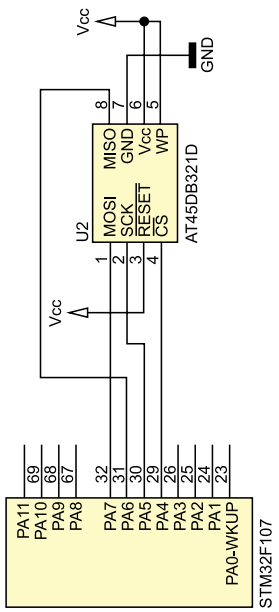
Komunikacja STM32 z pamięcią szeregową

Zagadnienia komunikacji między układami scalonymi w systemach mikroprocesorowych/mikrokontrolerowych cieszą się wciąż niesłabnącą popularnością. Z tego powodu w artykule przedstawiamy komunikację mikrokontrolera z rodziny STM32F107 z szeregową pamięcią DataFlash.

Współczesne mikrokontrolery mają pamięć Flash o coraz większej pojemności. Popularność zyskują układy z wbudowanym 1 lub nawet 2 MB pamięci nieulotnej. Wydawać by się mogło, że stosowanie zewnętrznych pamięci typu DataFlash nie ma już żadnego sensu. Jest to jednak z dwóch powodów złudny wniosek.

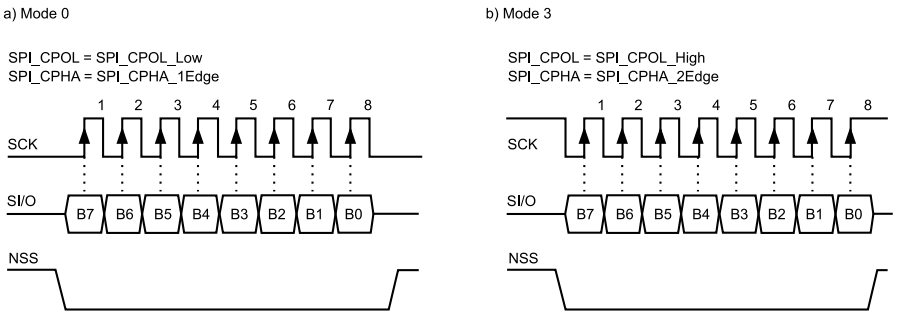
Po pierwsze, niekiedy jednak nawet pojemność rzędu megabajtów jest niewystarczająca. Nie chodzi tutaj o wielkość programu. W zwykłych aplikacjach, bez rozbudowanych systemów operacyjnych, dość trudno jest znaleźć takie, które potrzebowałyby aż tyle pamięci programu. Konieczność stosowania pamięci zewnętrznej może zaistnieć wtedy, kiedy wymagane jest zbieranie i zapisywanie dużej ilości informacji, jak choćby z czujników. Innym przykładem może być urządzenie wyposażone w rozbudowany interfejs użytkownika z wyświetlaczem graficznym. Dodatkowa pamięć nieulotna może w takim przypadku przechowywać bitmapy będące elementami interfejsu.

Po drugie - cena układów. Mikrokontrolery z dużą pamięcią Flash są zazwyczaj wyposażone w wiele układów peryferyjnych, a to bezpośrednio wpływa na ich cenę. Gdy projektowane urządzenie nie będzie musiało wykonywać zbyt skomplikowanych czynności, bardziej opłacalne może okazać się zastosowanie tańszego, a przez to słabiej wyposażonego, mikrokontrolera oraz pamięci zewnętrznej do przechowywania danych.



Rysunek 1. Schemat dołączenia układu AT45DB321D do mikrokontrolera STM32

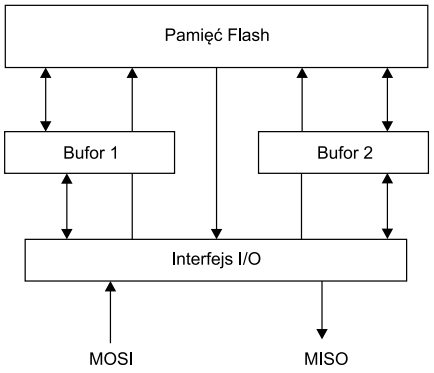
Nie wgłębiając się już bardziej w czynniki wpływające na decyzję konstruktora o stosowaniu pamięci zewnętrznej, przejdziemy do opisu stosownej aplikacji z wykorzysta-



Rysunek 2. Obsługiwane przez pamięć AT45DB321D tryby pracy magistrali SPI

Listing 1. Funkcja inicjująca niezbędne do komunikacji z pamięcią peryferia

```
void dataflashInit(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    SPI_InitTypeDef SPI_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    /* Wybranie grupy priorytetów */
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);

    /* Konfiguracja przerwania od SPI1 */
    NVIC_InitStructure.NVIC_IRQChannel = SPI1_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 2;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_SPI1 | RCC_
APB2Periph_AFIO, ENABLE);

    /* Konfiguracja wyprowadzeń wykorzystywanych przez SPI: SCK(PA5),
MISO(PA6), MOSI(PA7) */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    /* Konfiguracja wyprowadzenia chip select */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    /* Wybranie parametrów pracy kontrolera SPI1 */
    SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
    SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
    SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
    SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
    SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;
    SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;
    SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
    SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_4;
    SPI_Init(SPI1, &SPI_InitStructure);

    SPI_CalculateCRC(SPI1, DISABLE);
    SPI_Cmd(SPI1, ENABLE);
}
```



Rysunek 3. Schemat blokowy pamięci DataFlash AT45DB321D [źródło: Atmel]

niem mikrokontrolera STM32 i szeregowej pamięci DataFlash.

Interfejs komunikacyjny

Wybór pamięci zewnętrznej padł na układ AT45DB321D firmy Atmel. Jest to 32-megabitowa pamięć szeregowa, dostępna w 8-wyprowadzeniowej obudowie typu SOIC8. Wymiana danych z układem pamięci odbywa się przez interfejs SPI, który może pracować z częstotliwością zegarową transmisji, bagatela, 66 MHz. Pamięć do mikrokontrolera STM32 należy dołączyć jak zwykłe urządzenia pracujące z interfejsem SPI. Przykładowy schemat zamieszczono na **rysunku 1**.

Układ AT45DB321D może komunikować się z MCU w trybach *Mode 0* lub *Mode 3*. Szablony komunikacji w obydwu trybach przedstawiono na **rysunku 2**. W *Mode 0* linia zegarowa SCK w stanie spoczynkowym znajduje się w na poziomie niskim, natomiast bity zatrzaskiwane są na zboczach narastających. W drugim trybie (*Mode 3*), gdy dane nie są przesyłane, SCK znajduje się na poziomie wysokim, poszczególne bity są jednak zapisywane również na zboczach narastających.

Sterownik magistrali SPI wbudowany w mikrokontrolery STM32 można ustawić do pracy w obydwu trybach. Przykładowa aplikacja będzie wykorzystywała *Mode 0*. Kod konfigurujący MCU do pracy w tym trybie przedstawiono na **listingu 1** wraz z funkcją *dataflashInit()*, która inicjuje niezbędne peryferia.

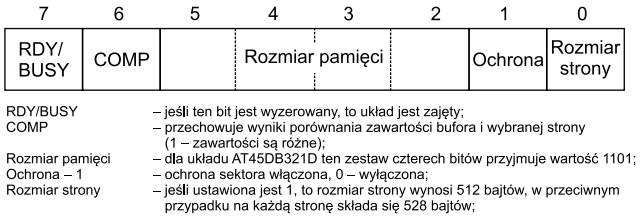
Przedstawiony kod jest wyraźnie uporządkowany w trzy bloki, ich zadaniem jest ustawienie parametrów odpowiednio: kontrolera przerwań NVIC, wykorzystywanych wyprowadzeń oraz samego kontrolera magistrali SPI. Użyto kontroler SPI1 oraz jego domyślne wyprowadzenia:

– PA5 – SCK,
– PA6 – MISO,
– PA7 – MOSI.

Sygnał CS (z ang. *Chip Select*) jest dostarczany programowo poprzez wyprowadzenie PA4. Komunikacja będzie odbywała się w trybie full dupleks, a mikrokontroler będzie oczywiście układem nadrzędnym (z ang. *Master*). Zgodnie z akceptowanym przez układ pamięci formatem ramki danych, jej długość jest ustalana na 8 bitów, przy czym pierwszy na magistralę będzie wystawiany bit najbardziej znaczący (MSB).

Listing 2. Funkcja odczytująca rejestr statusu AT45DB321D

```
void readStatusRegister(void)
{
    /* sprawdzanie muteksa */
    while(SPI_InUse);
    /* „weź” muteks */
    SPI_InUse = 0x11;
    SPI_Cmd(SPI1, DISABLE);
    SPI_I2S_ITConfig(SPI1, SPI_I2S_IT_RXNE, ENABLE);
    SPI_I2S_ITConfig(SPI1, SPI_I2S_IT_TXE, ENABLE);
    /* pierwszy bajt to komenda, drugi to zawartość rejestru */
    Nof_Bytes_To_TxRx = 2;
    /* Komenda „read status register” */
    TxBuf[0] = 0xD7;
    SPI_Cmd(SPI1, ENABLE);
    GPIO_SetBits(GPIOA, GPIO_Pin_4);
    /* CS w stan niski */
    GPIO_ResetBits(GPIOA, GPIO_Pin_4);
}
```



Rysunek 4. Budowa rejestru statusu w układzie AT45DB321D

Ustalanie trybu pracy SPI (czyli w przypadku układu AT45DB321D będzie to wybór pomiędzy Mode 0, a Mode 3) odbywa się przez wypełnienie pól SPI_CPOL i SPI_CPHA struktury inicjującej. Ostatnim parametrem jest wybranie prędkości przesyłania danych. Maksymalna szybkość komunikacji wynosi dla rodziny układów STM32F107 18 MBit/s. Kontroler SPI jest podłączony do wewnętrznej magistrali APB2, która może pracować z maksymalną częstotliwością 72 MHz. Jeśli właśnie taka jest częstotliwość sygnału taktującego APB2, to preskaler kontrolera SPI musi być ustawiony na 4 72 MHz / 4 = 18 MHz).

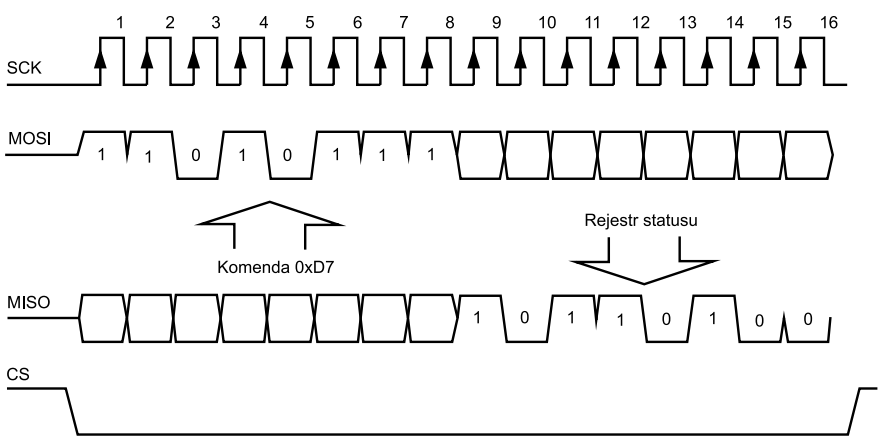
Właściwości pamięci

Producent deklaruje, że minimalna ilość cykli zapisu układu AT45DB321D wynosi co najmniej 100000, a dane przechowywane mogą być przez 20 lat. 100000 cykli zapisu może wydawać się dość dużą liczbą, ale jeśli aplikacja będzie zbyt często dokonywać zapisu, można się spodziewać, że dopuszczalny limit zostanie szybko wyczerpany i pamięć nie będzie się już nadawać do pracy. Aby nie doprowadzić do takiej sytuacji należy unikać cyklicznych lub niekontrolowanych zapisów, np. wyzwalanych z funkcji obsługi jakiegoś przerwania.

Jak już wyżej wspomniano, rozmiar wykorzystanej pamięci to 32 Mbit. Cała przestrzeń jest podzielona na strony i bloki. Rozmiar strony pamięci może wynosić 512 lub 528 bajtów (domyślnie), natomiast proces samego kasowania jest dość elastyczny, ponieważ pozwala na wymazanie całej strony, bloku (4 KB), sektora (64 KB) lub tez całego układu (32 Mbit).

Dostęp do pamięci może odbywać się bezpośrednio, lub z użyciem jednego z dwóch buforów. Schemat blokowy ilustrujący relacje pomiędzy buforami, a interfejsem I/O przedstawiono na **rysunku 3**.

Układ AT45DB321D obsługuje wiele wariacji zapisu i odczytu. Cześć z nich, która wydała się najważniejsza lub najciekawsza została omówiona poniżej.



Rysunek 5. Odczytywanie zawartości rejestru statusu układu pamięci

Odczyt rejestru statusu

Najłatwiejszym, ale zarazem często bardzo istotnym, etapem komunikacji z układem pamięci AT45DB321D jest odczytanie zawartości rejestru statusu. Na **rysunku 4** zamieszczono budowę tego rejestru, natomiast funkcję odczytującą jego zawartość przestawiono na **listingu 2**. Zawartość rejestru statusu jest wystawiana na magistralę po wysłaniu do układu pamięci bajta o wartości 0xD7, według schematu z **rysunku 5**.

Ponieważ do wymiany danych zastosowano przerwania, to „sztywne" blokowanie wykonywania programu w oczekiwaniu na zakończenie komunikacji byłoby niemądre. Z tego powodu wykorzystano nieco bardziej skomplikowany mechanizm. Jak będzie można zauważyć dalej, każda funkcja obsługi pamięci DataFlash ma na początku pętlę while(), która będzie się wykonywać dopóki, dopóty zmienna SPI_InUse będzie miała wartość niezerową (prawdziwą w sensie logicznym). Dopiero po niespełnieniu tego warunku program przejdzie do kolejnych instrukcji. Zaraz poniżej wymienionej pętli zmienna SPI_InUse jest z powrotem ustawiana na wartość niezerową (czyli typu PRAWDA).

Wspomniana wyżej niezerowa wartość zmiennej SPI_InUse wynosząca szesnastkowo 0x11 może wydawać się zupełnie bez sensu. Uzasadnienie znajdujemy w funkcji obsługi przerwania od kontrolera SPI przedstawionej na **listingu 3**. W części nadawczej, gdy wysłany zostanie ostatni bajt nasza flaga SPI_InUse ma kasowany najmłodszy bit (stąd maska 0x01). W części odbiorczej kasowany jest natomiast bit czwarty (maska 0x10). Te wartości mogą być ustalane dowolnie. Celowo tutaj nie zastosowano definicji w miejsce „magicznych" liczb, ponieważ w surowej postaci łatwiej zrozumieć, co się właściwie dzieje. Przedstawiony wyżej mechanizm to nic innego, jak uproszczona wersja zastosowania muteksów.

Zatrzymajmy się na chwile przy muteksach, aby wyjaśnić, co to takiego jest i po co się je stosuje. Każdy zasób, czy to będzie konkretny fragment pamięci, czy też jakieś urządzenie peryferyjne, musi być w danej chwili wykorzystywany jednoznacznie. Wyobraźmy sobie sytuację, w której aplikacja wykorzystuje jakieś urządzenie peryferyjne, niech będzie to kontroler SPI. Jeśli nie chcemy blokować wykonywania programu na czas komunikacji, to może wystąpić sytuacja, w której mikrokontroler będzie próbował uzyskać dostęp do kontrolera SPI w czasie, kiedy poprzedni fragment wymiany danych nie został jeszcze zakończony. Najpewniej spowoduje to wystąpienie błędu lub nie wysłania danych.

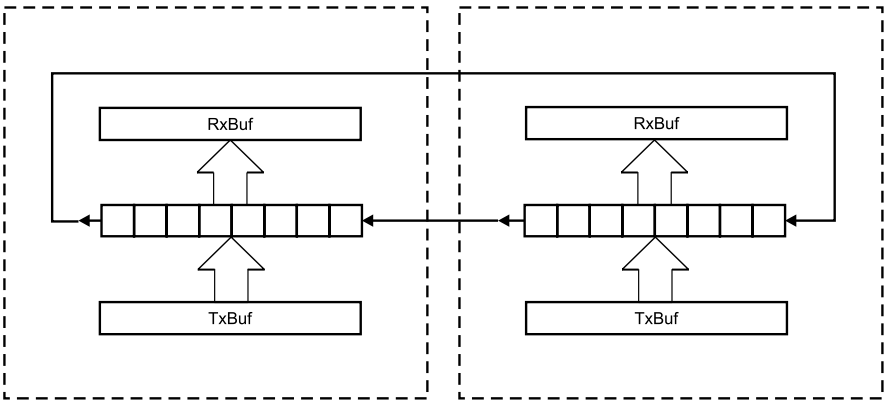
Rozwiązanie takiego konfliktu jest łatwe. Narażonemu na hazardy zasobowi przyporządkowuje się obiekt (zmienną). Owa zmienna, którą można już nazwać muteksem, może przyjmować dwie wartości, dajmy na to zerową (fałsz) i niezerową (prawda). W założeniach projektowych przyjmujemy, że zasób jest wolny, czyli może być wykorzystany, tylko wtedy, kiedy wartość muteksa będzie fałszywa, czyli zerowa. Takie założenie jest oczywiście dowolne, można by przyjąć odwrotną logikę.

Programista, który będzie chciał skorzystać z chronionego zasobu musi najpierw sprawdzić jaka jest wartość muteksa. Jeśli jest ona różna od zera, to należy poczekać, aż muteks zostanie „oddany", co będzie równoznaczne z przypisaniem mu wartości zerowej. Tylko w takiej sytuacji można użyć zasobu, pamiętając o tym, że zaraz na początku należy „zabrać" muteks, a po wykonaniu żądanych operacji „oddać" go.

Wracając do obsługi pamięci – główną funkcję main() programu zamieszczono na **listingu 4**. Po wywołaniu funkcji odczytu rejestru statusu następuje oczekiwanie na zwolnienie muteksa. Gdyby program był bardziej rozbudowany, można by w tym miejscu wykonać inne operacje, a do skopiowania

Listing 3. Obsługa przerwania od kontrolera SPI

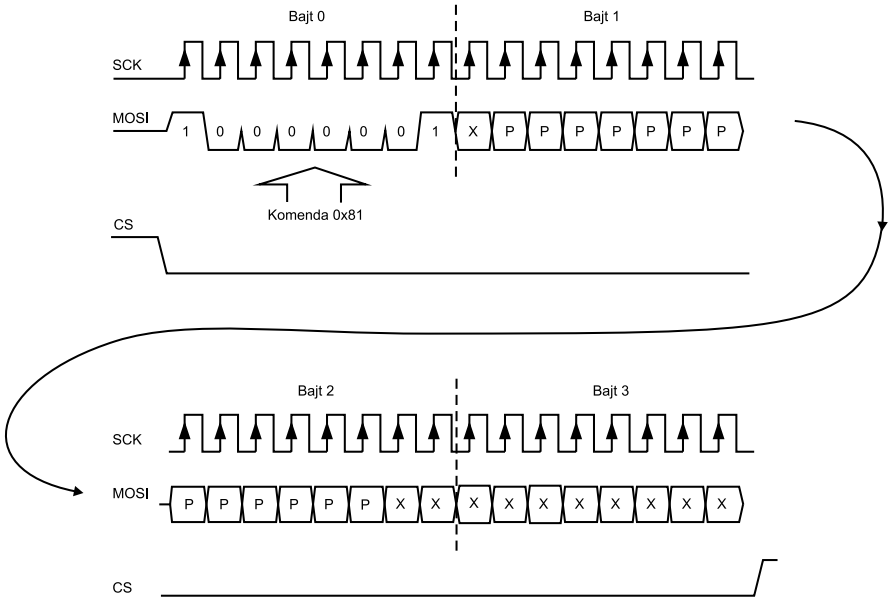
```
void SPI1_IRQHandler(void)
{
    /* Bufor nadawczy jest pusty */
    if (SPI_I2S_GetITStatus(SPI1, SPI_I2S_IT_TXE) != RESET)
    {
        /* Wyślij bajt */
        SPI_I2S_SendData(SPI1, TxBuf[TxIdx++]);

        /* Wyłączenie przerwania od SPI i oddanie połowy muteksa */
        if (TxIdx == Nof_Bytes_To_TxRx)
        {
            SPI_I2S_ITConfig(SPI1, SPI_I2S_IT_TXE, DISABLE);
            TxIdx = 0;
            SPI_InUse &= ~0x01;
        }
    }
    /* W buforze odbiorczym jest odebrany bajt */
    if (SPI_I2S_GetITStatus(SPI1, SPI_I2S_IT_RXNE) != RESET)
    {
        /* Odczyt odebranego bajta */
        RxBuf[RxIdx++] = SPI_I2S_ReceiveData(SPI1);

        /* Wyłączenie przerwania od SPI i oddanie połowy muteksa */
        if (RxIdx == Nof_Bytes_To_TxRx)
        {
            SPI_I2S_ITConfig(SPI1, SPI_I2S_IT_RXNE, DISABLE);
            RxIdx = 0;
            SPI_InUse &= ~0x10;
        }
    }
}
```



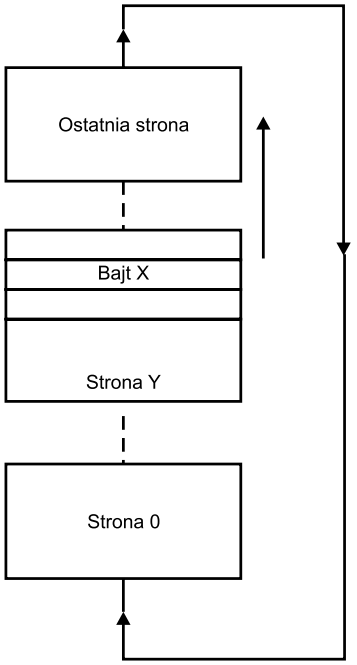
Rysunek 6. Schemat działania interfejsu SPI

Listing 4. Funkcja main() programu odczytującego rejestr statusu układu pamięci

```
int main(void)
{
    /* Włączenie i konfiguracja sygnałów zegarowych */
    SystemInit();

    /* Inicjalizacja niezbędnych peryferiów */
    dataflashInit();

    /* Wyzwolenie procesu odczytu rejestru statusu */
    readStatusRegister();

    /* Oczekiwanie na zwolnienie muteksa */
    while(SPI_InUse);

    /* Zawartość rejestru gotowa do skopiowania */
    statusRegister = RxBuf[1];

    while (1)
    {
    }
}
```



Rysunek 7. Kasowanie wybranej strony w pamięci



Rysunek 8. Ciągły odczyt pamięci bez użycia bufora

odebranej danej przystąpić później, już po zwolnieniu zasobu, w tym przypadku kontrolera SPI. Powód, dla którego odczytywany jest drugi bajt bufora odbiorczego RxBuf[] wynika wprost ze sposobu działania interfejsu SPI, co zobrazowano na **rysunku 6**. Rejestr przesuwny jest z jednej strony zapełniany bitami przychodzącymi po SPI, a drugiej strony poszczególne bity są wysyłane. Ponieważ przerwania od bufora nadawczego i odbiorczego są włączane w tym samym czasie (patrz funkcja readStatusRegister z listingu 2), to pierwsze przerwanie od części odbiorczej zostanie wygenerowane już po wysłaniu bajta komendy 0xD7. Ten pierwszy odebrany bajt jest nieistotny, dopiero drugie przerwanie odczytuje właściwą daną.

Kasowanie strony

Rozkaz kasowania całej wybranej strony ma wartość 0x81, a następujących po nim bajtach adresowych przesyłany jest adres strony według schematu z **rysunku 7**. Pierwszy bajt adresowy zawiera siedem bitów adresu, natomiast dugi sześć. Pozostałe pola, łącznie z całym trzecim bajtem adresowym są bez znaczenia. Proces kasowania strony rozpoczyna po przejściu linii CS w stan wysoki. Zgodnie z informacjami wynikającymi z noty katalogowej układu AT45DB321D maksymalny czas kasowania strony wynosi 35 ms, a typowo 15 ms. Zakończenie operacji kasowania jest również sygnalizowane na linii BUSY układu oraz przez zawartość rejestru statusu.

Na **listingu 5** zamieszczono funkcję kasującą stronę w pamięci erasePage(). Określenie zajętości układu rozwiązano poprzez sprawdzanie zawartości rejestru statusu. Jednak aby nie wstrzymywać pracy całego systemu mikroprocesorowego na niekrótki przecież czas kilkudziesięciu milisekund, użyty został timer SysTick. Funkcja erasePage() tak konfiguruje timer, żeby ten po upływie 35 ms wygenerował przerwanie. Obsługa przerwania, zamieszczona na **listingu 6**, oddaje muteks oraz wyłącza przerwanie.

Listing 5. Ciało funkcji kasującej zawartość strony

```
void pageErase(uint16_t address)
{
    /* sprawdzanie muteksa */
    while(SPI_InUse);
    /* „weź” muteks */
    SPI_InUse = 0x11;
    SPI_Cmd(SPI1, DISABLE);
    SPI_I2S_ITConfig(SPI1, SPI_I2S_IT_RXNE, ENABLE);
    SPI_I2S_ITConfig(SPI1, SPI_I2S_IT_TXE, ENABLE);
    /* pierwszy bajt to komenda, trzy pozostałe to bajty adresu */
    Nof_Bytes_To_TxRx = 4;
    /* Komenda „erase page” */
    TxBuf[0] = 0x81;
    /* Starsze 7 bitów adresu strony */
    TxBuf[1] = (uint8_t) address>>6;
    /* Pozostałe (młodsze) 6 bitów adresu strony,
     * ale wyrównane do lewej strony */
    TxBuf[2] = (uint8_t) address<<2;
    SPI_Cmd(SPI1, ENABLE);
    GPIO_SetBits(GPIOA, CS);
    /* CS w stan niski */
    GPIO_ResetBits(GPIOA, CS);

    /* Oczekiwanie na zakończenie wysyłania komendy i adresu strony */
    while(SPI_InUse);
    /* ponownie „weź” muteks */
    SPI_InUse = 0x11;

    /* SysTick bedzie taktowany z f = 72MHz/8 = 9MHz.
     * Przerwanie ma byc po 35 ms, f = 9MHz, czyli liczy od 320000 - 1 */
    SysTick->LOAD  =  320000 - 1;
    /* Włączenie przerwania od timera SysTick */
    NVIC_SetPriority (SysTick_IRQn, (1<<__NVIC_PRIO_BITS) - 1);
    /* Zerowanie aktualnej wartości licznika */
    SysTick->VAL  = (0x00);
    /* Wypełnienie rejestru kontrolnego timera SysTick:
     * -SYSTICK_CLKSOURCE - bit 2, 0 - HCLK/8, 1 - HCLK
     * -SYSTICK_TICKINT - bit 1, włączenie przerwania od timera
     * -SYSTICK_ENABLE - bit 0, włączenie timera */
    SysTick->CTRL = (0<<SYSTICK_CLKSOURCE) | (1<<SYSTICK_ENABLE) |
(1<<SYSTICK_TICKINT);
}
```

Ciągły odczyt bez użycia bufora

Zasadę działania ciągłego odczytu pamięci bez użycia bufora przedstawiono na **rysunku 8**. Z rysunku wynika, należy podać adres początkowy bajta w stronie, a odczyt następuje już przez całą przestrzeń adresową. Po osiągnięciu ostatniego bajta w ostatniej stronie układ pamięci automatycznie rozpoczyna odczyt od początku pierwszej strony. Jak już wspomniano, domyślny rozmiar strony wynosi 528 bajtów. Dla tego przypadku adresowanie jest następujące. W pierwszej kolejności należy wysłać do układu pamięci bajt komendy, dla ciągłego odczytu bez bufora będzie to wartość 0xE8, a zaraz po nim trzy bajty adresowe. Podział bajtów adresowych na adres strony oraz na adres bajta w stronie zamieszczono na **rysunku 9**, natomiast całą funkcję continousRead(), która inicjuje ciągły odczyt przedstawia **listing 7**. W argumentach do funkcji

należy przekazać adres strony, adres bajta w stronie oraz liczbę bajtów, jakie mają zostać odczytane.

Bufor odbiorczy RxBuf[] powinien być kopiowany dopiero od bajta o indeksie 8, a to dlatego, że pomiędzy ostatnim bajtem adresowym wysłanym do układu pamięci, a pierwszym odebranym bajtem z danymi muszą wystąpić cztery bajty nieznaczące. Zatem mamy „do przeskoczenia": komendę (czyli pierwszy bajt), trzy bajty adresowe oraz cztery bajty nieznaczące.

Zapis do bufora

Pamięć AT45DB321D ma wbudowane dwa bufory SRAM o rozmiarze 528/512 bajtów każdy. Poniżej przedstawiono sposób zapisu danych do jednego z nich. Zapis rozpoczyna się od wysłania do układu pamięci odpowiedniego rozkazu, dla pierwszego bufora będzie to kod 0x84, a dla drugiego 0x87. Po kodzie rozkazu mikrokontroler powinien wysłać trzy bajty adresowe, które, jeśli zinterpretować je jako jeden ciąg bitów, zawierają na początku 14 bitów nieznaczących, a następnie 10 bitów identyfikujących bajt, od którego będzie następował zapis bufora. Po bajtach adresowych można już przesyłać dane przeznaczone do zapisu do bufora. Dane będą zapisywane do czasu, aż na linii chip select (CS) nie pojawi się zbocze narastające. Warto również zaznaczyć, że po osiągnięciu końca bufora, tj, po zapisie jego ostatniego bajta, proces rozpoczyna się od początku bufora. Nieco więcej światła na przedstawioną wyżej komunikację z pewnością rzuci **rysunek 10**.

Funkcję realizującą zapis do bufora przedstawiono na **listingu 8**. W argumentach do funkcji należy przesłać rozkaz, do którego bufora zapis ma być przeprowadzony (0x84 lub 0x87) oraz liczbę zapisywanych bajtów. Globalny bufor nadawczy musi być wypełniony przez wywołaniem funkcji wysyłające dane dla bufora. Podobnie, jak było poprzednio, również funkcja bufferWrite(), zabiera muteks dla kontrolera SPI, blokując w ten sposób dostęp do kontrolera SPI. Jeśli dalej w programie zaistnieje potrzeba skorzystania z SPI, to w pierwszej kolejności należy sprawdzić, czy muteks został oddany, co będzie jednoznaczne z tym, że kontroler SPI jest gotowy do ponownego użycia.

Zapis bufora do pamięci

Gdy żądane operacje zapisu do bufora zostaną już przeprowadzone, należałoby jego zawartość zapisać do pamięci nieulotnej. Możliwe są dwa sposoby zapisu. Pierwszy polega na zapisie zawartości bufora bez kasowania docelowej strony w pamięci, natomiast drugi najpierw wymaże zawartość strony, a dopiero po tym wykonany będzie zapis danych z bufora. Przedstawimy ten drugi sposób.

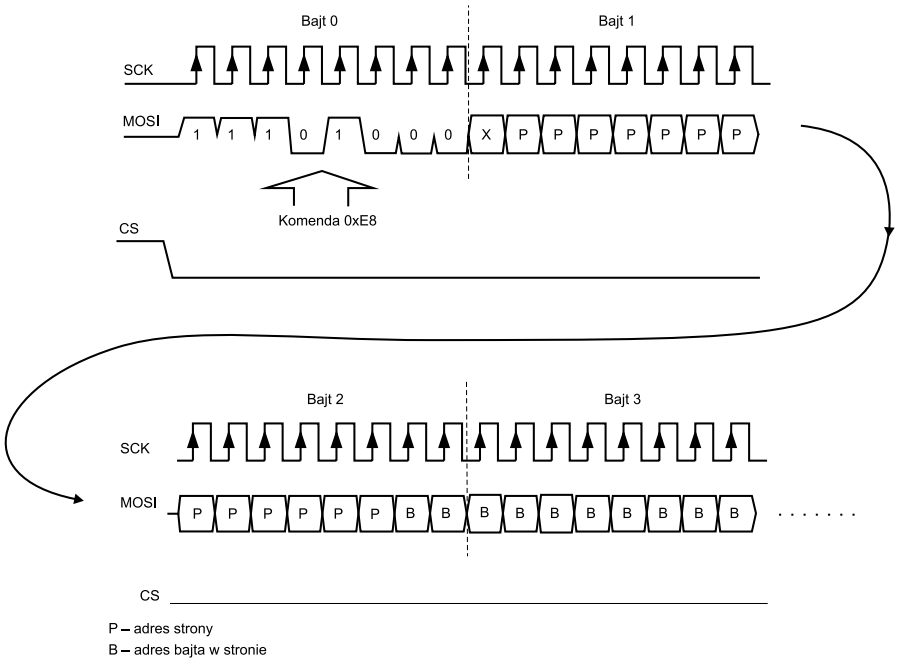
Komenda zapisu bufora do pamięci to odpowiednio na bufora 1 i bufora 2: 0x83 oraz 0x86. Po kodzie komendy muszą zostać wysłane trzy bajty adresowe, określające, która strona w pamięci będzie zapisywana. Trzynaście bitów adresowych powinno być rozmieszczone w wysyłanych bajtach analogicznie do adresowania strony przeznaczonej do skasowania, patrz **rysunek 7**.

Proces kasowania, a następnie zapisywania strony rozpoczyna się po wystąpieniu narastającego zbocza na linii CS. Podobnie, jak miało to miejsce w przypadku kasowania strony, czas wymagany, na wykonie całej operacji nie jest krótki i może wynosić od 17, do 40 ms. Problem oczekiwania na zakończenie operacji kasowania i programowania strony rozwiązano w takim sam sposób, jak przy poleceniu kasowania strony. Do odmierzania czasu użyty został systemowy timer SysTick. Ciało funkcji zapisującej bufor do pamięci przedstawiono na **listingu 9**. Jest to kod analogiczny, do funkcji kasowania strony, różnica, to oczywiście wartość pierwszego bajta (czyli komendy rozkazu), pozostały schemat komunikacji jest taki sam. Jedynie timer SysTick jest konfigurowany do wygenerowania przerwania po upływie dłuższego czasu – po 40 ms.

Krzysztof Paprocki, EP

Listing 6. Funkcja obsługi timera SysTick

```
void SysTick_Handler(void)
{
    /* Wyłączenie timera oraz jego przerwania */
    SysTick->CTRL = (0<<SYSTICK_ENABLE) | (0<<SYSTICK_TICKINT);
    /* Oddanie muteksa */
    SPI_InUse = 0;
}
```



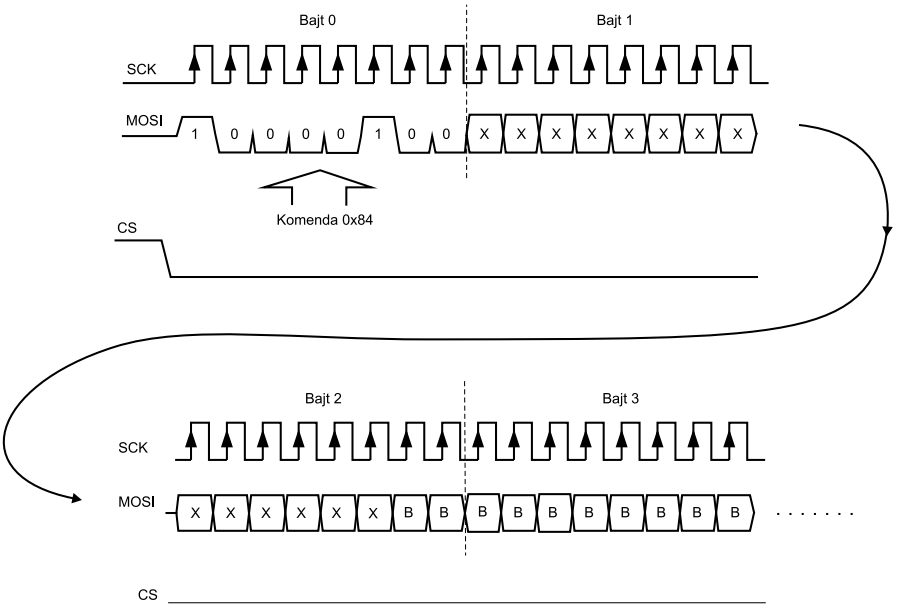
Rysunek 9. Adresowanie pierwszego do odczytu bajta w stronie

Listing 7. Funkcja rozpoczynająca ciągły odczyt pamięci

```
void continousRead(uint16_t pageAddress, uint16_t byteAddress, uint16_t
nofBytesToRead)
{
    /* sprawdzanie muteksa */
    while(SPI_InUse);
    /* „weź" muteks */
    SPI_InUse = 0x11;
    SPI_Cmd(SPI1, DISABLE);
    SPI_I2S_ITConfig(SPI1, SPI_I2S_IT_RXNE, ENABLE);
    SPI_I2S_ITConfig(SPI1, SPI_I2S_IT_TXE, ENABLE);
    /* Nof_Bytes_To_TxRx = nofBytesToRead + komenda + 3xAdres + 4 bajty
nieznaczące */
    Nof_Bytes_To_TxRx = nofBytesToRead + 8;
    /* Komenda „continuous read" */
    TxBuf[0] = 0xE8;
    /* Starsze 7 bitów adresu strony */
    TxBuf[1] = (uint8_t) pageAddress>>6;
    /* Pozostałe (młodsze) 6 bitów adresu strony,
     * ale wyrównane do lewej strony + dwa starsze bity
     * adresu bajta w stronie*/
    TxBuf[2] = ((uint8_t) pageAddress<<2) | ((uint8_t) byteAddress>>8);
    /* Młodsze 8 bitów adresu bajta w stronie */
    TxBuf[3] = (uint8_t) byteAddress;
    SPI_Cmd(SPI1, ENABLE);
    GPIO_SetBits(GPIOA, CS);
    /* CS w stan niski */
    GPIO_ResetBits(GPIOA, CS);
}
```

Listing 8. Ciało funkcji zapisującej bufor SRAM pamięci zewnętrznej

```
void writeBuffer(uint16_t address, uint16_t nofBytesToWrite)
{
    /* sprawdzanie muteksa */
    while(SPI_InUse);
    /* „weź” muteks */
    SPI_InUse = 0x11;
    SPI_Cmd(SPI1, DISABLE);
    SPI_I2S_ITConfig(SPI1, SPI_I2S_IT_RXNE, ENABLE);
    SPI_I2S_ITConfig(SPI1, SPI_I2S_IT_TXE, ENABLE);
    /* pierwszy bajt to komenda, następne trzy to bajty adresu */
    Nof_Bytes_To_TxRx = 4 + nofBytesToWrite;
    /* Komenda „write to buffer 0” */
    TxBuf[0] = 0x84;
    /* Starsze 2 bity adresu bajta w buforze */
    TxBuf[2] = (uint8_t) address>>8;
    /* Pozostałe 8 bitów adresu */
    TxBuf[3] = (uint8_t) address;
    SPI_Cmd(SPI1, ENABLE);
    GPIO_SetBits(GPIOA, CS);
    /* CS w stan niski */
    GPIO_ResetBits(GPIOA, CS);
}
```



Rysunek 10. Zapis do bufora SRAM pamięci AT45DB321D

Listing 9. Funkcja wyzwalająca zapis bufora do pamięci nieulotnej

```
void writeBuffer2Memory(uint16_t address)
{
    /* sprawdzanie muteksa */
    while(SPI_InUse);
    /* „weź” muteks */
    SPI_InUse = 0x11;
    SPI_Cmd(SPI1, DISABLE);
    SPI_I2S_ITConfig(SPI1, SPI_I2S_IT_RXNE, ENABLE);
    SPI_I2S_ITConfig(SPI1, SPI_I2S_IT_TXE, ENABLE);
    /* pierwszy bajt to komenda, trzy pozostałe to bajty adresu */
    Nof_Bytes_To_TxRx = 4;
    /* Komenda „write buffer to memory” */
    TxBuf[0] = 0x83;
    /* Starsze 7 bitów adresu strony */
    TxBuf[1] = (uint8_t) address>>6;
    /* Pozostałe (młodsze) 6 bitów adresu strony,
     * ale wyrównane do lewej strony */
    TxBuf[2] = (uint8_t) address<<2;
    SPI_Cmd(SPI1, ENABLE);
    GPIO_SetBits(GPIOA, CS);
    /* CS w stan niski */
    GPIO_ResetBits(GPIOA, CS);

    /* Oczekiwanie na zakończenie wysyłania komendy i adresu strony */
    while(SPI_InUse);
    /* ponownie „weź” muteks */
    SPI_InUse = 0x11;

    /* SysTick bedzie taktowany z f = 72MHz/8 = 9MHz.
     * Przerwanie ma byc po 40 ms, f = 9MHz, czyli liczy od 360000 - 1 */
    SysTick->LOAD  =  360000 - 1;
    /* Włączenie przerwania od timera SysTick */
    NVIC_SetPriority (SysTick_IRQn, (1<<__NVIC_PRIO_BITS) - 1);
    /* Zerowanie aktualnej wartości licznika */
    SysTick->VAL   = (0x00);
    /* Wypełnienie rejestru kontrolnego timera SysTick:
     * -SYSTICK_CLKSOURCE - bit 2, 0 - HCLK/8, 1 - HCLK
     * -SYSTICK_TICKINT - bit 1, włączenie przerwania od timera
     * -SYSTICK_ENABLE - bit 0, włączenie timera */
    SysTick->CTRL = (0<<SYSTICK_CLKSOURCE) | (1<<SYSTICK_ENABLE) |
(1<<SYSTICK_TICKINT);
}
```