. Jest to też łatwy sposób zmiany jasności świecenia diod LED. Na **rysunku 1** pokazano typowy przebieg PWM. Zgodnie z podaną wcześniej definicją jego okres jest stały, a zmianie podlega czas *tw* oznaczający czas trwania impulsu. Ze wzoru *wypełnienie = (tw/okres impulsu PWM)×100%* można obliczyć wypełnienie przebiegu PWM podawane w procentach. Dla zilustrowania opisu, na **rysunku 2** pokazano dwa przebiegi PWM o wypełnieniach 50% i 25%. W ekstremalnych przypadkach dla wypełnienia równego 100%, wyjście generatora PWM przyjmie poziom wysoki, natomiast dla wypełnienia równego 0% będzie wyzerowane.



Rysunek 1. Typowy przebieg PWM



Rysunek 2. Dwa przebiegi PWM – o wypełnieniu 50% i 25%

Te informacje są prawdziwe dla założenia, że poziomem aktywnym jest poziom wysoki.

Sterowanie diod LED przebiegiem PWM

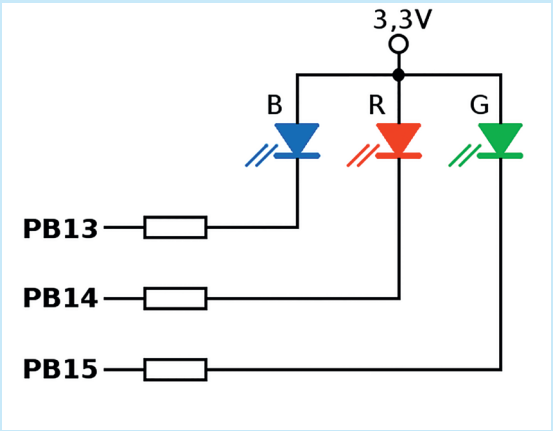
W typowym urządzeniu z mikrokontrolerem, do sterowania diodami LED wykorzystuje się wyjścia cyfrowe. Ponieważ taka linia może przyjąć jeden z dwóch stanów, można jej użyć do zaświecenia lub zgaszenia diody LED, ale nie da się zmienić jasności jej świecenia.

Jeżeli dioda LED będzie okresowo włączana z odpowiednio dużą częstotliwością, oko człowieka nie zarejestruje migotania, natomiast zmiana czasu trwania włączenia, czyli wypełnienie przebiegu PWM, będzie wpływała na subiektywnie odczuwaną jasność świecenia diody. W ten sposób w systemie cyfrowym można w prosty sposób uzyskać efekt sterowania analogowego.

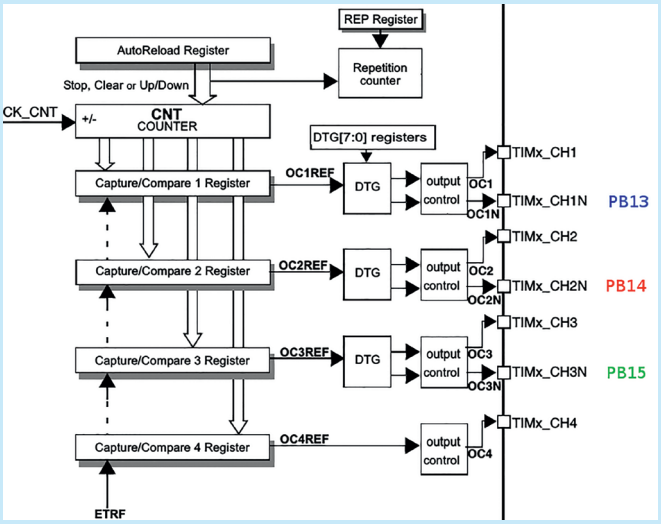
Użyta do eksperymentów płytka ewaluacyjna KA-NUCLEO-F411CE ma zamontowane dwie diody LED, pojedynczą i potrójną, którymi można sterować za pomocą aplikacji. W tym przykładzie zostanie użyta 3-kolorowa dioda LED D2. Sposób jej połączenia z portami kontrolera STM32F411CE pokazano na **rysunku 3**. Wybrane przez konstruktorów płytki porty kontrolera, nie są zupełnie przypadkowe, ponieważ są to wyprowadzenia kontrolowane przez Timer1, sprzętowy zegar STM32F411, a jedną z funkcji timerów może być generowanie przebiegów PWM.

Timer1 i generowanie przebiegu PWM

Układy czasowo – licznikowe (timery) stanowią blok funkcjonalny mikrokontrolera. Ich działanie jest



Rysunek 3. Dołączenie diody LED RGB do mikrokontrolera



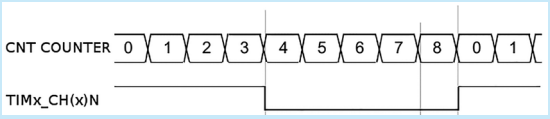
Rysunek 4. Uproszczona budowa układu czasowo-licznikowego Timer1

nadzorowane przez oprogramowanie. Używane są jako wyspecjalizowane struktury do zadań wiążących się z odmierzaniem czasu: generowanie precyzyjnie odmierzonych impulsów lub przebiegów o ustalonej częstotliwości, zliczania impulsów, pomiary czasu trwania zdarzeń. Mogą być także użyte do generowania przebiegów PWM o wypełnieniu stałym lub zmiennym. Uniwersalność timerów powoduje, że stają się coraz bardziej rozbudowanymi i skomplikowanymi układami. Na **rysunku 4** pokazano uproszczoną budowę Timera1. Pominięto na nim szczegóły niezwiązane z generowaniem przebiegu PWM.

Głównym elementem Timera1 jest licznik *CNT COUNTER*, do którego doprowadzone są zewnętrzne impulsy *CK_CNT*. Z licznikiem współpracują 4 rejestry porównań *Compare(x) Register*. Gdy zawartość licznika *CNT COUNTER* będzie równa zawartości rejestru *Compare(x)*, może wystąpić reakcja na ten fakt, np. zmiana poziomu określonego wyprowadzenia mikrokontrolera. Do każdego rejestru *Compare(x)* przyporządkowano określone wyprowadzenie mikrokontrolera oznaczone na rysunku symbolem *TIMx_CH*. Jeśli zawartość licznika *CNT COUNTER* zrówna się z zawartością rejestru, poziom logiczny wyprowadzenia może zmienić się albo na przeciwny, albo na wcześniej zaprogramowany. Rejestry *Compare(1-3)* wpływają na stan 2 portów: głównego *TIMx_CH(x)* i dodatkowego *TIMx_CH(x)N*. Przebieg na dodatkowym wyprowadzeniu może być taki sam, jak na głównym bądź w fazie przeciwnej, z zaprogramowanym przesunięciem fazowym. Jest to dodatkowe ułatwienie przy sterowaniu uzwojeniami silników bezszczotkowych.

Katody trójkolorowej diody D2 połączono do wyprowadzeń dodatkowych portów sterowanych przez rejestry *Compare(1-3)*. I tak:

– Katoda diody niebieskiej łączy się z portem PB13 pełniącym rolę wyprowadzenia *TIM1_CH1N* rejestru *Compare1*.
– Katoda diody czerwonej łączy się z portem PB14 pełniącym rolę wyprowadzenia *TIM1_CH2N* rejestru *Compare2*.
– Katoda diody zielonej łączy się z portem PB15 pełniącym rolę wyprowadzenia *TIM1_CH3N* rejestru *Compare3*.



Rysunek 5. Zależność wypełnienia od okresu zliczania licznika

Wykorzystując Timer1 do generowania przebiegu PWM, jego okres zależy od częstotliwości CK_CNT taktującej licznik *CNT COUNTER* oraz od okresu zliczania samego licznika. Wypełnienie zależy od ustawienia rejestru *Compare*. Tę zależność pokazano **rysunku 5**. *CNT COUNTER* pracuje z okresem zliczania równym 8+1, w rejestrze porównań *Compare* jest wpisana liczba 4. Zostanie dzięki temu wygenerowany przebieg o wypełnieniu (4/(8+1))*100=44,4% (dodawana w obliczeniach 1, to oznaczony jako zerowy pierwszy takt licznika).

Tworzenie szkieletu oprogramowania

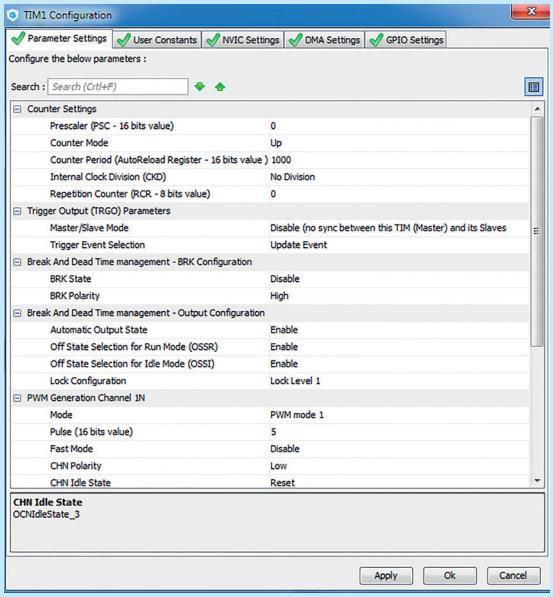
Teraz nadszedł czas, żeby zmusić Timer1 sterujący świeceniem podłączonych diod RGB, do wygenerowania przebiegu PWM. Do stworzenia szkieletu oprogramowania posłużymy się jednym z „układankowych" narzędzi: STM32CubeMX. W poprzednim artykule opisywałem dokładnie jak stworzyć nowy projekt. Po otwarciu zakładki *Pinout* wybieramy *TIM1*. Następnie zmieniamy następujące pozycje:

– *Clock Source–>Internal Clock.*
– *Channel1–>PWM Generation CH1N.*
– *Channel2–>PWM Generation CH2N.*
– *Channel3–>PWM Generation CH3N.*

Następnie przechodzimy do zakładki *Clock Configuration*. Jako źródło taktowania kontrolera (SYSCLK) wybieramy pętlę fazową PLL. Przy częstotliwości rezonatora kwarcowego 8 MHz i ustawieniu kolejnych podzielników: M=8, N=200, P=2, częstotliwość SYSCLK wyniesie 100 MHz, czyli maksymalnie tyle, ile dla STM32F411. Przechodzimy do zakładki *Configuration* i klikamy na *TIM1*. Otworzy się zakładka *TIM1 Configuration → Parameter Settings*. To dzięki tym ustawieniom „zmusimy" Timer1 do generowania przebiegu PWM o żądanych parametrach.

Okres generowanego przebiegu PWM zależy od częstotliwości taktowania Timera1 i od zakresu zliczanych impulsów. Częstotliwość impulsów zliczanych przez timer wynosi 100 MHz, czyli okres równy impulsów zegarowych jest równy 0,01 μs. Jeżeli ustawimy parametry Timera1 (*Counter Settings*) w następujący sposób: *Prescaler*=**0**, *Counter Mode*=**Up**, *Counter Period*=**1000**, *Internal Clock Division*=**0**, *Repetition Counter*=**0**, to okres przebiegu PWM będzie równy 0,01 μs×1000=10 μs. Następnie przechodzimy do ustawienia parametrów trzech kanałów sterujących katodami diody RGB. Ustawiamy parametry pierwszego kanału sterującego katodą diody niebieskiej *PWM Generation Channel 1N*: *Mode*=**PWM mode 1**, *Pulse*=**5**, *Fast Mode*=**Disable**, *CHN Polarity*=**Low**. Parametr *Pulse* wpływa na wypełnienie przebiegu. Dla wartości 5 będzie ono wynosiło (5/1000)×100=0,5%.

Poziomem aktywnym przebiegu PWM będzie poziom niski. W podobny sposób ustawiamy parametry pozostałych 2 kanałów sterujących katodami diod czerwonej



Rysunek 6. Widok zakładki *TIM1 Configuration*

i zielonej. Widok zakładki *TIM1 Configuration* pokazano na **rysunku 6**.

Na koniec trzeba skonfigurować wyprowadzenia mikrokontrolera. W tym celu otwieramy zakładkę *TIM1 Configuration* → *GPIO Settings* i klikamy na pozycję opisującą parametry portu PB13. Po jej rozwinięciu wprowadzamy następujące ustawienia: *GPIO mode*=**Alternate Function Open Drain**, *GPIO PullUp/PullDown*=**No pull-up and no pull-down**, *Maximum output speed*=**High**, *User Label*=**LED_B**. Podobnie postępujemy konfigurując wyprowadzenia PB14 (LED_R) i PB15 (LED_G).

Na koniec należy wygenerować pliki zawierające szkielet oprogramowania. Jeżeli używamy pakietu kompilatora *AC6 System Workbench for STM32*, należy w ustawieniach zaznaczyć opcję: *Project* → *Settings* → *Tolchain/IDE*: SW4STM32. Dokładny opis generowania plików szkieletu oprogramowania oraz sposobu ich zaimportowania do kompilatora AC6 był opisany w poprzednim artykule.

Uruchamianie przykładowego programu

W wygenerowanym przez STM32CubeMX szkielecie oprogramowania zawarte są wszystkie niezbędne procedury inicjujące: wewnętrznych przebiegów taktujących mikrokontrolera, wyprowadzeń sterujących diodami LED, Timera1 przygotowanego do generowania trzech przebiegów PWM o stałym wypełnieniu. Fragment automatycznie stworzonego oprogramowania inicjujący Timer1 pokazano na **listingu 1**.

Na koniec pozostało jedynie uruchomienie procesu generowania przebiegu PWM. Ponieważ korzystamy z biblioteki HAL, należy zajrzeć do jej dokumentacji np. zamieszczonej na stronie firmy ST *„UM1725: Description of STM32F4xx HAL drivers”*. Tam w sekcji poświęconej Timerom *HAL TIM Extension Driver* znajduje się procedura HAL_TIMEx_PWMN_Start(TIM_HandleTypeDef * htim, uint32_t Channel. Jej parametrami są: wskaźnik do struktury parametrów Timera1 i numer kanału, dla którego ma być uruchomione generowanie przebiegu PWM. Kilka linii dodanych do istniejącego szkieletu oprogramowania może wyglądać tak:

```
/* USER CODE BEGIN 2 */
htim1.Instance = TIM1;
```

Listing 1. Inicjalizowanie układu czasowo – licznikowego Timer1

```
/* TIM1 init function */
void MX_TIM1_Init(void)
{
    TIM_ClockConfigTypeDef sClockSourceConfig;
    TIM_MasterConfigTypeDef sMasterConfig;
    TIM_BreakDeadTimeConfigTypeDef sBreakDeadTimeConfig;
    TIM_OC_InitTypeDef sConfigOC;
    htim1.Instance = TIM1;
    htim1.Init.Prescaler = 0;
    htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim1.Init.Period = 1000;
    htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    htim1.Init.RepetitionCounter = 0;
    HAL_TIM_Base_Init(&htim1);
    sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
    HAL_TIM_ConfigClockSource(&htim1, &sClockSourceConfig);
    HAL_TIM_PWM_Init(&htim1);
    sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig);
    sBreakDeadTimeConfig.OffStateRunMode = TIM_OSSR_ENABLE;
    sBreakDeadTimeConfig.OffStateIDLEMode = TIM_OSSI_ENABLE;
    sBreakDeadTimeConfig.LockLevel = TIM_LOCKLEVEL_1;
    sBreakDeadTimeConfig.DeadTime = 0;
    sBreakDeadTimeConfig.BreakState = TIM_BREAK_DISABLE;
    sBreakDeadTimeConfig.BreakPolarity = TIM_BREAKPOLARITY_HIGH;
    sBreakDeadTimeConfig.AutomaticOutput = TIM_AUTOMATICOUTPUT_ENABLE;
    HAL_TIMEx_ConfigBreakDeadTime(&htim1, &sBreakDeadTimeConfig);
    sConfigOC.OCMode = TIM_OCMODE_PWM1;
    sConfigOC.Pulse = 5;
    sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
    sConfigOC.OCNPolarity = TIM_OCNPOLARITY_LOW;
    sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
    sConfigOC.OCIdleState = TIM_OCIDLESTATE_RESET;
    sConfigOC.OCNIdleState = TIM_OCNIDLESTATE_RESET;
    HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_1);
    sConfigOC.Pulse = 10;
    HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_2);
    sConfigOC.Pulse = 15;
    HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_3);
  HAL_TIM_MspPostInit(&htim1);
}
```

Listing 2. Procedura obsługi przerwania od Timera1

```
/* USER CODE BEGIN 4 */
void HAL_TIM_PWM_PulseFinishedCallback (TIM_HandleTypeDef * htim)
{
TIM_OC_InitTypeDef sConfigOC;
    if (htim->Channel ==HAL_TIM_ACTIVE_CHANNEL_1)
    {
        poziom_LED_B_podzielnik++;
        if (poziom_LED_B_podzielnik <200) return;
        poziom_LED_B_podzielnik =0;
        if (poziom_LED_B_UP ==1)
        {
            poziom_LED_B++;
            if (poziom_LED_B >=999) poziom_LED_B_UP =0;
        }
        else
        {
            poziom_LED_B--;
            if (poziom_LED_B <=1) poziom_LED_B_UP =1;
        }
        HAL_TIMEx_PWMN_Stop_IT(&htim1, TIM_CHANNEL_1);
        sConfigOC.OCMode = TIM_OCMODE_PWM1;
        sConfigOC.Pulse = poziom_LED_B;
        sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
        sConfigOC.OCNPolarity = TIM_OCNPOLARITY_LOW;
        sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
        sConfigOC.OCIdleState = TIM_OCIDLESTATE_RESET;
        sConfigOC.OCNIdleState = TIM_OCNIDLESTATE_RESET;
        HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_1);
        HAL_TIMEx_PWMN_Start_IT(&htim1, TIM_CHANNEL_1);
         }
}
/* USER CODE END 4 */
```

```
HAL_TIMEx_PWMN_Start(&htim1,
TIM_CHANNEL_1);
HAL_TIMEx_PWMN_Start(&htim1,
TIM_CHANNEL_2);
HAL_TIMEx_PWMN_Start(&htim1,
TIM_CHANNEL_3);
/* USER CODE END 2 */
```

Po skompilowaniu i zapisie do pamięci Flash mikrokontrolera zamontowanego na płytce KA-NUCLEO-F411CE, przebiegi PWM zaczną być generowane, a poszczególne diody zawarte w strukturze RGB zaczną świecić zmieniając jasność świecenia.

Płynna zmiana jasności świecenia diod LED

Dla uzyskania płynnej regulacji świecenia diody LED w czasie pracy należy zmieniać wypełnienie generowanego przebiegu PWM. Można w tym celu wykorzystać przerwanie generowane po zakończeniu każdego okresu PWM. Ten sposób wymaga powrotu na chwilę do *STM32CubeMX* w celu wykonania niewielkiej modyfikacji szkieletu oprogramowania.

Otwieramy zakładkę *TIM1 Configuration* → *NVIC Settings* i zaznaczamy opcję ***TIM1 capture compare interrupt***. Ponownie generujemy szkielet kodu. Teraz oprogramowanie jest gotowe do obsługi przerwań generowanych przez Timer1. Można się o tym przekonać podglądając plik **stm32F4xx_it.c**, w którym została dodana nowa sekcja:

```
* @brief This function handles
TIM1 capture compare interrupt.
*/
void TIM1_CC_IRQHandler(void)
{
    /* USER CODE BEGIN TIM1_CC_IRQn 0 */
    /* USER CODE END TIM1_CC_IRQn 0 */
    HAL_TIM_IRQHandler(&htim1);
    /* USER CODE BEGIN TIM1_CC_IRQn 1 */
    /* USER CODE END TIM1_CC_IRQn 1 */
}
```

Użytkownikowi pozostają do zrobienia dwie czynności: zmiana procedury inicjacji PWM i dopisanie obsługi przerwania.

Pierwsza zmiana będzie wyglądała następująco:

```
/* USER CODE BEGIN 2 */
htim1.Instance = TIM1;
HAL_TIMEx_PWMN_Start_
IT(&htim1, TIM_CHANNEL_1);
HAL_TIMEx_PWMN_Start_
IT(&htim1, TIM_CHANNEL_2);
HAL_TIMEx_PWMN_Start_
IT(&htim1, TIM_CHANNEL_3);
/* USER CODE END 2 */
```

Obsługa przerwania nastąpi w zdefiniowanej przez biblioteki HAL funkcji void HAL_TIM_PWM_PulseFinishedCallback (TIM_HandleTypeDef * htim). Po wystąpieniu przerwania generowanego po każdym okresie przebiegu PWM, oprogramowanie automatycznie wywoła funkcję, w której można dodać własną procedurę obsługi przerwania. Będzie ona polegała na okresowej, płynnej zmianie wypełnienia przebiegu PWM, np. sterującego diodą niebieską. W celu zapewnienia płynności zmiana wypełnienia musi być wykonywana w odpowiednim tempie. Przy ustawionych parametrach PWM (okres równy 10 μs) zmiana wypełnienia o 1/1000 będzie wykonywana co 200 okresów PWM.

W celu wykonania koniecznych obliczeń należy zadeklarować dodatkowe zmienne globalne:

```
/* USER CODE BEGIN PV */
/* Private variables */
int poziom_LED_B =5, poziom_LED_B_
UP=1, poziom_LED_B_podzielnik=0;
/* USER CODE END PV */
```

Wykonaną przez mnie procedurę obsługi przerwania pokazano na **listingu 2**. Efektem jej działania powinno być płynne rozjaśnianie i gaszenie diody niebieskiej trójkolorowego LED na płytce KA-NUCLEO-F411CE.

Ryszard Szymaniak, EP

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: **ftp://ep.com.pl**, user: **11875**, pass: **6hhcxxtt**