Sterowanie diodami WS2812 przez UART z AVR, Arduino...

Sterowanie diodami WS2812 nie jest proste ze względu na wymaganą stosunkowo dużą przepływność danych wynoszącą 800kHz. Jeśli diodami chce się sterować z interfejsu UART lub SPI, musi on pracować z prędkością około 2,4MB/s, ponieważ jednemu bitowi LED odpowiadają trzy bity SPI/UART. Jeśli wymagane jest, aby podczas transmisji funkcjonowały przerwania, zadanie jest z pewnością bardzo trudne, a wręcz niemożliwe do wykonania. W artykule przedstawiono rozwiązanie umożliwiające sterowanie LED-ami w przerwaniach. Artykuł dotyczy mikrokontrolera AVR, ale opisanych rozwiązań z powodzeniem można użyć na innych mikrokontrolerach.

Sterowanie WS2812 przez GPIO i funkcje opóźniające jest powszechnie znane. Wadą tego rozwiązania jest blokowanie CPU na czas transmisji. Gdy wysyłane są dane do dużej liczby LED, a odświeżanie jest częste, CPU jest praktycznie zablokowany, bo funkcje transmisji nie mogą być zatrzymywane nawet przez przerwania. Zastanawiałem się więc nad sposobem sterowania WS2812 innym niż użycie do tego celu PIO. W przypadku SPI problemem jest skomplikowane dzielenie wysyłanej informacji do LED, ponieważ aby wysłać 24 bity do LED, trzeba przez SPI wysłać 72 bity (24 bity LED * 3 bity przez SPI), co po podzieleniu przez osiem bitów daje 9 bajtów, a ta liczba nie jest okrągłą wartością z punktu widzenia mikrokontrolera, bo nie jest potęgą liczby dwa. Powstaje problem spowodowany tym, że transmisja nie może zostać zakłócona przez przerwanie, zwłaszcza w sytuacji, gdy bajt kończy się jedynką. Pomyślałem o wykorzystaniu IIC, który nadaje dane 9-bitowe. Gdy na magistrali nie ma układu *slave*, ostatni bit zawsze jest zerem (bit ACK), ale po zanegowaniu otrzymujemy jeden. Dzięki temu w bajcie można przesłać 3 bity, a czas pomiędzy transmisjami nie jest już tak krytyczny jak w SPI. Niestety kontroler wbudowany w AVR nie może pracować z dużymi prędkościami. Przyszedł mi do głowy sposób z użyciem USART w trybie synchronicznym. Później skierowałem swoje rozważania na tryb asynchroniczny. Jak się później okazało, nie byłem jedynym, który wpadł na ten pomysł. Rozwiązanie takie z powodzeniem jest wykorzystywane w Xmega i ARM. Niestety USTART w ATmega czy ATtiny nie może pracować z przepływnością rzędu 2,4MB/s.
Czy aby na pewno?

Zaprezentowane w artykule rozwiązanie pozwala sterować 60 LED-ami (których liczba jest ograniczona przez mały RAM ATmega164), co też ogranicza liczbę warstw wirtualnego ekranu do pięciu. Pod pojęciem warstw należy rozumieć warstwy jak w programie graficznym: tło – warstwa najniższa (warstwa 5), kolejna warstwa nr 4 zasłania tło, następna warstwa (numer 3) zasłania niższe warstwy (w tym przypadku tło i warstwę nr 4), warstwa 0 ma najwyższy priorytet i zasłania wszystko, co się pod nią znajduje. Oczywiście obraz danej warstwy zasłania inne pod warunkiem, że w danym punkcie znajduje się kolor inny niż przezroczysty. Podczas wyświetlania ekranu na diodach możliwe jest w tym samym czasie nawet dekodowanie transmisji DMX, która znacznie obciąża mikrokontroler! Prezentacje animacji LED można zobaczyć w materiałach dodatkowych dostępnych w Elportalu. Zapotrzebowanie na RAM wynosi: liczba LED * 8 + liczba LED * liczba warstw * głębia kolorów w bajtach (**tabela 1**).

Program wymaga minimum 2,4kB FLASH (zaprezentowany efekt na ATmega164 – 3,4kB). Do przechowywania danych można wykorzystać zewnętrzną pamięć RAM, gdyż większość dużych mikorokontrolerów z rodziny AVR posiada 4kB RAM.

Liczba LED	Liczba warstw	Głębia koloru	Wymagany RAM w kB
100	8	256 barw	1,6
100	8	24 bity	3,2
400	4	256 barw	3,6
200	5	16 bitów	3,6
400	8	24 bity	12,8

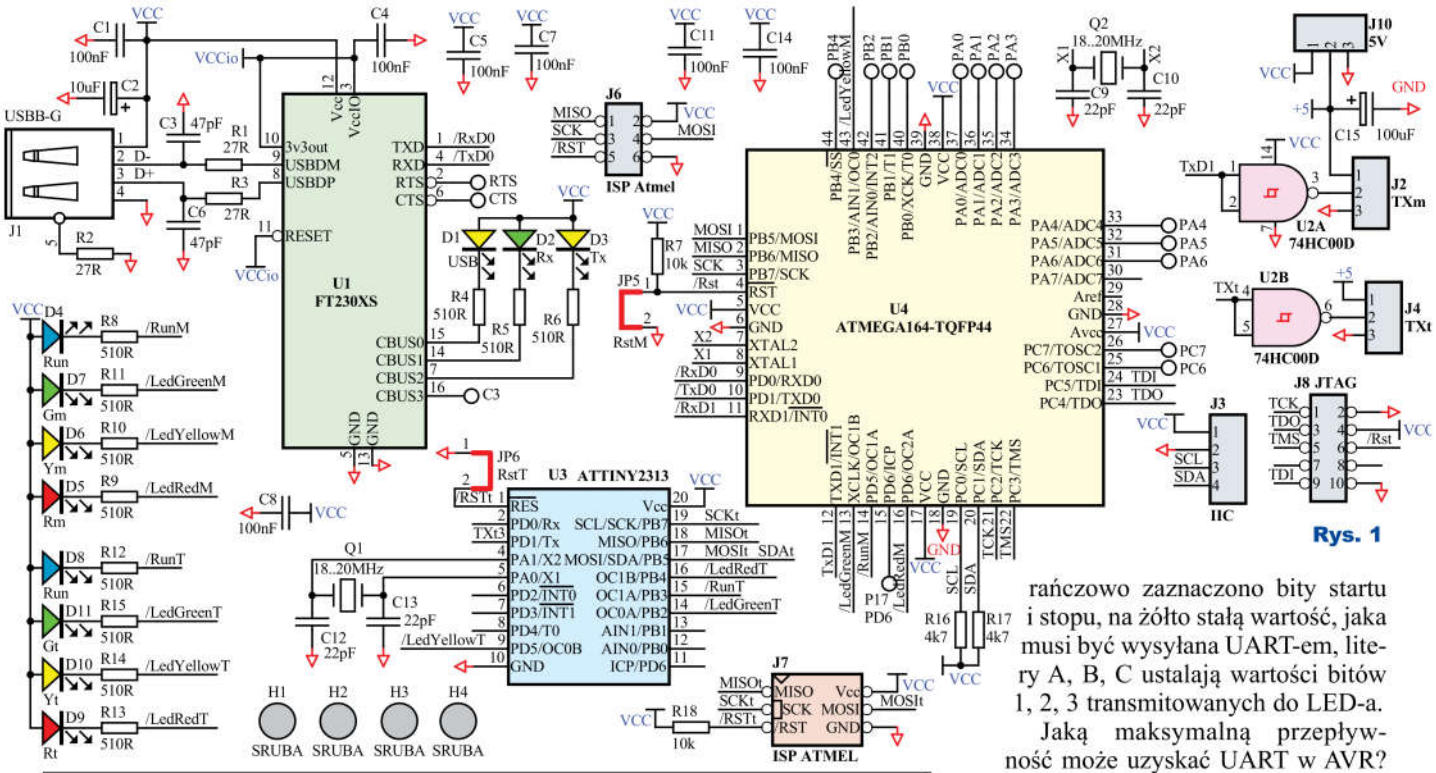
Tabela 1

Wyjątek stanowi ATmega1284 z 16kB RAM. Jeśli użyjemy zewnętrznej pamięci na dane, trzeba liczyć się z nieco dłuższym czasem obsługi przerwania, ponieważ dostęp do zewnętrznej RAM trwa dłużej.

Opis układu

Schemat ideowy pokazany jest na **rysunku 1**. Układ zasilany jest z USB lub zewnętrznego zasilacza 5V. Zwrócę uwagę, że pobór prądu przez LED może być dość duży (do ponad 50mA/LED), zwłaszcza podczas wyświetlania koloru białego i przy większej ich liczbie należy je zasilić z zewnętrznego zasilacza podłączonego do J10. Jeśli decydujemy się za zasilanie z USB, na złączu tym należy zewrzeć piny 2–3. Przy długich taśmach LED wskazane jest doprowadzenie dodatkowego zasilania co 50...100 diod.

Jak widać, strona sprzętowa jest nieskomplikowana, a bardziej wyrafinowany jest program. Diody sterowane są impulsami o okresie 1,25µs ±600ns, co daje przepływność na poziomie 800kb/s – **rysunek 2.** Czasy sygnałów przedstawiono na **rysunku 3.** Transmisja bitu jest podzielona na trzy odcinki czaso-



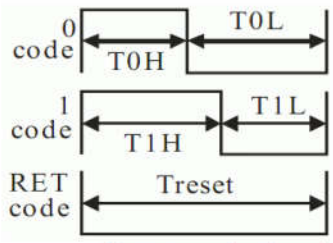
Rys. 1

Wymagana częstotliwość taktująca mikrokontroler wynosi od 18 do 22MHz. Z częstotliwością 20MHz może pracować większość ATtiny i część ATmega. Wolniejsze mikrokontrolery (na przykład ATmega w obudowie 64 lub więcej wyprowadzeń – 16MHz) muszą zostać nieznacznie przetaktowane, lecz nie polecam takiego postępowania, chyba że w celach dydaktycznych. Godny polecenia jest kwarc 18,4321MHz, który jest kwarcem tak zwanym „UART-owym".

Zalety rozwiązania na USART w stosunku do SPI:

1) 8, a nie 9 bajtów danych na jedną diodę.
2) Angażuje jedno, a nie trzy wyprowadzenia mikrokontrolera.
3) Czas pomiędzy przerwaniami nie jest tak krytyczny, jak w SPI, w związku z czym mogą być uruchomione inne przerwania, o ile czas ich obsługi nie jest dłuższy niż około 20µs (licząc od wystąpienia przerwania do chwili ustawienia znacznika I w SREG + jeden rozkaz). W przypadku SPI, gdzie wysyłany bajt może kończyć się jedynką, czas pomiędzy wysyłanymi bajtami nie może przekraczać, według danych producenta, 600ns.

we, co daje przepływność UART-a 2,4MB/s (1,6MB...4,6MB). Ramka UART **7n1** składa się z dziewięciu odcinków czasowych (bit startu, 7 danych, bit stopu), więc w jednej takiej ramce do diody można przesłać 3 bity informacji. Bity startu i stopu mają z góry ustaloną wartość. Dioda LED, a właściwie jej sterownik, przyjmuje impulsy, w których w pierwszym odcinku zawsze jest jedynka, drugi decyduje o wartości przesyłanej informacji, trzeci zawsze jest zerem. Niestety bit startu to zero, a stopu jeden. Sygnał wymagany przez LED-y otrzymujemy…



Rys. 2

po zanegowaniu sygnału UART!

Aby wysłać trzy bity o wartości 0, do UART-a należy wpisać wartość $26 (binarnie 100110). **Rysunek 4** wyjaśnia dlaczego. Na pomarańczowo zaznaczono bity startu i stopu, na żółto stałą wartość, jaka musi być wysyłana UART-em, litery A, B, C ustalają wartości bitów 1, 2, 3 transmitowanych do LED-a.

Data transfer time (TH+TL=1.25µs±600ns)

T0H	0 code, high voltage time	0.4us	±150ns
T1H	1 code, high voltage time	0.8us	±150ns
T0L	0 code, low voltage time	0.85us	±150ns
T1L	1 code, low voltage time	0.45us	±150ns
RES	low voltage time	Above 50µs	

Rys. 3

Rys. 4	Bit startu	7	6	5	4	3	2	1	Bit stopu
Wartość bajtu w UART	0	A	1	0	B	1	0	C	1
Zanegowane bity na wyjściu UART	1	/A	0	1	/B	0	1	/C	0

Jaką maksymalną przepływność może uzyskać UART w AVR? Według noty katalogowej: ***Fosc / 8 * UBRR + 1***.

Dla 20MHz i UBRR=1 otrzymamy 1,25Mb/s, czyli dwa razy za wolno. Okazało się jednak, że można tam wpisać zero, otrzymując 2,5Mb/s – **rysunek 5** (także w Elportalu).

Czy da się transmisję zrealizować na przerwaniach? Jak najbardziej jest to możliwe, nawet bez wstawki asemblerowej. Kody źródłowe programu dostępne są w materiałach dodatkowych w Elportalu. Wynik działania programu obsługi przerwania:

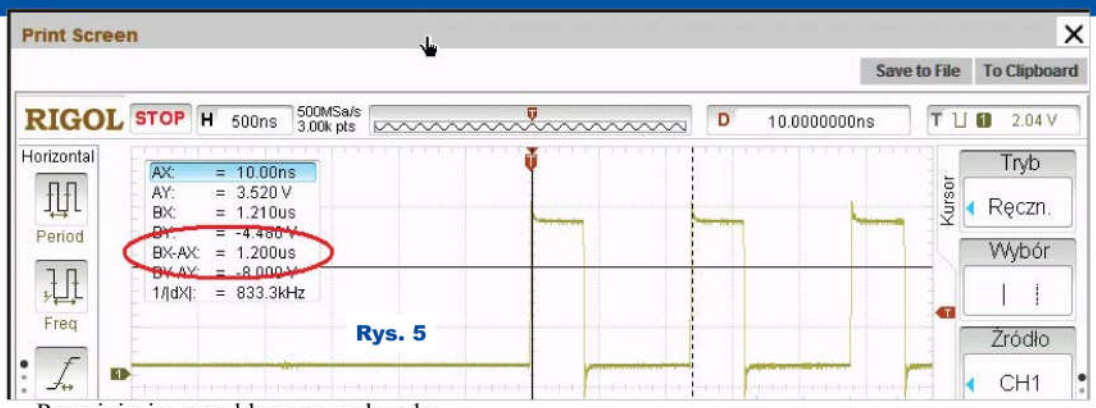
```
SIGNAL(USART1_TX_vect)
{
 if ( pozScr )
  {
  UDR1 = *(scr+pozScr++);
  pozScr &=SCRLEN-1;
}
 if ( pozScr >= SCRLEN ) FL_TransEnd=true;
}
```

pokazano na **rysunku 6**. Żółty przebieg to dane do LED, błękitny efekt działania pętli głównej programu

```
void main(){
 wdg();
 PORTA |= _BV(PA3);
 PORTA &= ~_BV(PA3);
 }
```

która po skompilowaniu wykonuje się w czterech cyklach maszynowych.

Czas obsługi IRQ wynosi 4,44µs. Widać duże obciążenie CPU przerwaniami. Sytuację może poprawić optymalizacja programu (**listing 1**).



Rys. 5

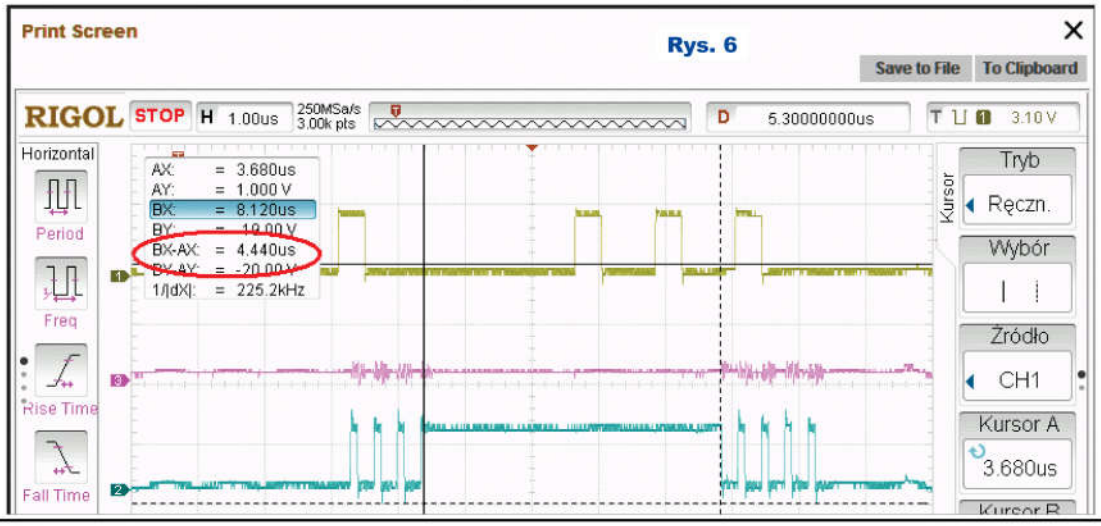
Rozwinięcie asemblerowe wykazało niepotrzebne operacje na R0 i R1 (to jest niestety cechą AVR-GCC), które nie są później używane (żółte tło) i przechowywanie na stosie SREG z użyciem R0 (pomarańczowe tło), którą to operację można wykonać, używając któregoś z rejestrów używanych w IRQ (w przypadku programu demonstracyjnego R18, fragmenty oznaczone zielonym tłem), które z konieczności muszą być odłożone na stosie. Wydawać by się mogło, że oszczędność 7 cykli zegarowych w przerwaniu to niewiele, ale należy pamiętać, że przerwania są wywoływane 267 (800kHz / 3 bity w bajcie) tysięcy razy na sekundę!

Wykorzystując fakt, że AVR UART ma 2-bajtowe FIFO, w jednym przerwaniu można wysyłać dwa znaki, co zmniejszy częstotliwość przerwań dwukrotnie.

Mała wskazówka dotycząca optymalizacji. Aby całego programu nie pisać od początku w asemblerze, wystarczy po skompilowaniu skopiować rozwinięcie asemblerowe (plik *lss). Operację ułatwi „TextPad", który ma opcję zaznaczania kolumnowego. Taki kod wystarczy tylko poprawić i umieścić w swoim programie. Dzięki tym zabiegom czas obsługi przerwania skrócił się do 4,08µs, co widać na **rysunku 7**.

Listing 1

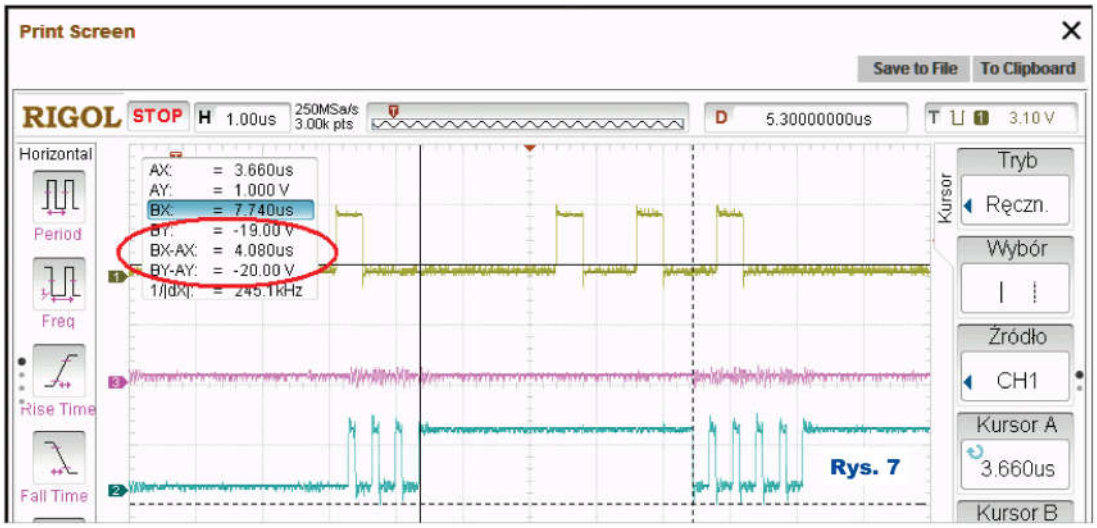
```
ISR( USART1_TX_vect, ISR_NAKED )
{
asm volatile (
// usunięta operacja na nieużywanym  R1 i R0
//        "push    r1                \n\t"
//        "push    r0                \n\t"
//        "in      r0, 0x3f \n\t"    // SREG
//        "push    r0                \n\t"
//        "eor     r1, r1   \n\t"    // Zerowanie R1
          "push    r18               \n\t"
//Dodane: SREG na stos przy użyciu używanego później R18 (+3)
          "in      r18, 0x3f         \n\t"    // SREG
          "push    r18               \n\t"
//Dalej tak jak zrobił kopilator
          "push    r24               \n\t"
          "push    r25               \n\t"
          "push    r30               \n\t"
          "push    r31               \n\t"
          );
//I dotychczasowy kod:
          if ( pozScr )
                  {
                  UDR1 = *(scr+pozScr++);
                  pozScr &=SCRLEN-1;
                  }
          if ( pozScr >= SCRLEN ) FL_TransEnd=true;
asm volatile (
          "pop     r31               \n\t"
          "pop     r30               \n\t"
          "pop     r25               \n\t"
          "pop     r24               \n\t"
//Dodane: SREG przy użyciu R18
          "pop     r18               \n\t"
          "out     0x3f, r18\n\t"
//Dalej normalnie R18
          "pop     r18               \n\t"
//Usunięte operacje na R0 i R1
//        "pop     r0                \n\t"
//        "out     0x3f, r0\n\t"
//        "pop     r0                \n\t"
//        "pop     r1                \n\t"
          );
          reti();;
}
```



Pomiędzy przerwaniami program główny wykonuje około 16 rozkazów. To mało i niewiele da się zrobić w takim czasie. Jeśli jednak weźmiemy pod uwagę to, że transmisja nie musi trwać bez przerwy i pomiędzy paczkami wstawimy pauzę o długości 20ms, co da odświeżanie LED 50Hz, to przy 60 LED-ach otrzymamy:
transmisja do LED: 60 * 1,2us * 24 bity ~1,7ms, czyli zajętość CPU na poziomie 8,5%.

Przykładowe obliczenia obciążenia CPU zawiera **tabela 2**.

Protokół transmisji do LED wymaga sygnału reset. Jest to utrzymanie linii nadawczej w stanie niskim przez co najmniej 50us. Wtedy to dane z zatrzasków w LED są przepisywane do rejestrów PWM. Sygnał reset można wygenerować na kilka sposobów. Po wysłaniu ostatniego bajtu przez UART wstawić instrukcje „delay_us(50)". Nie jest to jednak rozwiązanie eleganckie. Do odliczenia tego czasu można by użyć timera. Ja rozwiązałem to tak, że w przerwaniu od timera 0, wykonywanym co około jedną ms, odliczam timerem programowym czas 50ms. Co 50ms wywołuję funkcję „Refresh()", która ustawia wskaźnik danych na drugi bajt danych do wysłania, a pierwszy wpisuję do rejestru UDR, inicjalizując w ten sposób transmisję w przerwaniach. Aby uniknąć sytuacji, w której po zakończeniu transmisji do LED wystąpi przerwanie od timera i czas



sygnału reset 50us nie zostałby zachowany, po sprawdzeniu flagi „FL_TransEnd" badany jest stan flagi „FL_ref".

```
if ( !TimRefresh-- )
 {
 TimRefresh = 50; // Odświeżanie co xx ms
 if ( FL_TransEnd )
 {  if( !FL_ref )     {FL_ref = true;}
    else { FL_ref = false;
           FL_Refresh = true;
           Refresh();   }
   }
 }
```

Jeśli ma ona wartość 0, to następuje jej ustawienie i dalsze działania na LED zostają przerwane. W następnym przerwaniu, czyli po około 1ms, obie flagi (FL_TransEnd i FL_ref) są ustawione. Wtedy zostaje wykonana funkcja „Refresh()". Takie postępowanie gwarantuje reset o czasie co najmniej 1ms, a jednocześnie w IRQ nie ma żadnych *delay*. Przed wykonaniem „Refresh()" poza zerowaniem flag „FL_TransEnd" i „FL_ref" ustawiana jest flaga „FL_Refresh". Używana jest ona w programie głównym do stwierdzenia faktu odświeżania wyświetlacza. Przy dużej liczbie danych do wysłania i częstym odświeżaniu mogą pojawić się zakłócenia na wyświetlaczu. Można ich uniknąć na dwa sposoby. Użyć podwójnego buforowania, co niestety pochłania drogocenną pamięć, lub przenieść odświeżanie do pętli głównej i wywoływać je po wykonaniu operacji na pamięci „video". W programie o tym, czy ekran jest odświeżany w przerwaniu od timera, czy pętli głównej, decyduje opcja „MAIN_REFRESH".

Inne przerwania: W programie mogą być bez ograniczeń używane przerwania nieblokowane (INTERRUPT czy ISR z atrybutem NOBLOCK). Sytuacja się komplikuje, gdy istnieje konieczność użycia przerwań blokowanych, na przykład odbiorcze USART. Przerwanie takie musi być typu SIGNAL lub ISR (z domyślnym atrybutem BLOCK), bo w przeciwnym wypadku nastąpi przepełnienie stosu. Jeśli przerwanie wykonywało będzie się w czasie krótszym niż około 20us, nie będzie problemu. Dlaczego 20us? Jakkolwiek czas resetu WS2812 to 50us, ale jest to czas, który gwarantuje reset i w wielu przypadkach następuje on, gdy poziom niski trwa tylko 15...25us. Co więc zrobić, jeśli przerwanie trwa dłużej albo tego typu przerwań jest więcej? Istnieje kilka rozwiązań:

1) Najszybciej, jak to możliwe, w przerwaniu wykonać komendę sei(). W przypadku USART-a można to zrobić po odczytaniu rejestru UDR (w praktyce bezpośrednio przed, bo po wykonaniu sei() gwarantowane jest wykonanie co najmniej jednego rozkazu przez mikrokntroler).
2) Jeśli przerwanie dotyczy UART-a, można do sprawdzania jego stanu użyć przerwania od timera. Tu należy wspomnieć, że w przypadku przerwań nadawczych nie ma problemu, bo mogą one być nieblokowane (flaga przerwania jest kasowana po wejściu w IRQ, a nie po operacji na UDR).
3) Użyć ARM lub Xmega, który ma wielopoziomowy system przerwań. W przypadku bardziej zaawansowanych mikrokontrolerów znacznie rozsądniej jest jednak użyć DMA.

Liczba LED	Odświeżanie	Częstotliwość odświeżania	Czas transmisji	Zajętość CPU
400	20ms	50Hz	11,4ms	57%
400	50ms	20Hz	11,4ms	23%
60	50	20Hz	1,7ms	3,4%

Tabela 2

Warstwy: W demonstracji użyto 5 warstw. Każda warstwa to niezależne ekrany nakładane na siebie. Na warstwach mogą znajdować się różne obiekty. W modelu, ze względu na oszczędność RAM), obiekt może przyjąć jedną z 255 barw, a dokładniej jedną z siedmiu, każda w 32 odcieniach plus czerń. Warstwy mają priorytety. Warstwa 0 ma najwyższy i przesłania wszystko, co się pod nią znajduje. Aby przyspieszyć operacje na ekranie związane z priorytetami, najpierw ekran wyświetlacza jest czyszczony, po czym umieszczane są obiekty z warstwy 0. W następnym kroku umieszczane są obiekty z warstw kolejnych pod warunkiem, że nie było wcześniej nic narysowane na ekranie.

Opcje warstw można wykorzystać w grach, gdzie tło (tła) mogą być animowane niezależnie od innych obiektów, które będą przesłaniały tło. Można sobie wyobrazić grę, w której pojazd znajduje się na warstwie 2, tło na „niższych" warstwach 3, 4. Obraz samochodu przesłania więc tło, natomiast na warstwie 1 animowane będą np. tunele, wtedy samochód wjeżdżający w tunel znika pod nim.

Korekcja gamma: W programie wykorzystano korektę gamma, dzięki czemu płynne rozjaśnianie czy ściemnianie LED wygląda bardziej naturalnie. Warto poznać kilka informacji na temat korekty gamma. Aby nie wykonywać czasochłonnych obliczeń on-line przez mikrokontroler, wykonałem je przy użyciu arkusza kalkulacyjnego i tak powstałą tablicę przeniosłem do kodu źródłowego. Współczynniki w takiej tablicy wyznaczane są wg następującego wzoru:

LUT(i) = i ^ (1 / y)

gdzie i – nr elementu tablicy

y – wartość korekty

Y jest stałą, której wartość zależy od charakterystyki LED. Należy tutaj zauważyć, że wzór ten jest prawdziwy dla $0 \le i \le 1$, by wykorzystać ten wzór dla np.: $0 \le i \le 255$, należy go odpowiednio przekształcić do postaci:

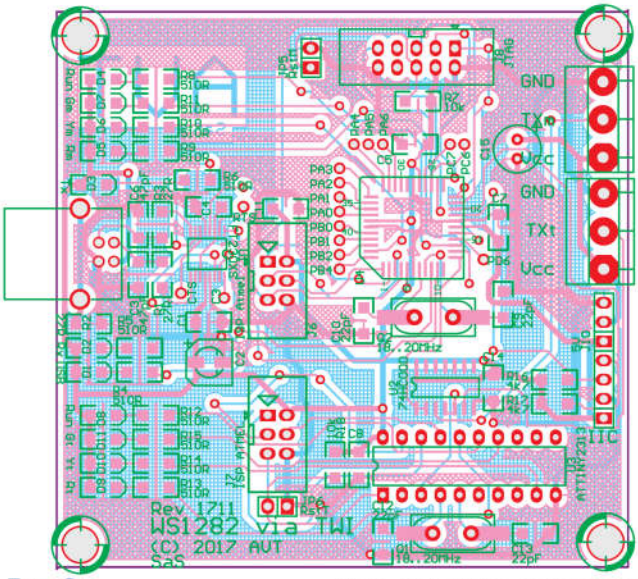
LUT(i) = 255 * (i / 255) ^ (1/y)

Wykresy krzywych, w zależności od wartości współczynnika, przedstawiono

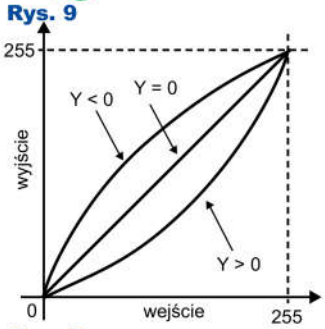
na **rysunku 8** (do przerysowania). Efekt działania korekty można zobaczyć w materiałach dodatkowych lub sprawdzić samemu, ustawiając w opcjach kompilacji „GAMMA_OFF”.

Montaż i uruchomienie

Układ można zmontować na płytce drukowanej, której projekt pokazany jest na **rysunku 9**. Standardowo montujemy układ, zaczynając od elementów najmniejszych, a kończąc na największych. Płytka służyła do testowania programu na procesory mega i tiny, dlatego zawiera dwa mikrokontrolery. Ze względu ma małą ilość pamięci RAM w AVRtiny zrezygnowano z testowania programu na nim, dlatego na płytce nie ma zamontowanych wszystkich elementów. Fotografia wstępna pokazuje model. Osoby niedoświadczone powinny poprosić kogoś o pomoc w zaprogramowaniu procesora.



Rys. 9



Rys. 8

Komendy:

Sterownik LED interpretuje komendy wysyłane po USB:

:xx rr gg bb powoduje, po zatrzymaniu demonstracji i wygaszeniu wyświetlacza, zaświecenie diody nr xx (zakres 1...60) o barwie RGB w zakresie każdego koloru 0...99. Separatorem pomiędzy parametrami może być dowolny znak. Komenda: **@54321** pozwala włączyć lub wyłączyć poszczególne warstwy. „@11111” włączy wszystkie, „@00001” włączy tylko warstwę 1.

SaS
sas@elportal.pl

Wykaz elementów

R1,R2,R3	27Ω – 1206
R4-R6,R8-R15	510Ω – 1206
R7,R18	10kΩ – 1206
C1,C4,C5,C7,C8,C11-C14	100nF - 1206 ceram.
C2	10uF
C3,C6	47pF – 1206
C9,C10,C12,C13	22pF – 1206
C15	100uF
D1,D3,D6	LED SMD 1206 żółta
D2,D7,D11	LED SMD 1206 zielona
D4,D8	LED SMD 1206 niebieska
D5,D9	LED SMD 1206 czerwona
U1	FT230XS – SSOP-16
U2	74HC00D – SO-14
U3	ATTINY2313 – DIP20
U4	ATMEGA164 – TQFP44
Q1,Q2	18-20MHz THT lub SMD
J1	Gniazdo kątowe USB THT
J2,J2	ATK3
J6,J7	ZL201-06G
J8	ZL201-10G