Sterowanie diodami LED RGB za pomocą PCA9633

Pojawienie się na rynku relatywnie tanich diod RGB wywołało zapotrzebowanie na układy sterowania tego typu źródłami światła. W artykule przedstawiamy sposób sterowania wielokolorowymi diodami elektroluminescencyjnymi z wykorzystaniem mikrokontrolerów STM32 i sterownika PCA9633.

Od kilku lat jesteśmy świadkami ciągłe postępującej rewolucji w oświetleniu diodami LED. Znaczny postęp w technologii pozwolił na uzyskanie półprzewodnikowych źródeł światła doskonałej jakości. W chwili obecnej znaczącą barierą jest jeszcze cena diod LED mocy, lecz należy się spodziewać, że w przyszłości staną się one na tyle tanie, iż z powodzeniem zaczną masowo zastępować klasyczne źródła światła. Obok diod białych, które są przeznaczone do celów oświetleniowych, coraz większą popularność zdobywają wielokolorowe diody mocy. Szczególnie interesujące są diody zawierające w jednej obudowie trzy struktury o barwach RGB.

Sterowanie diodami RGB można zrealizować albo bezpośrednio za pomocą timerów mikrokontrolera, albo też z użyciem specjalizowanego kontrolera w postaci układu scalonego. Przykładem takiego scalonego kontrolera diod RGB może być układ PCA9633 produkowany przez firmę NXP.

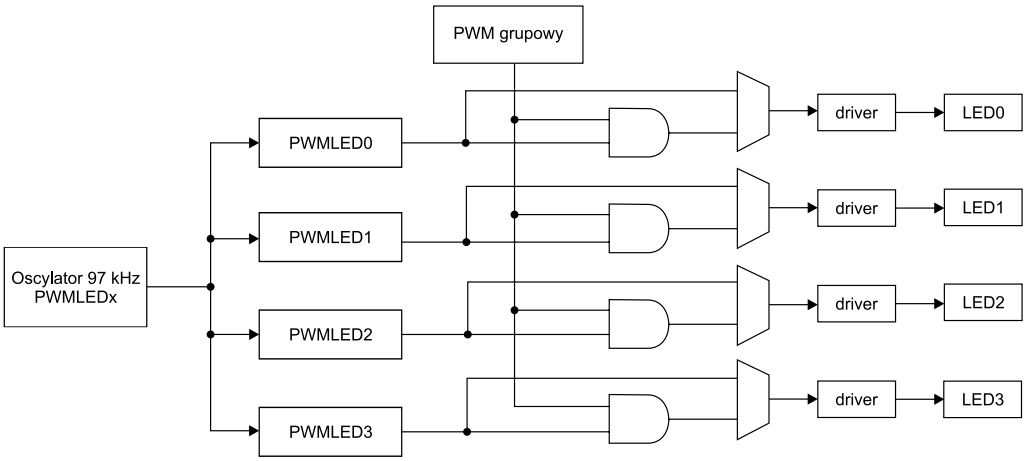
Układ PCA9633

Układ PCA9633 jest sterownikiem diod LED mocy z interfejsem I²C. Uproszczony schemat blokowy układu zamieszczono na **rysunku 1**. Kontroler może sterować pracą do czterech diod LED, przy czym każda indywidualnie ma przypisany swój kanał PWM. Ponadto dostępny jest również piąty generator, który może służyć do jednoczesnego sterowania wszystkimi czterema diodami. Umożliwia to miganie lub ściemnianie/rozjaśnianie wszystkich diod LED jednocześnie. Rozdzielczość generatorów PWM wynosi 8-bitów, przy czym generatory indywidualne pracują z częstotliwością 97 kHz, a generator wspólny (grupowy) z częstotliwością 190 Hz w trybie ściemniania lub z programowaną w zakresie od 24 Hz do około 0,1 Hz, w trybie migania.

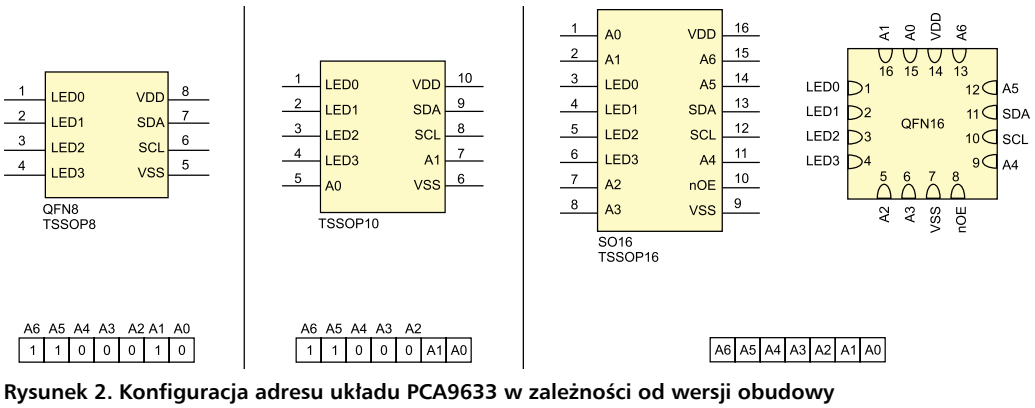
Układ jest produkowany w trzech odmianach różniących się ilością wyprowadzeń. Dostępne są obudowy z 8, 10 lub 16 wyprowadzeniami. Najistotniejszą konsekwencją wynikającą, z ze zróżnicowania liczby wyprowadzeń jest ilość adresów w magistrali I²C. Obudowy 8-wyprowadzeniowe mają ustalony podczas produkcji adres 0x62. Odmiany z 10-wyprowadzeniami mają konfigurowalne dwa najmłodsze bity adresu, pozostałe są, podobnie jak powyżej, ustalone w czasie produkcji, co przedstawiono na **rysunku 2**. Pełną swobodę ustawania siedmiu bitów adresu zapewnia wersja układu zamknięta w obudowie z 16-wyprowadzeniami.

Wyjścia sterujące kontrolera PCA9633 mają dwie możliwe konfiguracje pracy: jako otwarty dren albo „*totem pole*" (*push-pull*). W aplikacjach, gdzie diody są podłączone bezpośrednio do wyjść układu scalonego najlepszym rozwiązaniem jest użycie wyjścia typu otwarty dren. Do wyprowadzeń w tej konfiguracji może wpływać maksymalny prąd 25 mA. Schemat typowego wykorzystania układu PCA9633 w obudowie z 16 wyprowadzeniami w konfiguracji z otwartym drenem przedstawiono na **rysunku 3**, dalsze rozważania w artykule będą dotyczyć takiej właśnie aplikacji.

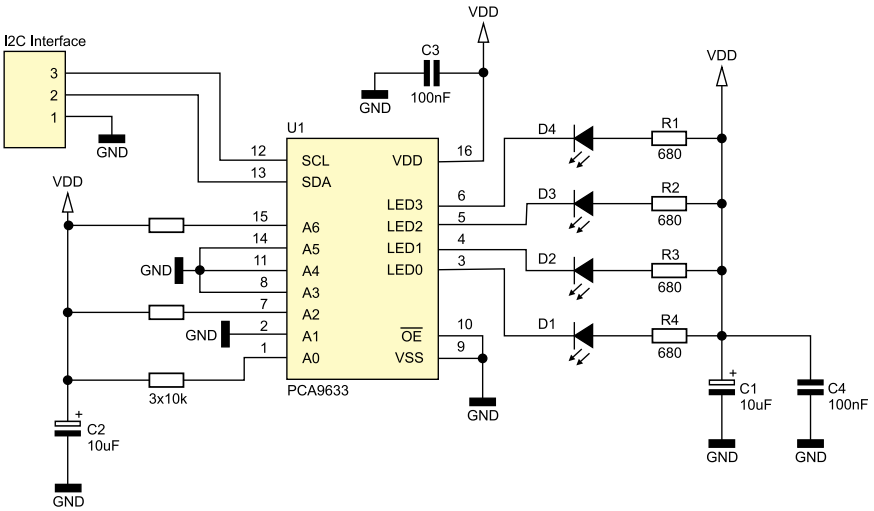
Oprócz podstawowego adresu, programowanego poprzez ustawianie poziomów logicznych na wyprowadzeniach adresowych, układ PCA9633 może mieć jeszcze do pięciu specjalnych adresów. Jeden specjalny



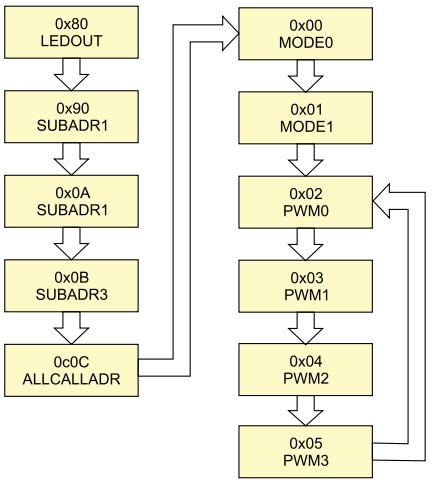
Rysunek 1. Schemat blokowy układu PCA9633



Rysunek 2. Konfiguracja adresu układu PCA9633 w zależności od wersji obudowy



Rysunek 3. Typowa aplikacja kontrolera PCA9633



Rys. 5. Komunikacja z autoinkrementacją adresów

adres zarezerwowany jest do jednoczesnego zerowania ustawień wszystkich układów PCA9633 (jeśli jest ich więcej niż jeden) podłączonych do magistrali I²C. Adres programowego zerowania jest już z góry ustalony, jego wartość wynosi 0x03. Bit zapisu/odczytu przysyłanego bajta adresującego musi być ustawiony na '0', w przeciwnym wypadku sterowniki nie potwierdzą wiadomości i rozkaz zerowania zostanie przez nie odrzucony. Takie programowe zerowanie jest tożsame z przywracaniem ustawień poprzez wyłączenie i włączenie zasilania.

Tabela 1. Mapa rejestrów układu PCA9633

Adres	Nazwa	Opis
0x00	MODE1	Rejestr ustawień
0x01	MODE2	Rejestr ustawień
0x02	PWM0	Wypełnienie kanału LED0
0x03	PWM1	Wypełnienie kanału LED1
0x04	PWM2	Wypełnienie kanału LED2
0x05	PWM3	Wypełnienie kanału LED3
0x06	GRPPWM	Wypełnienie kanału wspólnego
0x07	PRPFREQ	Częstotliwość kanału wspólnego
0x08	LEDOUT	Włącza/Wyłącza poszczególne kanały
0x09	SUBADR1	Adres grupy 1
0x0A	SUBADR2	Adres grupy 2
0x0B	SUBADR3	Adres grupy 3
0x0C	ALLCALLADR	Adresuje wszystkie układy PCA9633

Drugi ze specjalnych adresów pozwala na zaadresowanie jednocześnie wszystkich układów PCA9633 jakie są dostępne. Ta opcja pozwala na ustawienie jednakowych parametrów całej matrycy LED, jeśli tylko diody są sterowane przez przedstawiany kontroler. Domyślnym adresem wspólnym jest 0x70, lecz może być on zmieniony. Predefiniowanie dwóch powyższych adresów sprawia, że maksymalna liczba układów PCA9633 podłączonych do jednego interfejsu I²C zmniejsza się o dwa, do 126 urządzeń.

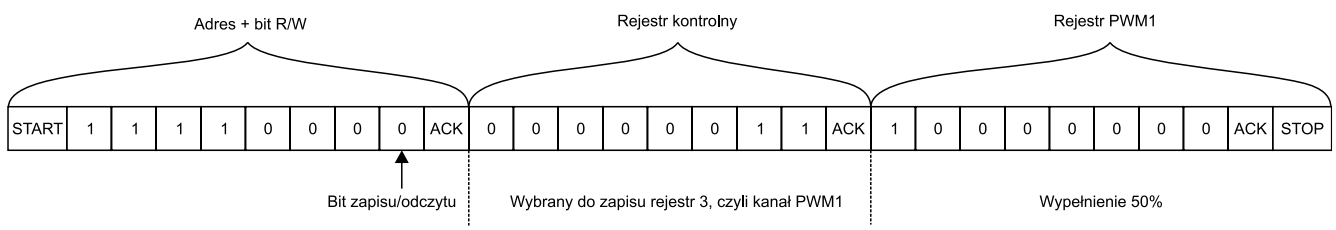
Wspomniano, że kontroler może mieć do 5 adresów specjalnych. Pozostałe 3, to programowane adresy grupowe. Przydatność adresów grupowych daje o sobie znać w sytuacjach, kiedy sterowana jest cała matryca diod LED. Wtedy można podzielić wszystkie kontrolery na grupy i ustawiać jednocześnie parametry pracy przykładowo tylko dla jednej trzeciej matrycy.

Kontroler PCA9633 ma 12 rejestrów ogólnego przeznaczenia oraz jeden rejestr specjalny (kontrolny). Mapę wszystkich rejestrów wraz z krótkim komentarzem zamieszczono w **tabeli 1**, natomiast schemat komunikacji z układem przedstawia **rysunek 4**. Rejestr specjalny jest zapisywany za każdym razem, kiedy wymiana danych jest inicjowana. W najprostszym przypadku wystarczy ustawiać tylko jego młodszy półbajt, co będzie oznaczało zapis lub odczyt tylko jednego zaadresowanego rejestru. Rozszerzenie funkcjonalności umożliwiają trzy najstarsze bity rejestru kontrolnego (AI2, AI1, AI0), odpowiadające za automatyczną inkrementację wewnętrznego licznika adresów.

Opcję autoinkrementacji włącza ustawnie bitu MSB (AI2), natomiast dwa po nim następujące bity dają możliwość kontroli sposobu zwiększania adresów zapisywanych lub odczytywanych rejestrów. Gdy wyzerowane są oba bity AI0 oraz AI1 to inkrementacja przebiega przez wszystkie 12 dostępnych rejestrów. Ta opcja jest najbardziej przydatna podczas inicjalizacji kontrolera. W trakcie normalnej pracy znacznie ciekawsze są pozostałe tryby adresowania automatycznego. Jeśli bity AI0 i AI1 wynoszą odpowiednio 1 i 0 to adresowanie przebiega tylko przez rejestry kontrolujące jasność pojedynczych kanałów. Ustawienie w rejestrze kontrolnym bitu AI0 na 0, a AI1 na 1 będzie skutkowało uzyskaniem automatycznego dostępu do rejestrów odpowiedzialnych za sterowanie piątym, wspólnym dla wszystkich kanałów, generatorem PWM. Jak wspomniano wyżej, generator wspólny umożliwia sterowanie jasnością lub miganiem wszystkich diod LED naraz.

Połączenie obydwu powyższych trybów można uzyskać przez ustawienie obu bitów AI0 i AI1 na 1. Wtedy to automatyczna inkrementacja będzie obejmowała zarówno indywidualne kanały LED jak i generator wspólny.

Wyjaśnienie sposobu komunikacji w trybie automatycznego zwiększania adresów przedstawiono na grafie z **rysunku 5**. Wybrany został tryb z autoinkrementacją tylko rejestrów odpowiedzialnych za przechowywanie współczynników wypełnienia dla poszczególnych diod LED (rejestry PWM0 do PWM4). Jako początkowy ustalono rejestr LEDOUT o adresie 0x08, który umożliwia włączanie lub wyłączanie oddzielnie każde-



Rysunek 4. Fragment komunikacji z kontrolerem PCA9633

go kanału. Podczas pierwszej iteracji dostępne do zapisu/odczytu są wszystkie rejestry układu PCA9633. Kolejne iteracje przebiegają już tylko po adresach rejestrów PWMx. Kontroler analogicznie zachowuje się dla pozostałych trybów automatycznego zwiększania adresów rejestrów. Szczegóły konfiguracji i sposobów sterowania kontrolerem PCA9633 przedstawiono poniżej przy okazji omawiania przykładowej aplikacji.

Magistrala I²C

Komunikacja przez interfejs I²C wymaga użycia dwóch linii sygnałowych: zegarowej SCL i danych SDA. Układy scalone przeznaczone do pracy z magistralą I²C mają dwukierunkowe wyprowadzenia typu otwarty dren. Z tego powodu, aby uzyskać stan wysoki na liniach sygnałowych należy zastosować rezystory podciągające do plusa (*pull-up*) zasilania według schematu z **rysunku 6**. Wartości rezystorów podciągających wynikają wprost z parametrów zastosowanych układów scalonych oraz, co bardziej istotne, z parametrów samej magistrali, to jest z jej pojemności i indukcyjności. W praktyce w standardowych przypadkach, przy długości linii sygnałowych, które nie przekraczają kilku centymetrów wystarczające są rezystory podciągające o wartości 4,7 kΩ.

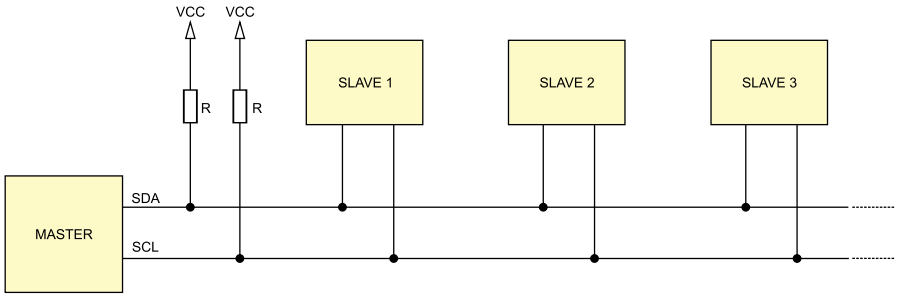
Podstawową jednostką informacji przesyłanej po magistrali I²C jest jeden bajt, a po przesłaniu każdych ośmiu bitów wymagane jest potwierdzenie ACK (z ang. Acknowledgment) wystawiane przez układ odbiorczy w trakcie dziewiątego cyklu zegarowego. W tym czasie linia danych SDA musi zostać zwolniona przez układ nadający, tak, aby odbiornik mógł wymusić na niej stan niski odpowiadający potwierdzeniu. Wygląd kompletnej ramki danych przedstawiono na **rysunku 7**.

Początek wymiany danych jest zawsze inicjowany przez układ nadrzędny (*master*) poprzez wystawienie na magistralę sygnału START. Znacznik ten jest zdefiniowany przez standard jako zmiana stanu na linii danych SDA z wysokiego na niski, podczas gdy na linii zegarowej SCL panuje stan wysoki, zaznaczono go na rysunku 7. Następnie jako pierwszy wysyłany jest najbardziej znaczący bit MSB.

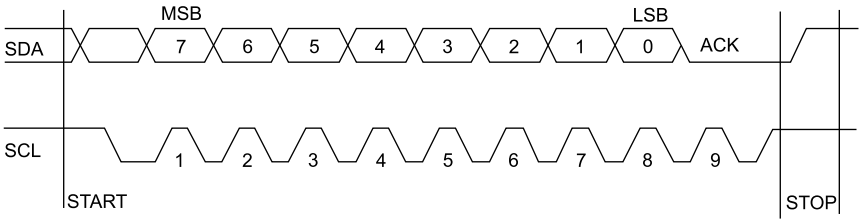
Transmisja bloku danych (bajtów) musi być zawsze zakończona sygnałem STOP, który jest zboczem narastającym na linii SDA, gdy na SCL jest logiczna '1'. Pomiędzy znacznikami START i STOP może zostać wysłana dowolna liczba bajtów.

Kodowanie pojedynczego bitu też jest ściśle zdefiniowane. Linia danych SDA podczas przesyłania bitu nie może się zmieniać wtedy, kiedy wyprowadzeniach zegarowych SCL panuje stan wysoki. Wyrażając to innymi słowami: stan SDA może ulegać zmianie tylko w chwilach, gdy SCL jest w stanie niskim. Sposób kodowania logicznych '0' i '1' zilustrowano na **rysunku 8**. Intuicyjnie czujemy, że skoro w trakcie komunikacji występuje jedno urządzenie nadrzędne to musi istnieć mechanizm odróżnienia, czy master chce z innego układu dane odczytać, czy do niego zapisać. Rozróżnienie, jaką akcję chce podjąć master odbywa się na etapie przesyłania adresu. W podstawowej wersji I²C adres jest 7-bitowy, a wolny bit R/W w przesyłanym bajcie jest wykorzystywany do zaznaczania, czy dane będą zapisywane, czy odczytywane. Wartość najmłodszego bitu w ramce adresowej wynosząca 0 oznacza, że dane mają być zapisywane do adresowanego urządzenia. W przeciwnym wypadku, czyli jeśli bit R/W będzie ustawiony na 1, to master będzie próbował dane odczytywać z urządzenia podrzędnego.

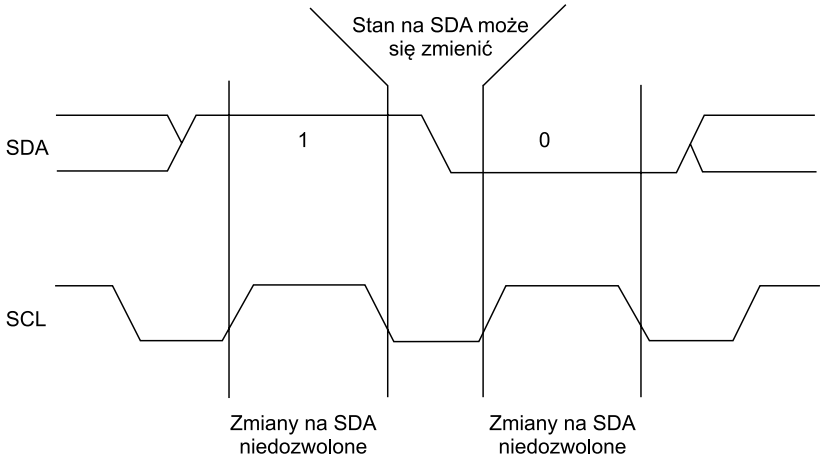
Obok standardowego trybu adresowania istnieje również możliwość zwiększenia puli dostępnych adresów przez użycie rozszerzonego, 10-bitowego, trybu adresowania. Na **rysunku 9** przedstawiono ramki adresujące w trybie rozszerzonym. Powiększenie maksymalnej liczby dołączonych do magistrali I²C urządzeń wymaga przesłania dwóch ramek adresowych. Pierwsza (tzw. nagłówek) zawiera dwa najstarsze bity 10-bitowego adresu urządzenia oraz na najmłodszej pozycji



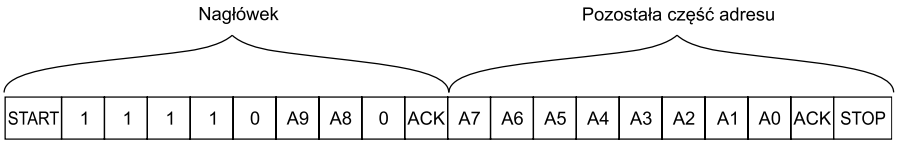
Rysunek 6. Magistrala I²C



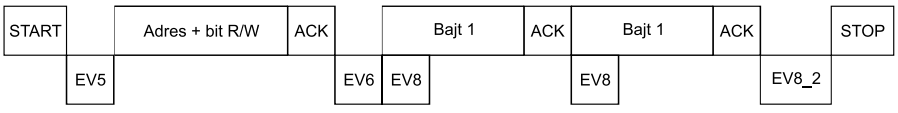
Rysunek 7. Ramka danych w standardzie I²C



Rysunek 8. Kodowanie wartości logicznych



Rysunek 9. Adresowanie w trybie 10-bitowym



Rysunek 10. Fragment komunikacji z zaznaczonymi zdarzeniami *EV*

Listing 1. Konfiguracja kontrolera I²C
```
void I2C_Configuration(void)
{
    I2C_InitTypeDef I2C_InitStructure;

    I2C_Cmd(I2C1, ENABLE);

    /* Tryb I2C, dostepne rowniez dwa tryby SMBus */
    I2C_InitStructure.I2C_Mode = I2C_Mode_I2C;

    /* Wypelnienie, parametr istotny w trybie szybkim (fast) */
    I2C_InitStructure.I2C_DutyCycle = I2C_DutyCycle_2;

    /* Wlaczenie potwierdzania odebrania bajta */
    I2C_InitStructure.I2C_Ack = I2C_Ack_Enable;

    /* SCL = 100 kHz */
    I2C_InitStructure.I2C_ClockSpeed = 100000;

    /* Adres ukladu STM32 w I2C, tutaj nie ma wiekszego znaczenia */
    I2C_InitStructure.I2C_OwnAddress1 = 0x28;

    /* Adresowanie 7-bitowe */
    I2C_InitStructure.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;

    I2C_Init(I2C1, &I2C_InitStructure);
}
```

bit zapisu/odczytu. Pozostałe bity są wyznaczone przez standard i mają stałe wartości. Ponieważ bit R/W jest przesyłany w pierwszej ramce, wszystkie osiem bitów drugiej może być wykorzystane do zakodowania adresu urządzenia.

Kontroler magistrali szeregowej I²C w STM32

Mikrokontroler STM32F107 jest wyposażony w jeden kontroler magistrali szeregowej I²C, który może pracować z częstotliwością do 400 kHz w trybie szybkim (*fast*) lub do 100 kHz w trybie standardowym. Urządzenie jest sterowane przez 13 rejestrów.

Sterownik magistrali I²C wbudowany w mikrokontrolery STM32 może zapewniać sprzętowe wsparcie dla protokołu komunikacyjnego SMBus (*System Management Bus*), który został opracowany przez firmę Intel na potrzeby wymiany małej ilości informacji z urządzeniami zasilającymi. Ze względu na powagę zadań, do których SMBus została stworzona, standard ściśle definiuje zachowanie się magistrali. Przykładowo, linii zegarowej została ograniczona do maksymalnie 100 kHz, wprowadzono ograniczenia czasowe, a i poziomy logiczne magistrali są zdefiniowane bezwzględnie, a nie jak to jest w przypadku I²C w odniesieniu do napięcia zasilania. Dodatkowe informacje na temat magistrali SMBus można znaleźć na internetowej stronie domowej standardu: www.smbus.org.

Kontroler I²C, poza oczywistymi trybami komunikacji przez odpytywanie i obsługę przerwań, udostępnia wszelkie dobrodziejstwa płynące z zastosowania DMA do przesyłania większej porcji danych. Użycie przerwań lub DMA w przypadku mikrokontrolerów z rodziny *connectivity line* (m. in. STM32F107) ma szczególne znaczenie, ponieważ producent, firma STMicroelektronics, nie ustrzegł się przed błędami w budowie kontrolera I²C. Dostępna errata informuje, że poprawna komunikacja wymaga podjęcia konkretnych kroków, które mają na celu silniejszą kontrolę nad pracą urządzenia peryferyjnego. Taką zwiększoną kontrolę właśnie zapewnia użycie przerwań lub kontrolera DMA. Z tego powodu w dalszej części artykułu przedstawione będą aplikacje korzystające z wymienionych wyżej sposobów na komunikację, a pominięty zostanie tryb wymiany danych przez odpytywanie.

Listing 2. Definicje zdarzeń *EV*
```
/* EV1 */
#define  I2C_EVENT_SLAVE_TRANSMITTER_ADDRESS_MATCHED       ((uint32_t)0x00060082) /* TRA, BUSY, TXE and ADDR */
#define  I2C_EVENT_SLAVE_RECEIVER_ADDRESS_MATCHED          ((uint32_t)0x00020002) /* BUSY and ADDR */
#define  I2C_EVENT_SLAVE_TRANSMITTER_SECONDADDRESS_MATCHED ((uint32_t)0x00860080)  /* DUALF, TRA, BUSY and TXE */
#define  I2C_EVENT_SLAVE_RECEIVER_SECONDADDRESS_MATCHED    ((uint32_t)0x00820000)  /* DUALF and BUSY */
#define  I2C_EVENT_SLAVE_GENERALCALLADDRESS_MATCHED        ((uint32_t)0x00120000)  /* GENCALL and BUSY */

/* EV2 */
#define  I2C_EVENT_SLAVE_BYTE_RECEIVED               ((uint32_t)0x00020040)  /* BUSY and RXNE */

/* EV3 */
#define  I2C_EVENT_SLAVE_BYTE_TRANSMITTED            ((uint32_t)0x00060084)  /* TRA, BUSY, TXE and BTF */

/* EV4 */
#define  I2C_EVENT_SLAVE_STOP_DETECTED               ((uint32_t)0x00000010)  /* STOPF */

/* EV5 */
#define  I2C_EVENT_MASTER_MODE_SELECT                ((uint32_t)0x00030001)  /* BUSY, MSL and SB */

/* EV6 */
#define  I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED  ((uint32_t)0x00070082)  /* BUSY, MSL, ADDR, TXE and TRA */
#define  I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED     ((uint32_t)0x00030002)  /* BUSY, MSL and ADDR */

/* EV7 */
#define  I2C_EVENT_MASTER_BYTE_RECEIVED              ((uint32_t)0x00030040)  /* BUSY, MSL and RXNE */

/* EV8 */
#define I2C_EVENT_MASTER_BYTE_TRANSMITTING           ((uint32_t)0x00070080) /* TRA, BUSY, MSL, TXE */

/* EV8_2 */
#define  I2C_EVENT_MASTER_BYTE_TRANSMITTED           ((uint32_t)0x00070084)  /* TRA, BUSY, MSL, TXE and BTF */

/* EV9 */
#define  I2C_EVENT_MASTER_MODE_ADDRESS10             ((uint32_t)0x00030008)  /* BUSY, MSL and ADD10 */
```

Listing 3. Kod ustawiający parametry pracy kontrolera przerwań NVIC
```
void NVIC_Configuration(void)
{
    /* 4 grupa priorytetow */
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4);

    /* Wlaczenie przerwan od zdarzen EV*/
    NVIC_SetPriority(I2C1_EV_IRQn, 0x00);
    NVIC_EnableIRQ(I2C1_EV_IRQn);

    /* Wlaczenie przerwan od bledow ER*/
    NVIC_SetPriority(I2C1_ER_IRQn, 0x01);
    NVIC_EnableIRQ(I2C1_ER_IRQn);
}
```

Komunikacja przez I²C z użyciem przerwań

Jak wspomniano, kontroler I²C wbudowany w układy STM32 nie jest pozbawiony wad, dlatego zalecanym przez producenta sposobem realizacji komunikacji jest użycie

przerwań lub kontrolera DMA. Opis pierwszego sposobu zamieszczono poniżej.

Przed przystąpieniem do ustawiania parametrów pracy kontrolera I²C należy włączyć dla niego sygnał taktowania oraz skonfigurować wyprowadzenia PB6 oraz PB7 mikrokontrolera jako otwarty dren do pełnienia funkcji alternatywnych. Wyjście zegarowe SCL (PB6) i linia danych (PB7) powinny być podłączone do wyprowadzeń odpowiednio numer 12 i 13 kontrolera PCA9633, jeśli używany jest układ zamknięty w obudowie 16-wyprowadzeniowej.

Parametry, które należy ustawić dla kontrolera I²C wbudowanego w STM32 to przede wszystkim: częstotliwość linii zegarowej (a więc prędkość transmisji), adres, tryb adresowania oraz włączenie lub wyłączenie potwierdzenia odebrania bajta, w ramce adresowej będzie to dziewiąty bit ACK. Konfigurację można przeprowadzić za pomocą biblioteki API, wtedy kod może wyglądać podobnie do tego zamieszczonego na **listingu 1**. Oczywiście powinna również zostać wybrana, odpowiednia do stosowanego rezonatora, właściwa częstotliwość taktowania magistral oraz rdzenia. Aplikacja była testowana z użyciem rezonatora do zastosowań audio, a więc o częstotliwości 14,7456 MHz. Niestety przy takim rezonatorze uzyskana częstotliwość dla rdzenia wynosi 71,88 MHz (zamiast maksymalnej 72 MHz), a tym samym częstotliwość wewnętrznej magistrali, do której podłączony jest kontroler I²C wynosi 35,94 MHz (zamiast maksymalnej 36 MHz), przez co nie jest możliwe uzyskanie „okrągłych" wartość zegara (np. 100 kHz). Nie ma to jednak większego znaczenia z punktu widzenia wymiany danych, komunikacja może odbywać się całkowicie poprawnie.

Listing 4. Funkcja obsługi przerwań od zdarzeń *EV* na magistrali I²C

```
void I2C1_EV_IRQHandler(void)
{
    uint32_t event = ( (uint32_t)(I2C1->SR1) | (uint32_t)(I2C1->SR2)<<16 ) & 0x00FFFFFF;

    switch (event)
    {
        /* Zdarzenie EV5 */
        case I2C_EVENT_MASTER_MODE_SELECT:

            /* Wyslanie adresu w trybie zapisujacego urzadzenia nadrzednego */
            I2C_Send7bitAddress(I2C1, SlaveAddress, I2C_Direction_Transmitter);
        break;

        /* Zdarzenie EV6 oraz pierwsze ze zdarzen EV8 */
        case I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED:

            /* Wyslanie pierwszego bajta */
            I2C_SendData(I2C1, Buffer_Tx[Tx_Idx++]);
        break;

        /* Zdarzenie EV8 */
        case I2C_EVENT_MASTER_BYTE_TRANSMITTING:
            if(Tx_Idx < (Tx1BufferSize))
            {
                /* Wysylanie pozostalych bajtow */
                I2C_SendData(I2C1, Buffer_Tx[Tx_Idx++]);
            }
            else
            {
                I2C_ITConfig(I2C1, I2C_IT_BUF, DISABLE);
            }
        break;

        /* Zdarzenie EV8_2 */
        case I2C_EVENT_MASTER_BYTE_TRANSMITTED:

            /* Wygenerowanie sygnalu STOP */
            I2C_GenerateSTOP(I2C1, ENABLE);

            /* Wylaczenie przerwan od zdarzen EV */
            I2C_ITConfig(I2C1, I2C_IT_EVT, DISABLE);
        break;

        default:
        break;
    }
}
```

Listing 5. Funkcja zapisująca rejestr układu PCA9633

```
void PCA9633_Write_Register(uint32_t NumBytesToWrite, uint8_t Address)
{
    Tx1BufferSize = NumBytesToWrite;
    SlaveAddress = Address;
    Tx_Idx = 0;

    /* Wlaczenie przerwan od zdarzen, bledow i buforow */
    I2C_ITConfig(I2C1, I2C_IT_EVT | I2C_IT_ERR | I2C_IT_BUF, ENABLE);

    /* Wystawienie znacznika START */
    I2C_GenerateSTART(I2C1, ENABLE);

    /* Oczekuj, az wystawiony zostanie znacznik START */
    while ((I2C1->CR1&0x100) == 0x100);

    /* Koniec transmisji sygnalizowany wystawieniem znacznika STOP */
    while ((I2C1->SR2 &0x0002) == 0x0002);
}
```

Jeżeli praca ma przebiegać w trybie adresowania 7-bitowego, to pole I2C_AcknowledgedAddress struktury inicjującej powinno zostać ustawione na wartość kryjącą się pod definicją I2C_AcknowledgedAddress_7bit. To, jakie wartości mają poszczególne definicje może być sprawdzone w pliku nagłówkowym urządzenia peryferyjnego, w tym przypadku będzie to plik stm32f10x_i2c.h.

Obsługa magistrali I²C z wykorzystaniem przerwań wymaga znajomości tzw. zdarzeń. Zdarzenia (*events, EV*) są to osobliwe fragmenty komunikacji, istotne, ponieważ sygnalizują potrzebę podjęcia akcji i mogą generować przerwanie. Zdarzenia zostały zdefiniowane w bibliotece API firmy STMicroelectronics w pliku stm32f10x_i2c.h, a jest ich w sumie kilkanaście. Z punktu widzenia komunikacji z kontrolerem PCA9936 ważne są jedynie zdarzenia zawiązane z pracą mikrokontrolera w trybie urządzenia nadrzędnego oraz zdarzenia raportujące błędy. Definicje zdarzeń przedstawiono na **listingu 2**. Zdecydowanie więcej światła na sposób wykorzystania zamieszczonych definicji rzuca fragment komunikacji z **rysunku 10** z zaznaczonymi miejscami wystąpień najważniejszych zdarzeń.

Włączenie przerwań należy do obowiązku funkcji NVIC_Configuration(), jej ciało zamieszczono na **listingu 3**. Jak widać, włączane są przerwania od zdarzeń oraz błędów komunikacji.

Funkcje obsługi przerwań zdarzeń i błędów od kontrolera I²C powinny nazywać się odpowiednio I2C1_EV_IRQHandler()i I2C1_ER_IRQHandler(). Deklaracje funkcji muszą zostać umieszczone w pliku stm32f10x_it.h, a definicje w pliku stm32f10x_it.c.

Funkcja obsługi błędów transmisji jest niezwykle przydatna wówczas, gdy wystąpią problemy i należy rozwiązać zagadkę, co tak naprawdę jest źródłem błędów. Zaawansowane aplikacje mogą natomiast wiedzę na temat źródła problemów stosować w odpowiednim zarządzaniu brakiem komunikacji.

Znacznie częściej podczas poprawnej pracy wywoływana będzie funkcja I2C1_EV_IRQHandler(), jej kod przedstawiono na **listingu 4**. Jest to zmodyfikowana wersja oprogramowania dostarczanego przez firmę STMicroelectronics. Na początku odczytywane są zawartości rejestrów specjalnych (SR1 i SR2) kontrolera magistrali. Jest to czynność niezbędna, ponieważ dzięki niej można rozpoznać, które zdarzenie wywołało przerwanie.

Listing 6. Aplikacja sterująca pracą kontrolera LED

```
/* Adres kontrolera PCA9633 */
#define PCA9633_address 0x0E

/* Bufor dla danych */
uint8_t Buffer_Tx[2];

/* Przechowuje rozmiar bufora do wyslania */
uint8_t Tx1BufferSize = 0;

/* Licznik indeksu bufora danych */
__IO uint8_t Tx_Idx=0;

/* Adres urzadzenia odbiorczego */
uint8_t SlaveAddress;

void GPIO_Configuration(void);
void NVIC_Configuration(void);
void SysTick_Configuration(void);
void I2C_Configuration(void);
void GPIO_Configuration(void);
void PCA9633_Write_Register(uint32_t NumBytesToWrite, uint8_t Address);

int main(void)
{
    RCC_Configuration();
    GPIO_Configuration();
    SysTick_Configuration();
    NVIC_Configuration();
    I2C_Configuration();

    while(1)
    {
        Buffer_Tx[0] = 0x00;   /* Rejestr MODE1 */
        Buffer_Tx[1] = 0x00;
        PCA9633_Write_Register(2, PCA9633_address);
        Buffer_Tx[0] = 0x01;   /* Rejestr MODE2 */
        Buffer_Tx[1] = 0x00;
        PCA9633_Write_Register(2, PCA9633_address);
        Buffer_Tx[0] = 0x08;   /* Rejestr LEDOUT */
        Buffer_Tx[1] = 0xFF;   /* Wszystkie wyjscia wlaczone */
        PCA9633_Write_Register(2, PCA9633_address);
        Buffer_Tx[0] = 0x02;   /* Rejestr PWM0 */
        Buffer_Tx[1] = 0x08;   /* Wypelnienie okolo 3 % */
        PCA9633_Write_Register(2, PCA9633_address);
        Buffer_Tx[0] = 0x03;   /* Rejestr PWM1 */
        Buffer_Tx[1] = 0xA0;   /* Wypelnienie okolo 63 % */
        PCA9633_Write_Register(2, PCA9633_address);
        Buffer_Tx[0] = 0x04;   /* Rejestr PWM2 */
        Buffer_Tx[1] = 0x40;   /* Wypelnienie 25 % */
        PCA9633_Write_Register(2, 0x0E);
        Buffer_Tx[0] = 0x05;   /* Rejestr PWM3 */
        Buffer_Tx[1] = 0x08;   /* Wypelnienie okolo 3 % */
        PCA9633_Write_Register(2, 0x0E);
        while(1);
    }
}
```

Następnie odczytane wartości są kodowane do schematu, jakim posługuje się biblioteka API, a zatem SR2 jest zapisywany na trzecim bajcie zmiennej status (dwa pierwsze do zawartość SR1). Czwarty bajt status jest zerowany, ponieważ nie niesie ze sobą żadnych informacji związanych ze zdarzeniami. Dalej na podstawie wartości zmiennej status określane jest zdarzenie i podejmowane są stosowne akcje.

W celu wysyłania danych do kontrolera LED napisano podstawową funkcję PCA9633_Write_Register(), którą przedstawiono na **listingu 5**. Przyjmowane argumenty to liczba bajtów, jakie będą zapisywane oraz adres kontrolera PCA9633. Bufor dla danych przeznaczonych do wysłania jest zdefiniowany jako zmienna globalna, ponieważ jego zawartość musi być widziana wewnątrz funkcji obsługi przerwania od kontrolera I²C.

Nieskomplikowany program używający kod przedstawiony powyżej zamieszczono na **listingu 6**. Aplikacja ustawia rejestry kanałów indywidualnych diod LED na zadanych wypełnieniach. Tablica globalna będąca buforem danych to Buffer_Tx. Pierwszy element tablicy jest będzie wysłany jako pierwszy po adresie, a więc jest to zawartość rejestru kontrolnego (sterującego). Następnie, jeśli bity autoinkrementacji są wyzerowane, to należy do drugiego elementu tablicy zapisać nową wartość adresowanego rejestru. W ten sposób przygotowana tablica jest gotowa do wysłania do kontrolera LED.

Krzysztof Paprocki, EP
krzysztof.paprocki@ep.com.pl