Akcelerator LED WS2812

Diody WS2812 zdobyły niezwykłą popularność. Niestety sterowanie nimi nie jest łatwe. Częstym problemem jest fakt, że w czasie transmisji danych do diod zawieszane są przerwania, a nawet jeśli nie, to obciążenie CPU może okazać się zbyt duże. Nie bez znaczenia jest też potrzeba rezerwacji dużego obszaru RAM na bufor diod, co jest problemem w mikrokontrolerach z małą ilością RAM. Wszystkie te problemy rozwiązuje akcelerator opisany w artykule.

Opisywane urządzenie służy do sterowania diodami WS2812. Do komunikacji z akceleratorem wykorzystano interfejs I2C lub opcjonalnie SPI. Dzięki temu, w czasie transmisji danych, CPU może obsługiwać przerwania, a także możliwe jest, aby sama transmisja odbywała się w przerwaniach. Interfejs SPI umożliwia wysłanie danych szybciej, niż miałoby to miejsce w przypadku bezpośredniej transmisji do LED. Opcjonalne tryby koloru 8 i 16-bit pozwalają zmniejszyć zapotrzebowanie na bufor pamięci RAM do 33 lub 66% w stosunku do bezpośredniego sterowania diodami oraz zmniejszają czas wysyłania danych. Gotowe biblioteki dla Arduino umożliwiają łatwą obsługę akceleratora.

Opis układu

Schemat ideowy pokazany jest na **rysunku 1**. Układ może być zasilany napięciem z przedziału 5...12V doprowadzonym do złącza J3 lub 5V z portu USB (J4). Napięcie 3,3V wymagane przez mikrokontroler zapewnia stabilizator U1. Odbiorem i konwersją danych dla diod zajmuje się niedrogi mikrokotroler U2. W swej strukturze zawiera 16kB pamięci RAM, co pozwala zrealizować bufor danych dla diod, jak też bufory odbiorcze i nadawcze dla interfejsów SPI, I2C, UART, USB, służące do komunikacji z mikrokontroleren. Zawarte w U2 układy DMA umożliwiają odciążenie CPU od zadań związanych z transmisją danych.

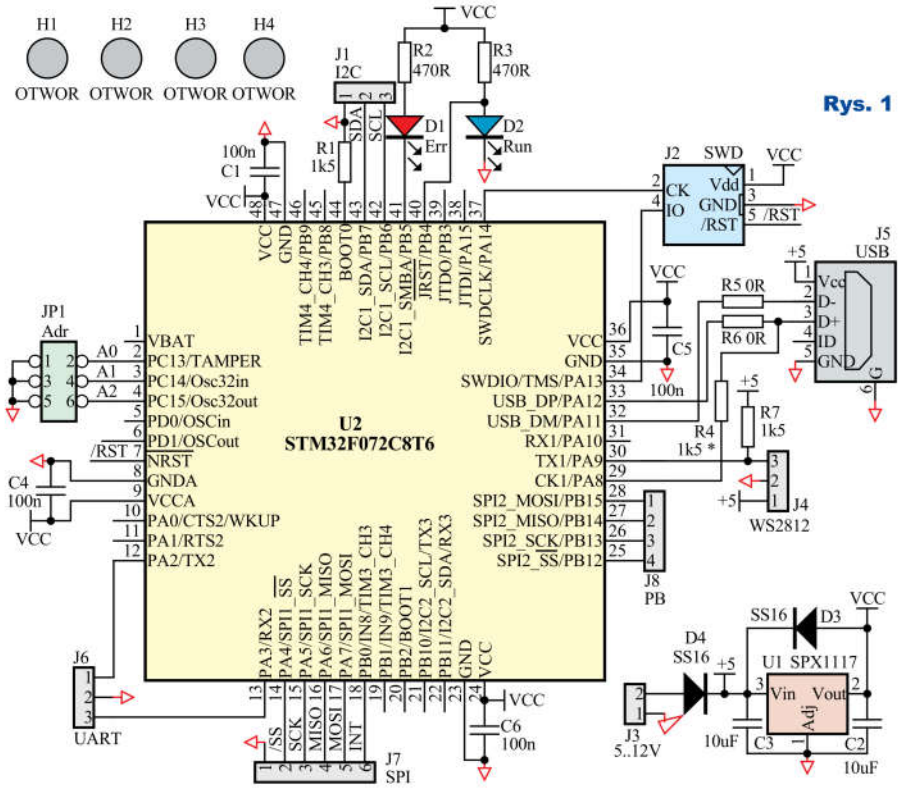
Zanim zostanie omówiona zasada działania akceleratora, należy przypomnieć, co sprawia kłopoty podczas sterowania diodami. Aby wyjaśnić problemy, trzeba zaznajomić się z sposobem komunikacji z diodami. Każda dioda WS2812 zawiera trzy 8-bitowe generatory PWM, po jednym dla każdej składowej koloru.

Charakterystyka:

Maksymalna liczba diod:	640
Liczba akceleratorów	8 (na jednym fizycznym interfejsie)
Paleta barw:	
24	R8G8B8 – 16 mln barw
16	R5G6B5 – 65536 barw
8-bit	R3G3B2 – 255 barw, każda w 63 odcieniach (16 065 barwy)
218 barw	218+16 barw, każda w 63 odcieniach (14 742 barw) opcjonalne 256 barw definiowanych przez użytkownika w 63 odcieniach (16 128 barw)
Interfejs komunikacyjny:	I2C, SPI, USB, UART
Prędkość transmisji:	
SPI	6 (8) MHz
I2C	800kHz
UART	921,6kb/s
USB	12Mb/s

Akcelerator akceptuje napięcia do 5V na wejściach interfejsów komunikacyjnych!

Częstotliwość pracy PWM wynosi ok. 2kHz. Dane o kolorach przesyłane są szeregowo. 24 bity decydują o składowych RGB – **rysunek 2**. Dioda po odebraniu 24 bitów przesyła pozostałe informacje na swoje wyjście Dout. Łącząc diody szeregowo (**rysunek 3**), można wysterować ich niemal dowolną liczbę, a ograniczeniem jest wymagana częstotliwość odświeżania informacji. W czasie transmisji dane są przechowywane w rejestrze diody. Do PWM są przepisywane, gdy wejście przyjmie poziom niski na czas co najmniej 50µs – rysunek 3. Czas 50µs gwarantuje przepisanie informacji do PWM, ale w praktyce może to nastąpić już po około 20...30µs. Diody sterowane są impulsami o okresie 1,25µs ±600ns, co daje przepływność na poziomie 800kb/s – **rysunek 4.** Czasy sygnałów przedstawiono na **rysunku 5.** Nawet na AVR taktowanym zegarem 8MHz uzyskanie wymaganych rygorów czasowych nie jest trudne. Niestety generowanie wymaganych czasów najczęściej odbywa się przez pętle opóźniające.



Rys. 1

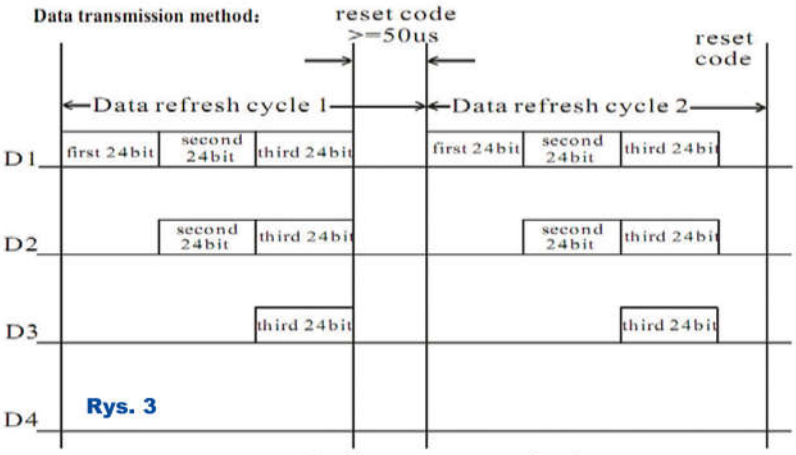
Composition of 24bit data:

Rys. 2

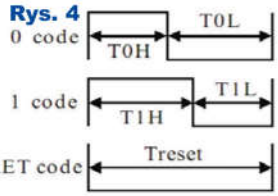
G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0	B7	B6	B5	B4	B3	B2	B1	B0

Note: Follow the order of GRB to sent data and the high bit sent at first.

W konsekwencji w czasie transmisji danych program nie może robić nic innego. W AVR konieczne jest zawieszenie przerwań, bo ewentualne wydłużenie poziomu wysokiego na wejściu Din WS2812 o ponad 600ns (jeden takt CPU w AVR) może powodować błędy w transmisji, natomiast przerwanie trwające ponad 20µs, gdy Din znajduje się w stanie niskim, może zostać zinterpretowane jak sygnał RESET (LATCH) dla WS2812.



Rys. 3



Rys. 4

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	

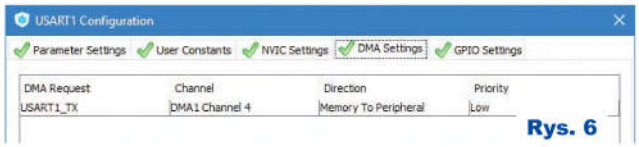
Rys. 5

Problem może rozwiązać transmisja z użyciem UART lub SPI, ale wymaga to przepływności 2,4MB/s, możliwej do uzyskania dla AVR. Jednak skorzystanie z mechanizmu przerwań USART powoduje 80...90% obciążenie CPU, nawet gdy zastosuje się wstawki asemblerowe. Pojawiają się także trudności z obsługą innych przerwań w czasie transmisji do diod spowodowane tym, że większość AVR nie ma wielopoziomowego systemu przerwań, a nadawanie do WS2812 musi mieć najwyższy priorytet. Innym mankamentem jest zapotrzebowanie na pamięć RAM, które wynosi osiem bajtów na diodę, gdy korzystamy z UART, dziewięć, gdy ze SPI, a to oznacza, że na 1000 LED potrzeba 8 lub 9kB RAM. W pamięć 8kB wyposażone są tylko nieliczne AVRmega, a program wymaga też pamięci na stos, stertę, zmienne. Co ważne, w przypadku użycia USART, mikrokontroler musi być taktowany zegarem co najmniej 18MHz. Wszystko to powoduje, że jedynym AVRmega, który jest w stanie wysterować dużą liczbę diod, jest ATmega1284. Wydawać się może, że problem rozwiąże zewnętrzna pamięć danych, którą można podłączyć do niektórych AVR, ale dostęp do takiej pamięci zajmuje dwa razy więcej czasu niż do wewnętrznej SRAM, przez co obciążenie CPU wzrośnie.

Wyjściem z sytuacji jest akcelerator, który odbiera dane dla LED przez interfejs I2C lub SPI. Opcjonalnie przewidziano też możliwość komunikacji przez UART i USB. Wykorzystanie I2C/SPI pozwala na transmisję danych bez blokowania przerwań, a nawet pozwala zrealizować ją na przerwaniach, dzięki czemu CPU może realizować inne zadania w czasie transmisji. Opcjonalne tryby 8- i 16-bitowego kodowania koloru pozwalają zmniejszyć zapotrzebowania na RAM o 66/33% i tyleż samo przyspieszyć transmisję danych. Użyty w akceleratorze mikrokontroler STM32F072C8 ma 16kB RAM, co pozwala umieścić bufory dla 640 LED i danych odbieranych przez I2C/SPI. Liczba 640 jest, a może raczej była znana dzięki Billowi Gatesowi, który na pytanie: „Dlaczego pamięć RAM w PC jest ograniczona do 640kB”, odpowiedział: „640 kB powinno wystarczyć każdemu”.

Nie ma pewności, czy faktycznie Bill Gates wypowiedział te słowa, ale przyznał, że konsultował ograniczenie wielkości pamięci. Fragment oryginalnego nagrania można odsłuchać *https://youtu.be/Cw7AKxzPtkU?t=1060,* ale warto zapoznać się z całym materiałem.

Jak wiadomo z historii komputerów, szybko się okazało, że 640kB nie wystarcza praktycznie nikomu. Aby nie powielić błędu firmy IBM, nie stwierdzam, że 640 LED powinno wystarczyć każdemu. Mało tego, dałem możliwość

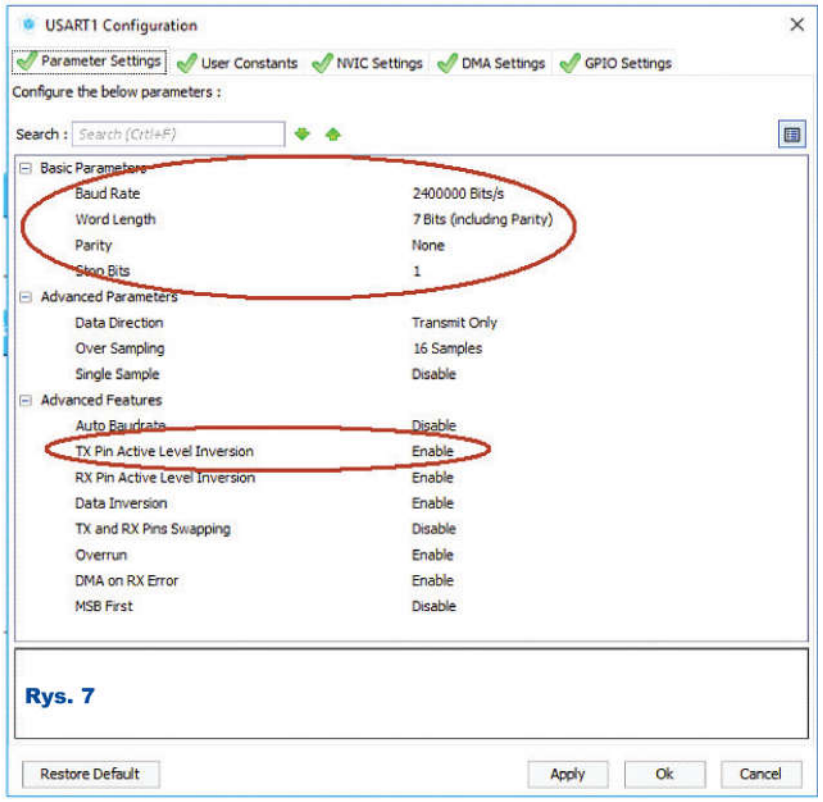


Rys. 6

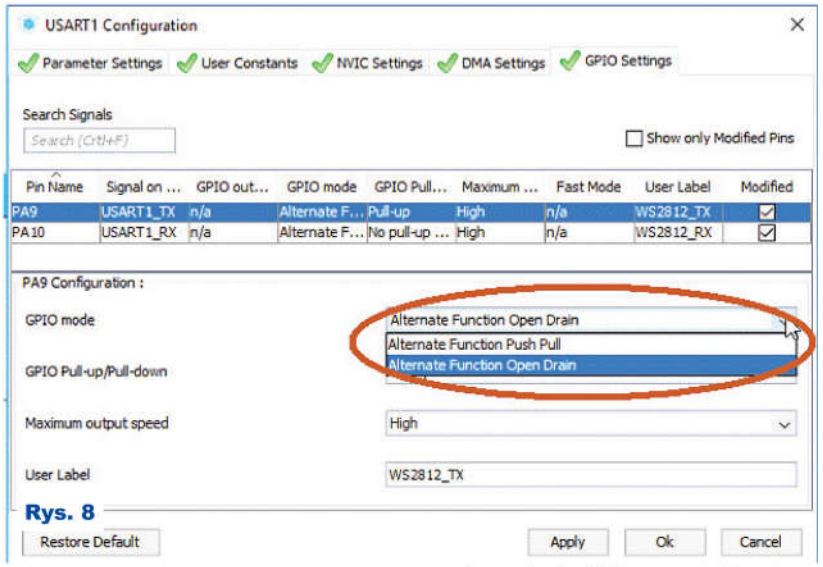
podłączenia ośmiu akceleratorów do jednej magistrali I2C, co pozwala wysterować 5120 diod (ekran o rozdzielczości 128×32 ma 4096 pikseli). W przypadku SPI można podłączyć więcej niż osiem akceleratorów, przy czym każda kolejna ósemka wymaga osobnego sygnału strobu. Biorąc pod uwagę fakt, że przy zegarze SPI 8MHz transmisja danych dla 640 LED trwa około 4,8ms, wysłanie danych do ośmiu akceleratorów zajmie 39ms. Maksymalna częstotliwość odświeżania wyniesie 26Hz, co oznacza, że do obsługi większej liczby LED niż 5120 potrzebny będzie mikrokontroler z większą liczbą interfejsów SPI oraz DMA. Tak jak w przypadku AVRmega, liczba SPI może zostać łatwo zwiększona, ponieważ przeważnie USART może pracować w roli SPI, jednak problem DMA pozostaje. W takiej sytuacji należy sięgnąć po większy mikrokontroler. Xmega mają DMA, ale w porównaniu do ARM ich możliwości nie są oszałamiające, a cena bardzo często wyższa. ARM rozwiąże problem DMA i RAM, ale w takim przypadku po co akcelerator, skoro sam ARM bez problemu wysteruje tysiące LED, a większość czasu spędzi w uśpieniu? Przed zastosowaniem akceleratora warto zastanowić się, czy nie lepiej, zamiast na AVR czy PIC, program napisać na ARM? W Internecie krążą mity na temat skomplikowania ARM. Prawda jest nieco inna: program na ARM pisze się łatwiej niż na AVR, zwłaszcza jak korzysta się z darmowych narzędzi dostarczanych przez producenta.

Wróćmy do głównego tematu. Dane odebrane z I2C lub SPI trafiają do bufora. Z niego, po ewentualnym transkodowaniu, są zapisywane w buforze LED. Transkodowanie potrzebne jest w przypadku kodowania kolorów w 8 lub 16 bitach, jak pokazuje **tabela 1** i **tabela 2**.

Dane dla LED transmitowane są przez UART1 z wykorzystaniem DMA – **rysunek 6.** W przypadku transmisji przez UART z prędkością 2,4Mb/s wymagane jest, aby dane z jego wyjścia były zanegowane. W przypadku STM32F072 można to zrobić programowo – **rysunek 7.** Zaletą STM32 jest możliwość ustawienia pracy wyjścia w trybie otwartego drenu (OD) – **rysunek 8.** Dzięki temu dodanie pojedynczego rezystora (R7 na rysunku 1) rozwiąże problem konwersji sygnału z poziomu 3,3V na 5V. Dzieje się tak za sprawą tego, że wyprowadzenie TX, gdy pracuje w roli wejścia, akceptuje napięcie 5V – **rysunek 9**. Vdd_FT na rysunku nie jest przyłączone do Vdd, jak w przypadku wyprowadzeń nieakceptujących 5V, ale do potencjału 5V (diody Zenera, transila 5V). Dzięki temu rezystor R7, gdy wyjście jest nieaktywne (OD, praca w roli wejścia), wymusza 5V. W przypadku, gdyby wejście nie akceptowało 5V, rezystor wymusiłby napięcie Vdd + około 600mV, co w przypadku zasilania 3,3V



Rys. 7



Rys. 8

Transkodowanie 16-bit na 24 bit: **Tabela 1**

bajt:	HIGH							LOW								
bit słowa:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
bit koloru:	R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0

bajt:	RED								GREEN								BLUE							
bit bajtu:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
bit koloru:	R4	R3	R2	R1	R0	R4	R3	R4	G5	G4	G3	G2	G1	G0	G5	G4	B4	B3	B2	B1	B0	R4	R3	R4

Transkodowanie 8-bit na 24-bit: **Tabela 2**

bit:	7	6	5				4	3	2				1	0		
bit koloru:	R2	R1	R0				G2	G1	G0				B1	B0		

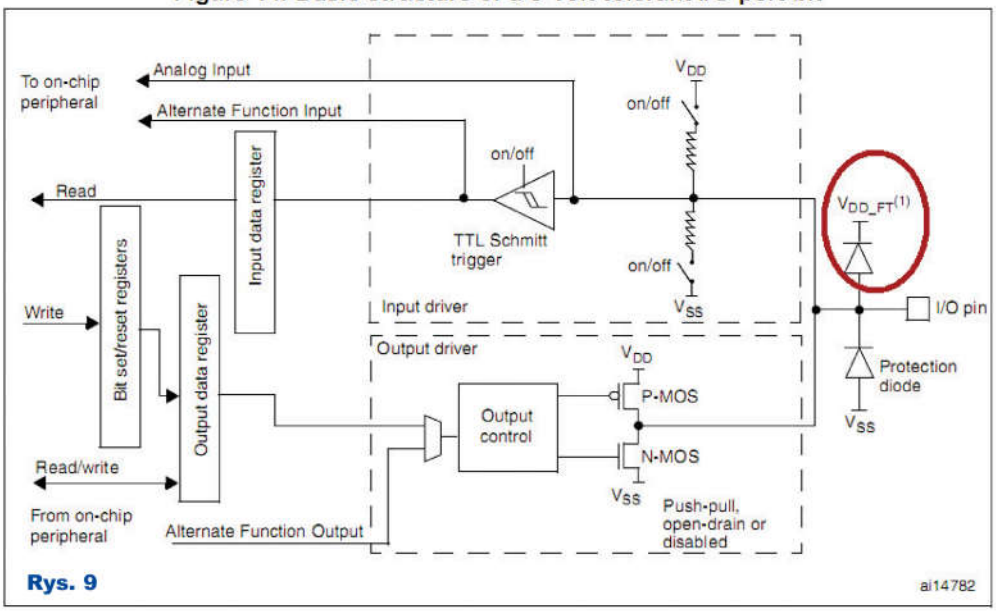
bajt:	RED								GREEN								BLUE							
bit bajtu:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
bit koloru:	R2	R1	R0	R2	R1	R0	R2	R1	G2	G1	G0	G2	G1	G0	G2	G1	B1	B0	B1	B0	B1	R0	B1	B0

daje 3,9V – **rysunek 10**. Napięcie to wystarczyłoby z zapasem do poprawnego wysterowania wejścia WS2812 w stanie wysokim, które wymaga minimum Vdd * 0,7, co przy 5V daje 3,5V.

Montaż i uruchomienie

Układ można zmontować na płytce drukowanej, której projekt pokazany jest na **rysunku 11**. Standardowo montujemy układ, zaczynając od elementów najmniejszych, a kończąc na największych. Rezystor R4 nie jest wymagany. Byłby on konieczny w przypadku serii STM32F1. R5, R6 w przypadku F072 zastępujemy zworą, dla F1xx miałyby one wartość 22Ω. J8 używane jest w celach diagnostycznych i nie ma potrzeby montowania go. Fotografia wstępna oraz **fotografia 1** pokazują model. Aby akcelerator wykonywał powierzone mu zadania, trzeba zaprogramować mikrokontroler programem dostępnym na Elportalu. Służy do tego złącze J2. W czasie pracy programu dioda D2 pulsuje. D1 sygnalizuje błędy – jeśli zaświeci, to należy skontaktować się z autorem. Interfejs I2C mikrokontrolera sterującego akceleratorem przyłączamy do J1. Rezystory podciągające powinny mieć wartość 2,2kΩ. Na fotografii 1 widać tymczasowo przylutowane rezystory podciągające. Jeśli zdecydujemy się na sterowanie akceleratorem za pomocą interfejsu SPI, przyłączamy go do J7. Linii MISO i INT nie trzeba podłączać. Akcelerometr mógłby komunikować się za pośrednictwem UART, ale ze względu na to, że AVR cierpią na ich niedobór, nie zdecydowałem się na implementację UART. Ideę pracy w roli urządzenia 1-Wire także odrzuciłem. Wynika to z faktu, że obsługa 1-Wire dla Arduino zawiesza przerwania i nie obsługuje overdrive.

Figure 14. Basic structure of a 5-Volt tolerant I/O port bit

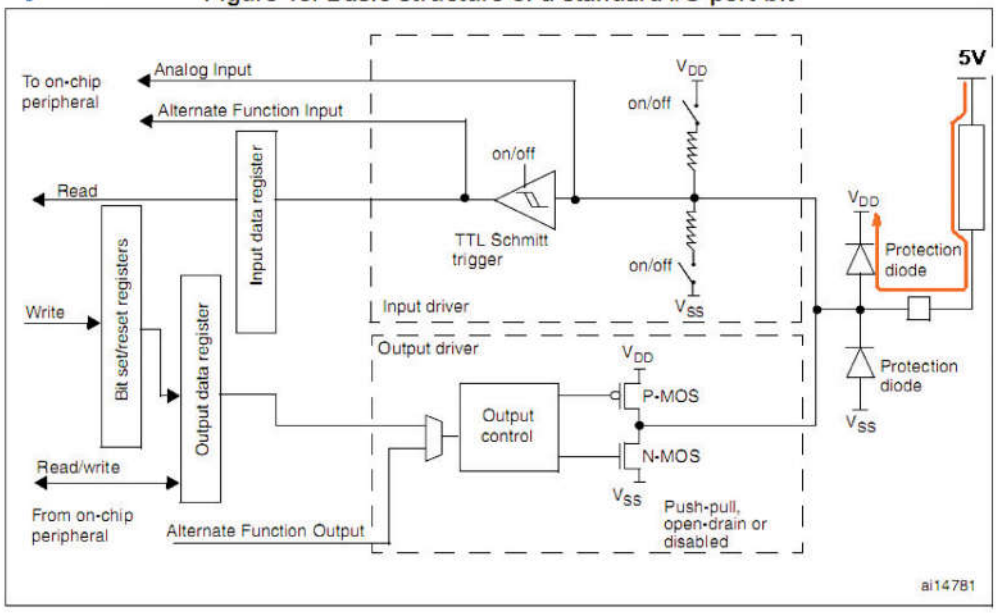


Rys. 9

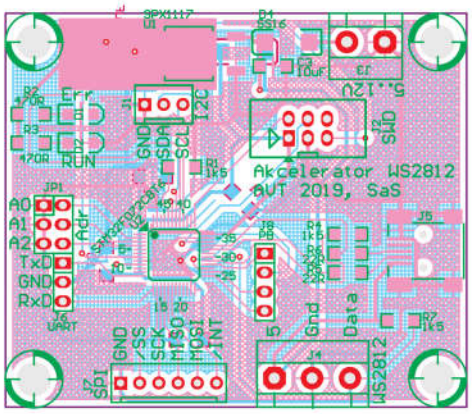
1. V_{DD_FT} is a potential specific to 5-Volt tolerant I/Os, and different from V_{DD}.

Rys. 10

Figure 13. Basic structure of a standard I/O port bit

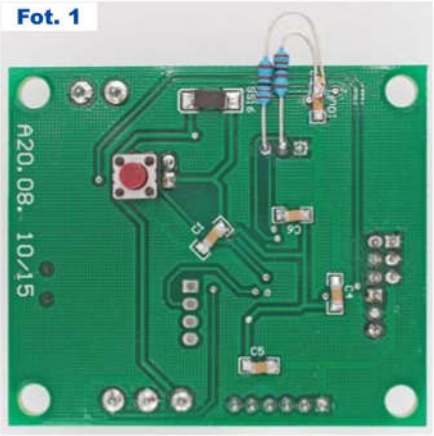


Rys. 11



Konieczne byłoby więc napisanie funkcji obsługi 1-Wire, ale standardowa prędkość, możliwa do zrealizowania bez zawieszania przerwań, jest mała. Przesłanie informacji o 640 LED z wykorzystaniem palety 24-bit zajęłoby prawie sekundę, natomiast na realizację overdrive na przerwaniach AVRmega jest trochę za wolny. Nawet gdyby zawieszać przerwania tylko na czas transmisji bitu (maksymalnie 7µs), czas transmisji dla 640 LED wyniesie ponad 100ms. Rozwiązaniem jest układ DS2482 (master 1-Wire), ale to dodatkowy koszt i zajmuje 2 piny (I2C). W takiej sytuacji nie

Fot. 1



widzę sensu korzystania z dodatkowego układu, jeśli akcelerator jest wyposażony w I2C. Podobna sytuacja ma miejsce w przypadku DS2480, który używa UART. Akcelerator może obsłużyć komunikację z prędkością 921,6kb/s czy 1Mb/s, a DS2480 tylko 115200b/s. Sama idea 1-Wire nie jest do końca pozbawiona sensu. Może być dobrym rozwiązaniem, gdy diod nie będzie dużo, a dodatkowo paleta barw ograniczona do 16 lub 8 bitów. W takiej sytuacji sterowanie 64 LED-ami zajmie około 65ms przy standardowej prędkości i 8ms dla overdrive. Jeśli Czytelnicy będą zainteresowani opcją komunikacji UART, USB lub 1-Wire, proszę o e-maile.

Komunikacja przez USB może rozwiązać problem sterowania diodami z komputera PC czy Raspberry Pi (RPi). Niby RPi może obsługiwać LED WS2812, ale aby komunikacja działała prawidłowo, konieczne jest wyłączenie układu audio.

Gdy akcelerator jest połączony z mikrokontrolerem, do J4 przyłączamy diody WS2812. Do złącza tego można doprowadzić również zasilanie 5V. Zworkami na JP1 ustawia się adres płytki według tabeli 3.

Tabela 3

Zwora A0	Zwora A1	Zwora A2	Adres SPI	Adres I2C
-	-	-	0	0x18 (0x30)
+	-	-	1	0x19 (0x32)
-	+	-	2	0x1A (0x34)
+	+	-	3	0x1B (0x36)
-	-	+	4	0x1C (0x38)
+	-	+	5	0x1D (0x3A)
-	+	+	6	0x1E (0x3C)
+	+	+	7	0x1F (0x3E)

Możliwości akceleratora można sprawdzić programem napisanym dla ArduinoUNO/Mega. Zanim jego i towarzyszącą mu bibliotekę omówię, przybliżę ramki danych wymagane do komunikacji z akceleratorem. Każda ramka składa się z czterech bajtów nagłówka, po którym wysyłane są dane (**tabela 4**).

Tabela 4

Format ramki wysyłanej do akceleratora:

Nr bajtu	Funkcja	Komentarz
0	Konfiguracja	bity 0...2 - adres slave konwertera bit 3 - display: 0-odśwież wyświetlacz, 1-nie odświeżaj bit 4 - korekta gamma: 0-wyłączona, 1-włączona bit 5, 6 - paleta barw: 00 - 24-bit 01 - 16-bit (R5G6B5) 10 - 8-bit (R3G3B2) 11 - 281+16 barw bit 7 - rezerwa (ustawić na 0)
1	Komenda	00 - 24-bit 01 - 16-bit (R5G6B5) 10 - 8-bit (R3G3B2) 11 - 281+16 barw bit 7 - rezerwa (ustawić na 0) 0 - dane do pamięci CGRAM 1...255 - rezerwa
2	AdresH, Bright	bity 0, 1 - starsze (8, 9) bity adresu CGRAM bity 2...7 - jasność świecenia LED dla trybów koloru 8-bit
3	AdresL	bity 0...7 adresu CGRAM
4	bajt 0 danych	
5	bajt 1 danych	
...		
1923	1919 bajt danych	

Liczba danych może być dowolna. Jeśli będzie więcej, niż jest to wymagane, dane zostaną obcięte. Wysłanie danych komendy = 0 od adresu 0 (bajt 3 oraz bity 0, 1 bajtu 2 = 0) czyści CGRAM (pamięć kolorów diod WS2818) przed zapisem danych do niej. Bit DISPLAY decyduje, czy po zakończeniu transferu wysłać dane do LED, czy nie. Może to być użyteczne, jeśli dane do akceleratora są wysyłane w kilku częściach, jak pokazuje **tabela 5**.

Tabela 5

Przykład wysłania danych w czterech częściach po 16 bajtów:

Nr części	Adres	Bit DISPLAY	Uwagi
0	0	1	Czyszczenie pamięci CGRAM
1	16	1	
2	32	1	
3	48	0	Wyświetlenie danych z CGRAM

Format danych koloru:

Nr bajtu danych	paleta 24-bit	paleta 16-bit	paleta 8-bit	218 barw
0	led 0, RED	led 0, LOW	led 0, R3G3B2	led 0, kod koloru
1	led 0, GREEN	led 0, HIGH	led 1, R3G3B2	led 1, kod koloru
2	led 0, BLUE	led 1, LOW	led 2, R3G3B2	led 2, kod koloru
3	led 1, RED	led 1, HIGH	led 3, R3G3B2	led 3, kod koloru
itd				

Tabela 6

Kodowanie 16-bit R5G6B5:

bajt:	HIGH								LOW							
bit bajtu:	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
bit słowa:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
bit koloru:	R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0

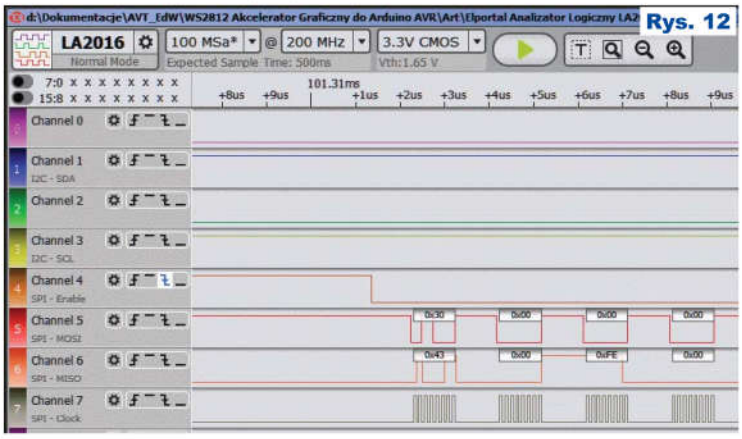
Tabela 7

Kodowanie 8-bit R3G3B2:

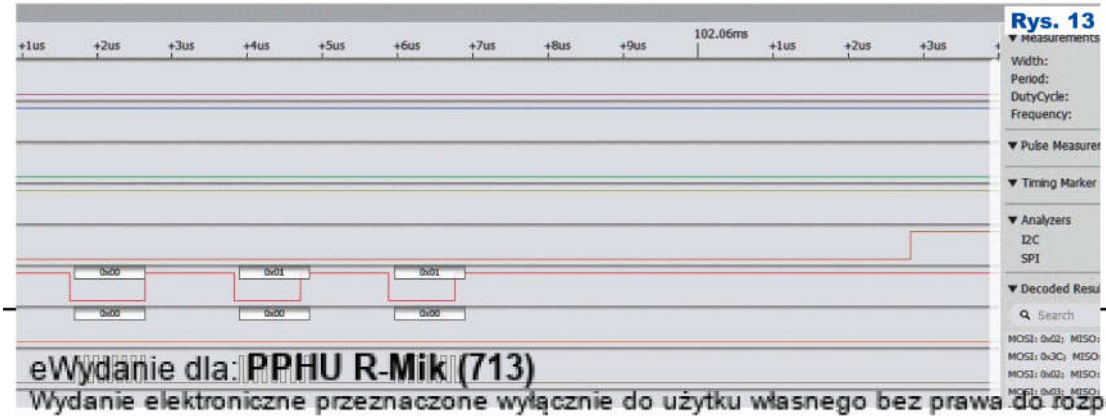
bit:	7	6	5	4	3	2	1	0
bit koloru:	R2	R1	R0	G2	G1	G0	B1	B0

Tabela 8

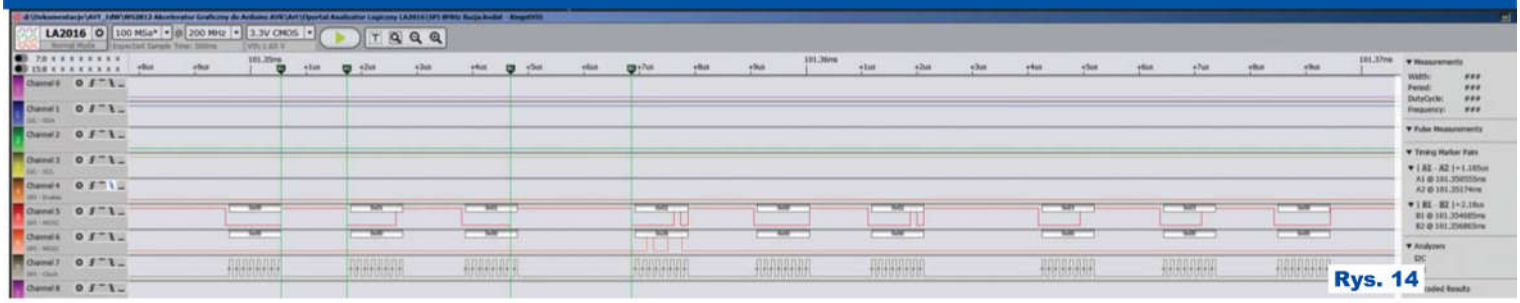
Kolory mogą być kodowane na jeden z czterech sposobów. 24-bitowa paleta niczym nie różni się od naturalnego kodowania dla diod WS2812. Kodowanie 16-bitowe jest identyczne jak w wyświetlaczach graficznych LCD/TFT/OLED. Tryb 8-bit ogranicza liczbę składowych czerwonej i zielonej do trzech bitów (siedem odcieni) niebieskiej do dwóch bitów (trzy odcienie). Aby zwiększyć liczbę stopni intensywności świecenia, bity 3...7 bajtu numer 2 ramki (AdresH) zawierają dane o intensywności świecenia diod. Ustawienie obowiązuje dla całej ramki danych, można więc podzielić dane na kilka części i każdą z nich wyświetlić z inną intensywnością. Paleta 218 barw jest rozszerzona o 16 barw znanych z kart CGA czy terminali VT50/100. Poszczególne barwy i ich składowe można zobaczyć w pliku

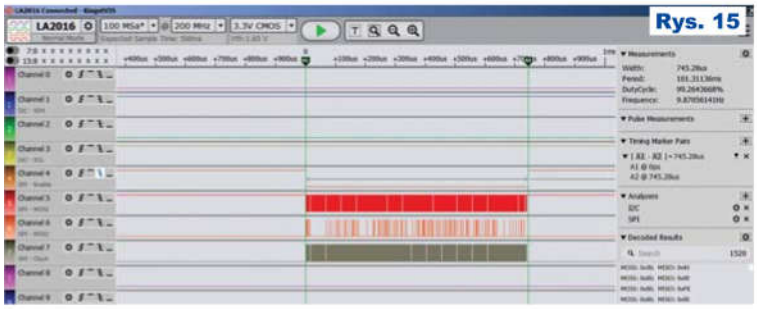


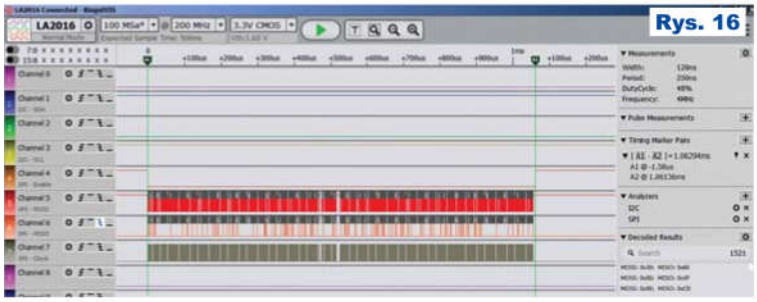
Rys. 12

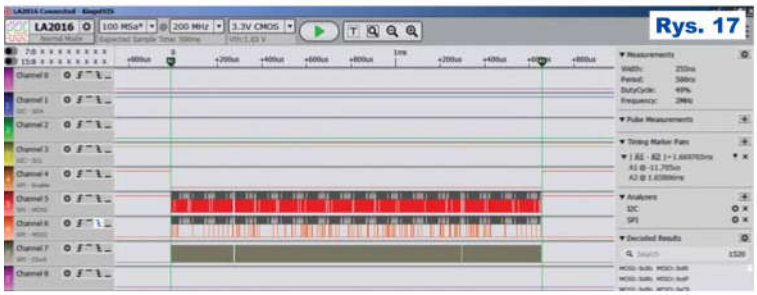


Rys. 13


Rys. 14


Rys. 15


Rys. 16


Rys. 17

„Paleta218.txt" dostępnym w materiałach dodatkowych na Elportalu. Dane wysyłane do akcelerometru mają format, zależny od wybranej palety barw, co jest pokazane w **tabelach 6...8**.

Ramka danych wysyłana po SPI rozpoczyna się zmianą sygnału /SS z wysokiego na niski (**rysunek 12**), a kończy zmianą z niskiego na wysoki – **rysunek 13. Uwaga – oryginalna wersja rysunków – zrzutów z artykułu jest dostępna w Elportalu, wśród materiałów dodatkowych do tego numeru.**

Maksymalna prędkość odbioru danych wynosi 6Mb/s. Mikrokontroler potrafi odbierać dane taktowane zegarem nawet 8MHz, ale pomiędzy bajtami muszą być pauzy. Takie pauzy powstają podczas wysyłania danych na AVR funkcjami Arduino, co widać na **rysunku 14**. Pomiędzy bajtami jest przerwa 1,1µs (markery A1, A2), pomiędzy paczkami trzech bajtów 2,1µs (markery B1, B2). Ta dłuższa pauza jest wywołana pętlą „for" w bibliotece Arduino dla WS2812. W tym przypadku niedoskonałość AVR jest korzystna, ale okazuje się, że wysłanie danych dla 100 LED w formacie 24-bit przy 8 MHz zajmuje 745µs – **rysunek 15**, przy 4MHz 1ms – **rysunek 16**, a nie jakby się wydawało 1,5ms. Dopiero dla zegara 2MHz jest to 1,67ms – **rysunek 17**. Nieproporcjonalny spadek wydajności transferu wynika z tego, że przy 2MHz pauza jest ok. cztery razy krótsza niż czas transmisji bajtu – **rysunek 18.** W przypadku transmisji z prędkością 8MHz pauza trwała 1,1µs (rysunek 14) w sytuacji, gdy bajt był transmitowany w czasie 1µs. Naturalnie można zmodyfikować oprogramowanie tak, aby przerwy były krótsze, ale wymaga to różnych modyfikacji dla każdej prędkości transmisji, a sytuację komplikuje fakt, że podczas jednego obiegu pętli wysyłającej dane może być wysłany jeden, dwa lub trzy bajty, zależnie od tego, na jaką paletę barw zdecydował się użytkownik. Wobec niekorzystnego stosunku pauz do czasu transmisji danych oraz uwzględniając fakt, że im większa prędkość transmisji, tym większa podatność na błędy, zaleca się używanie taktowania SPI zegarem 1MHz, maksymalnie 2MHz.

Dane wysyłane przez I2C muszą być poprzedzone bitem startu oraz adresem płytki – **rysunek 19**.

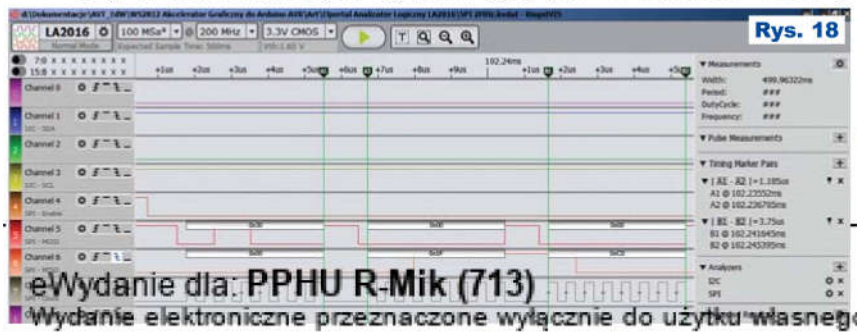
Każdy transfer jest sygnalizowany aktywnym sygnałem ACK. Odbiór kończy warunek stopu – **rysunek 20.** W materiałach dodatkowych na Elportalu znajdują się dane z analizatora. Można je przeglądać programem „LA2016 Kingst VIS" dostępnym bezpłatnie na stronie *http://www.qdkingst.com* w wersji na Windows, Linux oraz MAC.

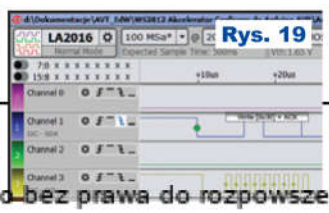
Aby ułatwić wykorzystanie akceleratora, powstała biblioteka EdWws2812 przeznaczona dla ArduinoUNO/Mega i innych płytek z mikrokontrolerami AVRmega. Biblioteka i przykładowy program znajdują się na Elportalu. Funkcja inicjalizująca *EdWws2812(int liczbaLed, byte cfg, byte csPin, uint32_t spd);* przyjmuje następujące parametry:

liczbaLed	Liczba diod WS2812 przyłączonych do akceleratora.
Cfg	Bajt konfiguracji (patrz tablica „Format ramki wysyłanej do akceleratora").
csPin	Numer pin CS dla trybu SPI. Jeśli wartość 0 to tryb komunikacji przez I2C.
spd	Szybkość transmisji: SPI 8, 4, 2, 1MHz, 500, 250, 125kHz. I2C maksymalnie 800kHz, zalecane 400kHz.

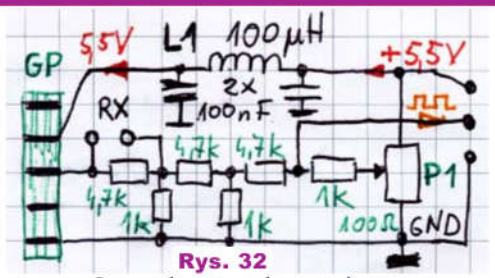
Funkcja *begin()* zwraca kod błędu. Jeśli wartość wynosi 0, to błędu nie ma, jeśli 1 – brak wystarczającego obszaru pamięci RAM dla danych koloru LED.

Funkcja ustawiająca kolor wybranej diody *setLed(int nrDiody, byte r, byte g, byte b)* przyjmuje jej numer oraz składowe RGB w przypadku trybów 8/16 i 24-bit, przy czym w przypadku 8 i 16-bit dane są kodowane (patrz tablice 7 i 8). Dla trybu 218 barw składowe RGB przypisane są na stałe w tablicy (wartości składowych dla poszczególnych numerów kolorów można znaleźć w pliku „Paleta218.txt" na Elportalu). Istnieje możliwość przeniesienia tej tablicy do RAM, a co za tym idzie, wprowadzenia mechanizmów umożliwiających jej modyfikację.

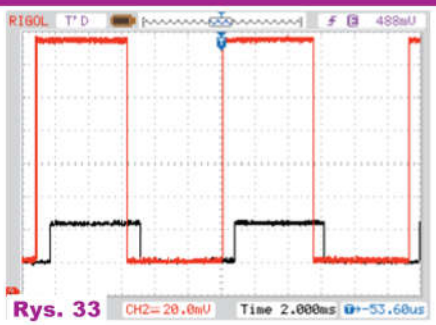

Rys. 18


Rys. 19

Ciąg dalszy na stronie 29



Rys. 32



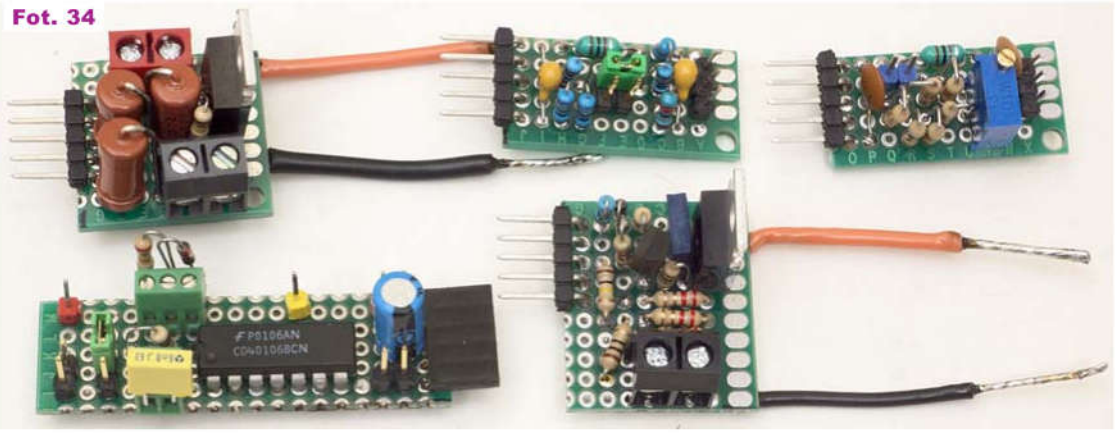
Rys. 33

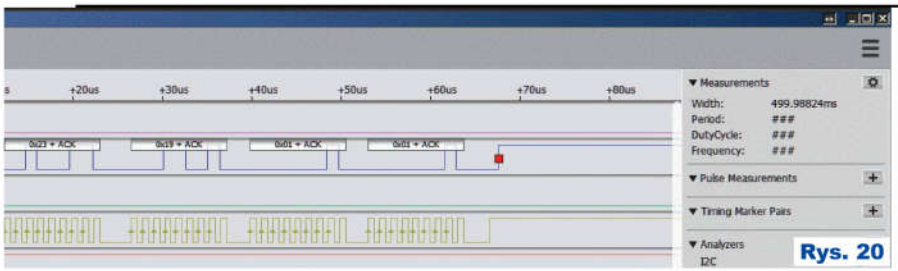
Jego schemat pokazany jest na **rysunku 32**. Aby uniknąć wpływu pojemności montażowych, w układzie nie ma rezystorów o dużej wartości – dlatego zamiast pojedynczego, zastosowany jest tłumik trzystopniowy z rezystorami 4,7kΩ i 1kΩ. Potencjometrem P1 można regulować składową stałą od zera do prawie 5,5V. Przy rezystancji 100 omów w tym 0,5-watowym potencjometrze wydziela się bezpieczne 0,3W mocy strat. Jak pokazuje czarny przebieg na **rysunku 33**, amplituda impulsów na wyjściu wynosi około 24mV, co da skok napięcia na wyjściu zasilacza o około 0,24V. W razie potrzeby skok ten można zwiększyć, włączając dodatkowy rezystor lub zworę między punkty oznaczone RX. Jak pokazuje przebieg czerwony, przy zwarciu tych punktów impulsy na wyjściu mają amplitudę około 135mV, co da na wyjściu zasilacza skoki 1,35V.

Mając narzędzia: uniwersalny generator i cztery opisane przystawki testowe, pokazane na **fotografii 34**, od razu je wykorzystałem. Nie tylko do sprawdzenia wstępnej wersji naszego zasilacza, ale dla porównania także różnych innych posiadanych zasilaczy. Wyniki testów opisane będą w następnym odcinku. Ty, jeśli chcesz, zmontuj takie układy i sam przeprowadź podobne eksperymenty.

Piotr Górecki

Fot. 34





Rys. 20

Wykaz elementów

Rezystory 1206:

R1,R4,R7	(1206) 1,5kΩ
R2,R3	(1206)470Ω
R5 R6	(1206)0Ω

Kondensatory 1206:

C1,C4,C5,C6	(1206)100nF
C2,C3	(1206)10uF
U1	SPX1117 SOT-223
U2	STM32F072C8T6
D1	Dioda led czerwona 1206
D2	Dioda LED niebieska 1206
D3,D4	SS16
JP1	Listwa 3×2 Goldpin 2,54mm
J1,J6	NS25-W3P
J2	T821-1-06-S1
J4	TB-5.0-PP-3P + TB-5.0-PIN
J5	USB-B Mini: MUSB-B5-S-RA-SMT
J7	NS25-W6P
J8	Listwa 4×1 Goldpin 2,54mm
J3	TB-5.0-PP-2P + TB-5.0-PIN

Ciąg dalszy ze strony 25

Jeśli taka opcja zainteresuje Czytelników (proszę o e-maile), program zostanie zmodyfikowany.

Funkcja *clr()* czyści pamięć przeznaczoną na dane o kolorach i nie wymaga argumentów, natomiast *show(byte bright)* transmituje dane do akceleratora. Jedynym jej argumentem jest intensywność świecenia LED, przy czym parametr ten jest ważny tylko dla trybów koloru 8-bit. Przyjmować on może wartość 0...255, przy czym istotne jest tylko sześć najstarszych bitów. Aby wartość zero nie wyłączyła wyświetlacza, dla tej wartości przyjmuje się świecenie z 40% jasnością.

Przykładowy program demonstrujący działanie akceleratora można znaleźć na Elportalu. Komentarze pozwolą na zorientowanie się, jak zmienić interfejs komunikacyjny, liczbę LED czy paletę barw.

Przy okazji konwertera pierwszy raz pisałem bibliotekę i program użytkowy na Arduino. Nie mogę napisać, że praca ta była przyjemnością. Mimo że wspomagałem się oscyloskopem i analizatorem logicznym, brak debuggera odczułem bardzo dotkliwie. Przy okazji wyszły na jaw duże ograniczenia bibliotek dla Arduino, w konsekwencji musiałem dodatkowo napisać własne funkcje obsługi SPI i TWI. W przypadku TWI problemem był bufor nadawczy, ograniczony do 32 bajtów. Jego wielkość można zmienić deklaracją „TWI_BUFFER_LENGTH", ale deklaracja dotyczy zarówno bufora nadawczego, jak i odbiorczego. Aby wysłać dane do 640 LED 24-bit, potrzebny byłby bufor wielkości 1925 bajtów, na co składa się 640 * 3 bajty danych koloru, 4 bajty nagłówka i jeden bajt adresu. Już sam bufor nadawczy dyskwalifikuje z użycia ArduinoUNO (2kB RAM), a i na Mega szkoda zająć 50% pamięci (prawie 4kB bufor nadawczy i odbiorczy z dostępnych 8kB). Nie bez znaczenia jest fakt, że zmienna wskazująca miejsce w buforze ma rozmiar bajtu, co wyklucza skorzystanie z dużego bufora bez modyfikacji kodu biblioteki. Obsługa TWI, którą napisałem na potrzeby WS2812, nie korzysta w ogóle z bufora, a przy okazji jest odporna na błąd statusu 0xF8, który w Arduino doprowadza do zawieszenia programu.

Praca z Arduino bez debuggera, zwłaszcza z wykorzystaniem kiepskiego IDE, to istna katorga, jeżeli porówna się pisanie softu na ARM czy nawet AVR ze wsparciem debuggera.

SaS
sas@elportal.pl