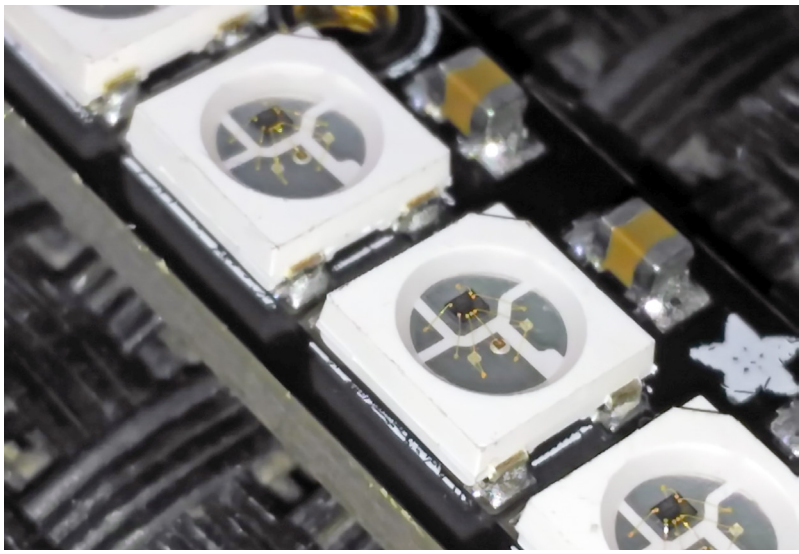
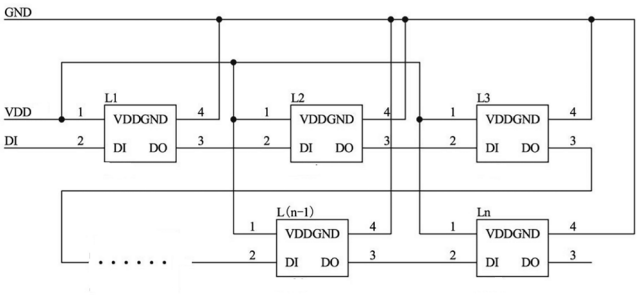
NOTATNIK KONSTRUKTORA

Sterowanie WS2812 – nowe podejście

Od jakiegoś czasu można kupić trójkolorowe diody LED zintegrowanie ze sterownikiem sterowanym cyfrowo. Sterownik jest oferowany w dwóch odmianach: WS8211 i WS8212. Oba działają tak samo, a różnice w oznaczeniach oznaczają trochę inne zależności czasowe przy przesyłaniu danych do układów. Z diod LED można budować „ekrany", węże świetlne czy inne świecące elementy ozdobne.

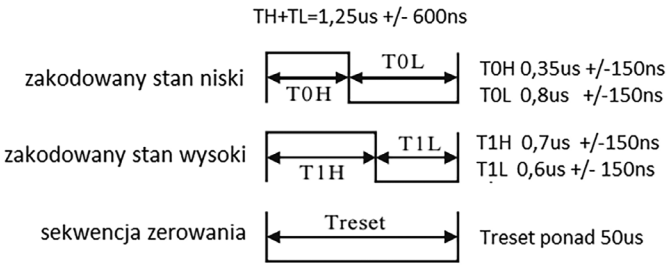


Idea budowania zestawów wielu diod jest bardzo nieskomplikowana – przyjęto, że układy oprócz wyprowadzeń zasilania będą miały jedną linię danych wejściowych DIN i jedną linię danych wyjściowych DOUT. „Magistrala" sterująca jest jedną linią, po której są przesyłane szeregowo dane i powstaje z połączeń wyprowadzeń DOUT jednego układu z linią DIN następnego układu. Zostało to pokazane na **rysunku 1**. Maksymalne uproszczenie sprzętowe zawsze jest połączone ze skomplikowanym przesyłaniem danych. Każdy, kto próbował realizować komunikację z układami 1-Wire, wie, że jest z tym pewien kłopot. Nie inaczej jest w tym wypadku. Ponieważ nie ma linii taktującej przesyłaniem danych, to muszą być one tak zakodowane, aby układ sterownika LED mógł odtworzyć sobie zegar sterujący transmisją. Poza tym musi być jakiś sposób na adresowanie kolejnych diod w łańcuchu, ponieważ animacje graficzne będą wymagały szybkiego, szeregowego przesyłania danych do wielu diod w układzie. Należy się spodziewać dużej prędkości transmisji danych kodowanych długością impulsu.



Rysunek 1. Łączenie układów WS2811/WS2812 w łańcuchy

Zajmiemy się sterownikiem WS1282 – dioda typu WS8211 jest sterowana bardzo podobnie. Jak już wspominałem, przesyłanie danych jedną linią będzie wymagało kodowania bitów czasem trwania impulsów. Przyjęto, że czas przesłania jednego bitu będzie taki sam dla poziomu niskiego i wysokiego, a bity będą rozróżniane poprzez różny czas trwania poziomu wysokiego. Sposób kodowania poziomów logicznych bitów pokazano na **rysunku 2**. Aby ułatwić implementację algorytmów sterujących, dopuszczono odchyłki od standardowych czasów stanów TH i TL wynoszące ±150 ns i ±600 ns dla czasu przesłania jednego bitu.



Rysunek 2. Sposób kodowania danych przesyłanych magistralą

Czas potrzebny na przesłanie 1 bitu to nominalnie 1,25 µs. Z tego wynika prędkość transmisji równa 800 kb/s. Oprócz dwóch bitów, jedynki i zera, na magistrali można wystawić polecenie zerowania. Jest to poziom niski na linii danych przez czas dłuższy niż 50 µs.

Każdy ze sterowników oczekuje na przesłanie trzech bajtów, czyli 24 bitów. W tych trzech bajtach jest zawarta informacja o intensywności świecenia każdej z trzech diod według zasady pokazanej na **rysunku 3**. W ten sposób otrzymujemy świecący „punkt", którego kolor ma głębię 24-bitową.

Po włączeniu zasilania sterowników WS2812 wszystkie diody są zgaszone i każdy ze sterowników oczekuje na dane. Pierwsze 3 bajty wysłane na magistralę są odbierane przez sterownik WS2812, którego linia DIN jest połączona z linią portu mikrokontrolera i zaświeca się dioda tego sterownika. Kolejne bajty będą już ignorowane przez ten układ, ale zostaną przekazane bezpośrednio na linię DOUT. Inaczej mówiąc, po odebraniu „swoich" 3 bajtów sterownik staje się przeźroczysty dla danych. Drugi sterownik WS2812 odbierze kolejne 3 bajty (mikrokontroler wysyła w sumie 6), zaświeci swoją diodę, a kolejne bajty zignoruje i będzie przesyłał na swoją linię DOUT. W ten sposób kolejne sterowniki są adresowane kolejnością przesyłanych danych. Kolejne bajty danych muszą być przesyłane jeden za drugim. Jeżeli na magistrali wystąpi poziom niski przez czas dłuższy niż 50 µs, to wszystkie sterowniki w łańcuchu wyzerują swoje układy wykrywające odebrane dane i kolejne przesyłane dane będą odbierane ponownie od pierwszego sterownika, czyli tego, który ma linię DIN połączoną z linią portu sterownika mikroprocesorowego (**rysunek 4**).

Na pierwszy rzut oka sterowanie nie wydaje się skomplikowane. Jednak jeżeli rozważymy zależności czasowe, to praktyczna realizacja transmisji okaże się niełatwa do wykonania, ponieważ sterownik mikroprocesorowy musi wysyłać zakodowane czasowo bajty z dużą prędkością 800 kb/s.

G7	G6	G5	G4	G4	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0	B7	B6	B5	B4	B3	B2	B1	B0

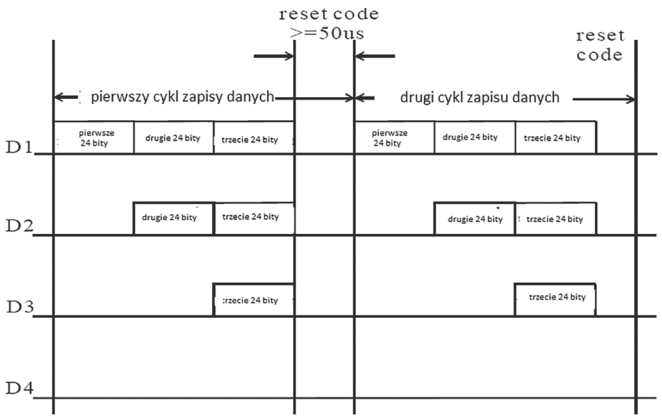
Rysunek 3. Sterowanie intensywnością każdej z diod WS2812

Najprostszą implementacją algorytmu sterowania będzie odliczanie programowych opóźnień i modyfikacja poziomów logicznych na linii portu zależnie od przesłanej wartości. To rozwiązanie ma jedną zaletę – jest łatwe do zrealizowania. Niestety, ma też wadę, która je dyskwalifikuje w praktycznych zastosowaniach. Załóżmy, że chcemy sterować diodami za pomocą taniego mikrokontrolera PIC16F taktowanego częstotliwością 16 MHz. Proste oszacowanie czasu potrzebnego na wykonanie zadania pokazuje, że zajmie nam to prawie 100% wydajności mikrokontrolera i to przy dobrze zoptymalizowanym kodzie programu. Możemy zaświecać diody, ale nie będziemy w stanie wykonać żadnych sekwencji sterujących wyświetlanymi kolorami. Zastosowanie wydajniejszej jednostki bez zmiany podejścia do sposobu sterowania również niewiele pomoże.

Skoro programowo nie można, to trzeba próbować wykorzystać układy sprzętowe. Pierwsza myśl, która mi przyszła do głowy, to użycie generatora PWM i programowa modyfikacja współczynnika wypełnienia na podstawie wartości przesyłanego bitu. Jednak w typowej implementacji modułu PWM nie ma sprzętowej informacji o tym, kiedy kończy się okres przebiegu. Użycie poolingu sprowadza się wprost do metody pierwszej. Można użyć przerwania od timera taktującego PWM, ale całość robi się skomplikowana i może wprowadzać niedopuszczalne opóźnienia.

Kolejny pomysł to użycie układu SPI do przesłania pojedynczego bitu. Do przesłania logicznego 0 wysyłalibyśmy sekwencję 11100000, a dla logicznej jedynki 11111000. Wysłanie jednego bitu wiązałoby się z wysłaniem jednego bajtu przez SPI, czyli dane przesyłane przez SPI musiałyby mieć prędkość 8×0,8 Mb/s=6,4 Mb/s, a zegar taktujący częstotliwość 12,8 MHz. W „dużych”, 32-bitowych mikrokontrolerach taktowanych przebiegiem o częstotliwości 70…100 MHz jest to możliwe i takie implementacje są spotykane, jednak w naszym wypadku to się nie uda. Nie da się uzyskać taktowania SPI o częstotliwości 12,8 MHz przy taktowaniu mikrokontrolera częstotliwością 16 MHz. Kolejny pomysł to użycie przerwania od timera i sterowanie z zastosowaniem maszyny stanów. Przerwania musiałyby być zgłaszane z częstotliwością równą 1/0,35 μs=2,85 MHz. Przy taktowaniu częstotliwością 16 MHz rdzeń i peryferie PIC16F są taktowane częstotliwością FOSC/4, czyli 4 MHz. Ten pomysł również nie może być zrealizowany.

Najlepszym praktycznym rozwiązaniem dla sterowników tego typu jest nieblokująca praca w tle. Zapisujemy jakiś bufor w pamięci RAM mieszczący n×3 bajtów (gdzie „n” to liczba diod na magistrali), a oprogramowanie wysyła dane, jak najmniej absorbując rdzeń mikrokontrolera. Użyteczne sterowanie prostymi metodami przy użyciu standardowych układów peryferyjnych jest albo niemożliwe, albo bardzo skomplikowane. Jednak wykorzystując nowe układy peryferyjne Microchipa wbudowane w mikrokontrolery z rodziny PIC16F1xxx pracujące niezależnie od rdzenia mikrokontrolera i stosując niestandardowe, „sprytne” podejście do problemu, można wykonać taki sterownik mikroprocesorowy obciążający rdzeń w minimalnym stopniu.



Rysunek 4. Cykle zapisu danych do sterowników diod

Idea takiego rozwiązania została opisana w nocie aplikacyjnej Microchipa AN1606. Oryginalnie nota jest przeznaczona dla mikrokontrolera PIC16F1509 i mimo że mam taki układ, zdecydowałem się użyć nowszego – PIC16F1619. Ma on możliwość bardziej elastycznej konfiguracji wejść modułów peryferyjnych CLC i przez to cały projekt jest łatwiejszy do wykonania. Całość była testowana z użyciem firmowego modułu Curiosity, ale można do tego celu zbudować dowolny układ. Do budowy sterownika diod WS2812 zastosujemy 4 sprzętowe układy peryferyjne:

1. Układ licznika TMR2 do generowania przebiegu taktującego transmisją.
2. Układ PWM do generowania części przebiegu odpowiedzialnego za czasową sekwencję poziomu niskiego.
3. Układ interfejsu SPI pracującego w trybie Master do wysyłania 8 bitów danych.
4. Układ komórki logicznej CLC do wykonania funkcji logicznych niezbędnych do zakodowania czasowego wysyłanych bitów.

Jak już wiemy, bity muszą być wysyłane z prędkością 800 kb/s. Intuicyjnie czujemy, że interfejs SPI należy zaprogramować tak, aby wysyłał dane z taką prędkością. Z rysunku 2 wynika, że czas trwania pojedynczego bitu to 1,25 μs ±0,6 μs. Ta tolerancja pomoże nam w realizacji zadania.

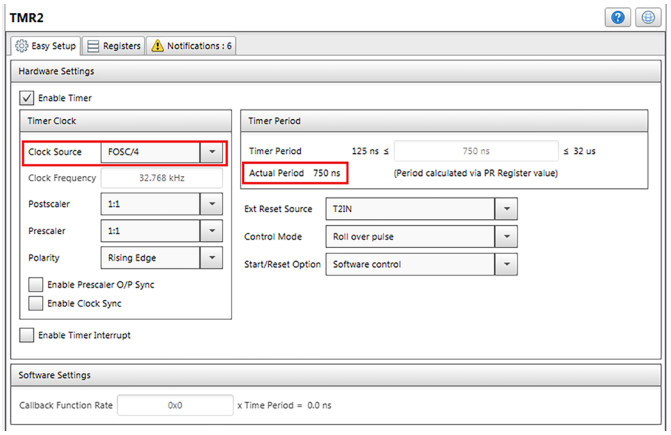
Prędkość transmisji 800 kb/s oznacza, że zegar taktujący transmisją musi mieć częstotliwość 800 kHz×2=1,6 MHz. Okres tego przebiegu to 625 ns. Licznik T2 w mikrokontrolerze taktowanym z częstotliwością 16 MHz może generować przepełnienia z rozdzielczością 250 ns. Czyli możemy uzyskać okres 500 ns lub 750 ns – ja zaprogramowałem okres 750 ns. Czas przesyłania jednego bitu przez SPI będzie wynosił 1,5 μs, czyli mieści się w tolerancji. Te 750 ns przyda się też do generowania czasu potrzebnego do zakodowania logicznej jedynki, ale o tym za chwilę.

Licznik T2 będzie taktować transmisję SPI, ale też będzie źródłem zegara taktującego dla modułu PWM. Jest to bardzo ważne, bo zapewnia czasową synchronizację generowanych przebiegów. Do skonfigurowania układów peryferyjnych wykorzystałem środowisko MPLAB X IDE i wtyczkę MCC (MPLAB Code Configurator). Całość została skompilowana bezpłatną pełną wersją kompilatora MPLAB XC8.

Programowanie układów peryferyjnych zaczniemy od licznika T2. Licznik będzie się cyklicznie przepełniał co 750 ns, czyli z częstotliwością 1,3333333 MHz. Jeżeli ten licznik będzie źródłem przebiegu taktującego transmisję, to dane przez interfejs SPI będą przesyłane z prędkością ok. 667 kHz.

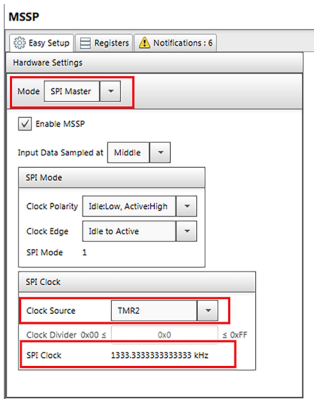
Na **rysunku 5** pokazano konfigurację Timera2 wykonaną za pomocą MCC. Po skonfigurowaniu timera możemy wykorzystać sygnał cyklicznego przepełnienia do taktowania układów peryferyjnych. W naszym wypadku będzie to interfejs SPI i układ PWM3. W konfiguracji SPI wybieramy: tryb pracy Master (moduł SPI jest źródłem sygnału zegarowego SCK), źródło sygnału taktującego moduł Timer2 (**rysunek 6**). Tak skonfigurowany moduł będzie wysyłał bity z prędkością 666,67 kb/s. Na **rysunku 7** pokazano oscylogram przebiegów sygnału danych SDO (kanał 2) i zegarowego SCK (kanał 1) w czasie wysyłania przez SPI bajtu 0x55.

Przechodzimy teraz do zasadniczej części projektu, czyli kodowania danych przesyłanych do sterowników WS2812. Jak już wiemy,



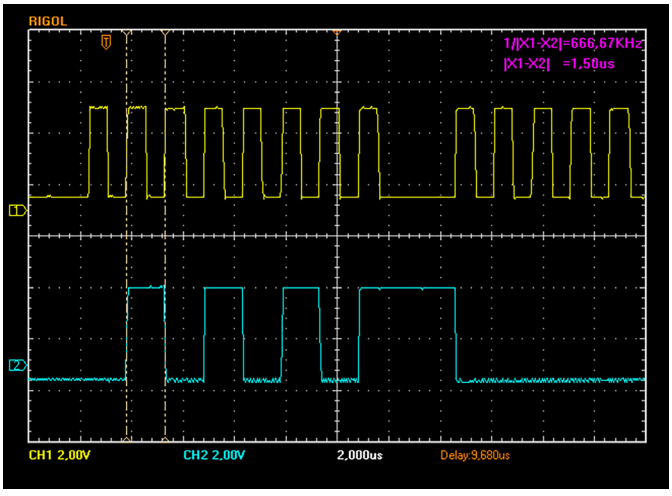
Rysunek 5. Konfiguracja licznika Timer2

potrzebne są dwa czasowo zakodowane bity – logiczne zero i jedynka. Jedynkę będziemy kodowali przez wykonanie funkcji logicznej AND sygnału danej i sygnału zegarowego interfejsu SPI. Kiedy na linii danych jest poziom wysoki, to na wyjściu układu realizującego pojawi się okres sygnału zegarowego taktującego transmisję. Jak już wiemy, częstotliwość SCK jest równa 666,67 kHz, a okres 1,50 µs. Czas trwania poziomu wysokiego wynosi 750 ns. Według specyfikacji czas trwania stanu T1H przy przesyłaniu zakodowanego poziomu wysokiego powinien wynosić 700 ns ±150 ns, czyli wszytko jest w porządku.

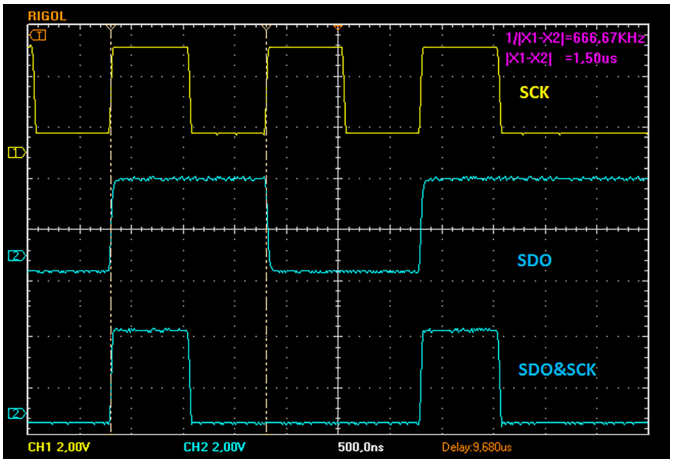


Rysunek 6. Konfiguracja modułu SPI

Jak już wiemy, okres przesyłania zakodowanego bitu według specyfikacji to 1,25 µs ±600 ns i nasze 1,5 µs również się mieści w tolerancji. Na **rysunku 8** pokazano przebiegi sygnału SCK, SDA, a na samym dole SCK&SDO. Kiedy na linii danych jest poziom wysoki, to na linii reprezentującej SCK&SDA jest jeden impuls zegara, oczywiście zboczami zsynchronizowany z linią danych SCK. Kiedy na linii danych jest poziom niski, to na SCK&SDA jest również poziom niski. Oscylogram z rysunku 8 zarejestrowano na wyprowadzeniu z modułu CLC wykonującego funkcję SCK&SDA. Kodowanie logicznej jedynki jest banalne, bo możemy wykorzystać tolerancje czasowe protokołu komunikacyjnego WS2812. W kolejnym kroku musimy znaleźć sposób na zakodowanie bitu logicznego niskiego. Do tego celu wykorzystamy układ peryferyjny PWM3. Źródłem sygnału dla tego układu

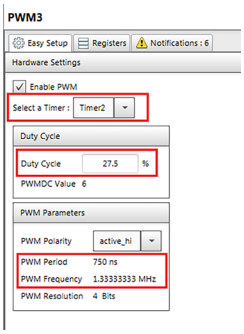


Rysunek 7. Wysyłanie danych przez SPI z prędkością 666,67 kbit/s



Rysunek 8. Kodowanie poziomu niskiego za pomocą funkcji SCK&SDO

będzie również Timer2. Jak już wspomniałem, gwarantuje to bardzo ważną synchronizację zboczy wszystkich generowanych sygnałów. Generator PWM3 skonfigurujemy tak, aby poziom wysoki przebiegu był na tyle krótki, żeby mieścił się w tolerancji dla T0H równej 0,35 µs ±150 ns. Konfiguracja PWM3 została pokazana na **rysunku 9**. Okres PWM wynosi 750 ns, a współczynnik wypełnienia ok. 30%, więc czas trwania poziomu wysokiego to ok. 210 ns. Jest to na granicy tolerancji, ale w praktyce nie ma z tym problemu. Przy tych częstotliwościach rozdzielczość sygnału PWM spada do 4 bitów i w praktyce nie ma żadnego „pola manewru".

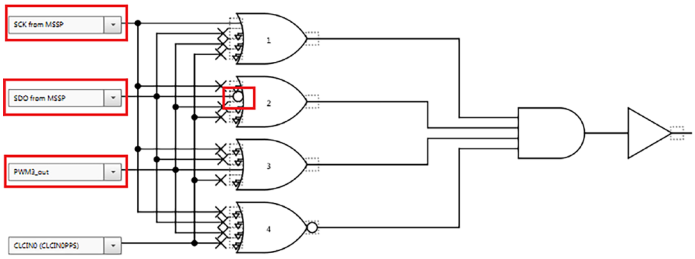


Rysunek 9. Konfiguracja modułu PWM3

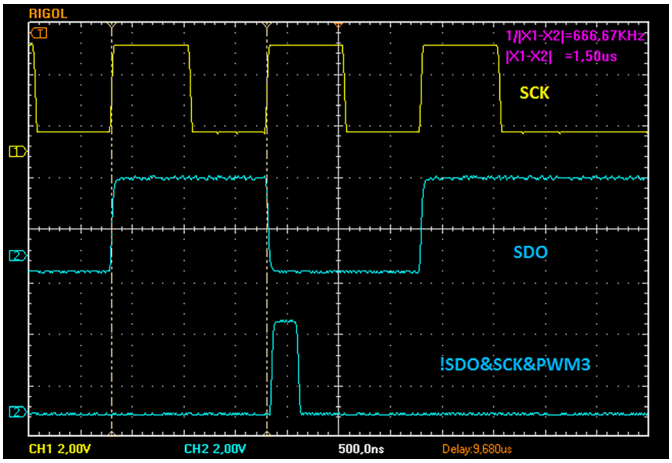
Teraz musimy określić funkcję logiczną, która w czasie trwania logicznej jedynki wygeneruje na wyjściu jeden impuls sygnału PWM. Najprościej będzie zanegować sygnał SDO i zrobić iloczyn logiczny z sygnałami SCK i PWM, czyli funkcja generująca poziom niski będzie równa (negacja) SDO&SCK&PWM3. Na potrzeby testowania układu zdefiniowałem układ CLC1 wykonujący tę funkcję (**rysunek 10**). Oscylogram generowania logicznego zera pokazano na **rysunku 11**.

Mamy dwa osobne przebiegi, które teraz trzeba złożyć w jeden. Ponieważ oba przebiegi dla przeciwstawnych bitów mają poziom niski, można je zsumować. Otrzymujemy zatem funkcję (SDO&SCK)|||(!SDO&SCK&PWM3). Moduł CLC1 wykonuje operację !SDO&SDA&PWM3. Skonfigurujemy teraz kolejny moduł CLC3 sumujący wyjście z CLC1 z iloczynem sygnałów SDO i CLC odpowiadającym za kodowanie bitu wysokiego – **rysunek 12**. Przebieg na wyjściu CLC3 pokazano na **rysunku 13**. Sygnał z wyjścia modułu CLC3 przekierowujemy na jedną z linii portów i można sterować układami WS2812.

Żaden z układów CLC nie może wykonać samodzielnie funkcji (SDO&SCK)|||(!SDO&SCK&PWM3) i trzeba użyć dwóch modułów. Można zastosować metody minimalizacji funkcji logicznych i tak ją



Rysunek 10. Konfiguracja CLC1 realizującej funkcję !SDO&SDA&PWM3



Rysunek 11. Kodowanie poziomu logicznego niskiego

przekształcić, żeby była wykonywana tylko przez jedne CLC. Na **rysunku 14** pokazano konfigurację CLC równoważną naszej funkcji.

Pierwsze próby działania polegały na przesyłaniu danych za pomocą funkcji *uint8_t SPI_Exchange8bit(uint8_t data)* wpisującej daną do rejestru SSPBUF modułu SPI i czekającej w pętli na ustawienie bitu informującego o wysłaniu wszystkich 8 bitów przez wyjście SDO (**listing 1**). Oczywiście, użycie tej funkcji nie może być docelowe, bo czekanie w pętli na przesłanie bajtu całkowicie blokuje mikrokontroler. Jednak do początkowych testów zupełnie to wystarcza. Początkowo napisałem sobie funkcję wysyłającą 3 kolejne bajty wyliczane na podstawie 24-bitowej liczby określającej kolor (**listing 2**). Niestety, użycie tej funkcji nie dawało spodziewanych rezultatów. Oscyloskop pokazał, że pierwszy bajt jest wysyłany z opóźnieniem ok. 10 µs. Ten czas jest wykorzystywany przez mikrokontroler do wywołania

Listing 1. Funkcja wysyłająca jeden bajt przez SPI

```
uint8_t SPI_Exchange8bit(uint8_t data)
{
    // Clear the Write Collision flag, to allow writing
    SSP1CON1bits.WCOL = 0;
    SSPBUF = data;
    while(SSP1STATbits.BF == SPI_RX_IN_PROGRESS)
    {
    }
    return (SSPBUF);
}
```

Listing 2. Wysłanie 3 bajtów koloru

```
void SendColor(uint24_t color)
{
    SPI_Exchange8bit(color>>16);
    SPI_Exchange8bit(color>>8);
    SPI_Exchange8bit(color);
}
```

Listing 3. Zmodyfikowana procedura wysłania 3 bajtów koloru

```
void SenColor(uint8_t G,uint8_t R, uint8_t B)
{
    SPI_Exchange8bit(G);
    SPI_Exchange8bit(R);
    SPI_Exchange8bit(B);
}
```

funkcji i wykonania przesunięcia zmiennej *color* o 16 pozycji w prawo i jest wynikiem użycia stosunkowo wolnego mikrokontrolera i nieoptymalizowanej wersji kompilatora XC8. Czas 10 µs jest dłuższy niż potrzebny na wysłanie 1 bajtu i najprawdopodobniej sterownik WS81282 po takiej przerwie ignorował ten bajt. Można to było ominąć, stosując fragment funkcji napisany w asemblerze, ale to nie było istotne w prowadzonych testach.

Zmodyfikowaną procedurę wysłania 3 bajtów pokazano na **listingu 3**. Funkcja używa trzech 8-bitowych argumentów, osobnego dla każdego z kolorów składowych. Po użyciu tej funkcji pierwszy bajt był wysyłany z opóźnieniem ok. 5 µs i wszystko zaczęło działać

Listing 4. Nieskończona pętla zaświecająca cyklicznie 24 diody

```
color=0x550000;
while(1)
{
    g=color>>16;
    r=color>>8;
    b=color;
    for(i=0;i<8;i++)
    {
        SPI_Exchange8bit(g);
        SPI_Exchange8bit(r);
        SPI_Exchange8bit(b);
        SPI_Exchange8bit(b);
        SPI_Exchange8bit(g);
        SPI_Exchange8bit(r);
        SPI_Exchange8bit(r);
        SPI_Exchange8bit(b);
        SPI_Exchange8bit(g);
        SPI_Exchange8bit(0);
    }
    __delay_ms(100);
    color=(color>>8);
    if(color==0)
    {
        color=0x550000;
    }
}
```

Listing 5. Funkcja obsługi przerwań

```
void interrupt INTERRUPT_InterruptManager (void)
{
    //interrupt handler
    if(INTCONbits.PEIE == 1 && PIE1bits.SSP1IE == 1 && PIR1bits.
SSP1IF == 1)
    {
        SPI_ISR();
    }
    else
    {
        //Unhandled Interrupt
    }
}
```

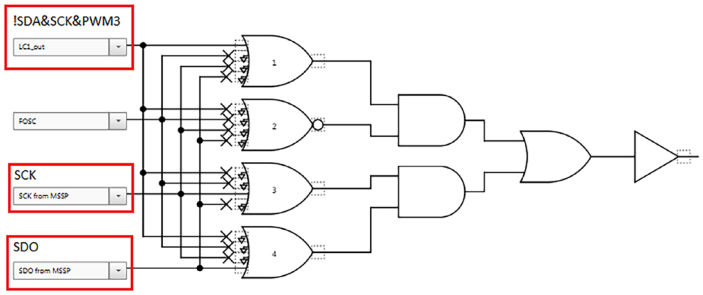
Listing 6. Procedura obsługi przerwania od SPI

```
void SPI_ISR(void)
{

    // clear the SPI interrupt flag
    PIR1bits.SSP1IF=0;

    WS2812_Send();

}
```

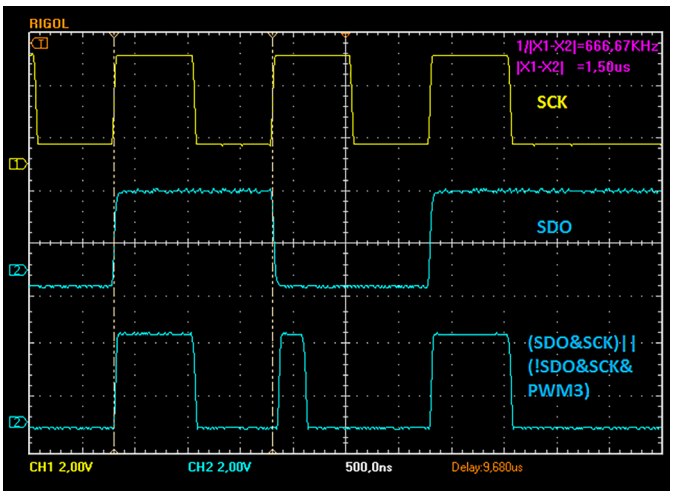
prawidłowo. Kolejny listing pokazuje fragment programu zaświecający cyklicznie wszystkie diody w moim module składającym się z 24 diod.

Ta faza testów miała za zadania określenie, czy generowane przebiegi sterujące powtarzane w nieskończonej pętli nie powodują błędnego sterowania układami WS2812. Uruchomienie programu przez dłuższy czas nie powodowało widocznego błędnego działania. Kolejny etap testów to wykonanie aplikacji, która nie blokuje czasu procesora. Do tego celu postaramy się użyć przerwań generowanych przez moduł SPI. To, co ma być wyświetlane, zostanie zawarte w buforze pamięci mieszczącym 24×3=72 bajty. Wszelkie modyfikacje wyświetlanej informacji będą się sprowadzały do modyfikacji zwartości tego bufora i zainicjowania procesu wysłania jego zwartości do sterowników WS2812.

W pierwszym kroku trzeba zdefiniować obsługę przerwania od modułu SPI. Przerwanie jest zgłaszane przez ustawienie znacznika SSP1IF po wysłaniu ostatniego, ósmego bitu przez wyjście SDO. Wcześniej trzeba odblokować: globalny system przerwań,



Rysunek 12. Generowanie przebiegu (SDO&SCK)||(!SDO&SCK&PWM3)



Rysunek 13. Generowanie pełnego przebiegu sterującego

system przerwań od modułów peryferyjnych i przerwanie od modułu SPI. Globalny system przerwań jest odblokowywany przez biblioteczną funkcję, a właściwie makro *INTERRUPT_GlobalInterruptEnable()*, system przerwań przez makro *INTERRUPT_PeripheralInterruptEnable()*, a przerwania od modułu MSSP skonfigurowanego do pracy SPI Master przez ustawienie bitu SSP1IE w rejestrze *PIE1: PIE1bits.SSP1IE=1*. Generowanie funkcji obsługi możemy „zlecić" wtyczce MCC. W oknie *Interrupt Module* zaznaczamy opcję przerwania zgłaszanego po opróżnieniu rejestru SSPBUF (**rysunek 15**).

Po wygenerowaniu plików przez MCC w pliku *interrupt_manager.c* zostaje zdefiniowana funkcja *interrupt INTERRUPT_InterruptManager(void)* pokazana na **listingu 5**. Oryginalnie procedura obsługi po ustawieniu SSPI1IF miała nie wiedzieć czemu nazwę *I2C_ISR()* – zmieniłem ją na odpowiedniejszą *SPI_ISR()*. MCC nie generuje szkieletu SPI_I2C i po zmianie nazwy SPI_ISR trzeba ją sobie samemu napisać od początku (**listing 6**). Po koniecznym programowym wyzerowaniu wskaźnika (flagi) SSP1IF jest wywoływana właściwa funkcja wysłania zawartości bufora Buffer[] o rozmiarze 72 bajtów przez interfejs SPI. Tę funkcję pokazano na **listingu 7**.

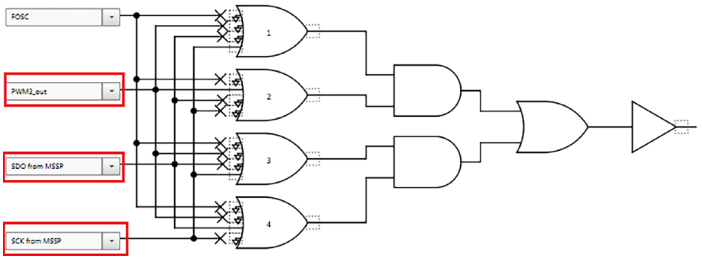
Listing 7. Wysłanie bufora buffer przez SPI

```
volatile uint8_t point=0; //licznik adresujący elementy w buforze
uint8_t buffer[72]={0x55,0,0,0,0x55,0,0,0,0x55,//bufor z predefinio-
wanymi wartościami
0x55,0,0,0,0x55,0,0,0,0x55,
0x55,0,0,0,0x55,0,0,0,0x55,
0x55,0,0,0,0x55,0,0,0,0x55,
0x55,0,0,0,0x55,0,0,0,0x55,
0x55,0,0,0,0x55,0,0,0,0x55,
0x55,0,0,0,0x55,0,0,0,0x55,
0x55,0,0,0,0x55,0,0,0,0x55,};

void WS2812_Send(void){
    if(point>=MAX_BUF_SPI)
    {
        PIE1bits.SSP1IE=0;
        point=0;
        return;
    }
    SSPBUF=buffer[point++];
}
```

Listing 8. Inicjowanie wysyłania zawartości bufora buffer

```
SPBUF = 0;
__delay_ms(1);
PIE1bits.SSP1IE=1;
```

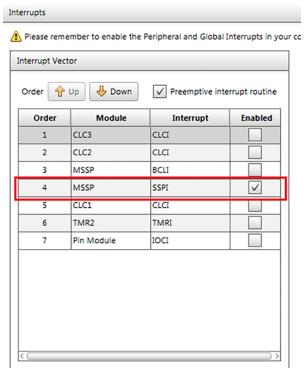
Zmienna point adresuje kolejne bajty z bufora. Żeby transmisja wielu bajtów z wykorzystaniem przerwań mogła działać, najpierw musimy ją zainicjować przez zapisanie bajtu do rejestru SSPBUF. Można wysłać pierwszy bajt z bufora, ale ja wysyłam bajt zerowy, odczekuję 1 ms dla wyzerowania magistrali danych WS1282 i odblokowuję przyjmowanie przerwań. Po wysłaniu bajtu zerowego ustawi się flaga SSPIF i po odblokowaniu przerwań zostanie ono przyjęte. Przyjęcie pierwszego przerwania uruchamia automatyczne wysyłanie kolejnych bajtów z bufora. Po wysłaniu 72 bajtów procedura obsługi przerwania wyzeruje wskaźnik adresowy point i zablokuje przyjmowanie przerwań przez wyzerowanie bitu SSP1IE. Na **listingu 8** pokazano fragment programu uruchamiający wysyłanie zawartości bufora.



Rysunek 14. Funkcja równoważna funkcji (SDO&SCK)||(!SDO&SCK&PWM3)

Tak oto otrzymaliśmy mechanizm pozwalający na sterowanie łańcuchem układów WS1282 z wykorzystaniem unikalnych modułów peryferyjnych mikrokontrolerów rodziny PIC16F1xxx i nietypowe, sprytne podejście do rozwiązania problemu. Wszystko odbywa się w tle programu głównego z wykorzystaniem przerwań i sprzętowych zasobów mikrokontrolera, jednocześnie obciążając go w niewielkim stopniu. Można teraz wymyślać i programować sposoby wizualizacji swoich własnych efektów graficznych. Jednym ze sposobów na zbudowanie w pełni funkcjonalnego sterownika jest dołączenie poprzez magistralę I²C pamięci Flash o dużej pojemności, w której zapiszemy swoje wzorce wyświetlanych sekwencji efektów.

Tomasz Jabłoński, EP



Rysunek 15. Konfigurowanie przerwania od SPI