Kurs AVR – lekcja 16

Rozwiązania zadań z ostatniego odcinka

Pierwsze zadanie domowe z poprzedniego odcinka to zegar z budzikiem. Można go napisać, rozbudowując projekt zegara RTC o funkcję alarmu. Jak wspomniano w ostatniej lekcji, układ MCP79410 nie pozwala jednak na wygodne ustawienie alarmu na określoną godzinę i minutę. Z podanych kilku możliwości obejścia tej niedogodności najrozsądniejsze wydaje się ustawienie alarmu z maską sekundową, aby wyzwalał się co minutę, gdy rejestr sekund będzie wynosił 0. W tym celu możemy napisać sobie funkcję pomocniczą zerującą rejestry alarmu (**listing 1**).

Jako parametr funkcja przyjmuje numer alarmu: 0 lub 1. Gdy podamy 0, wyzerowane zostaną rejestry pierwszego alarmu, czyli zaczynając od ALM0SEC. Gdy podamy 1, zerowanie rozpocznie się od ALM1SEC. Do zerowania można było użyć po prostu tablicy bajtów (uint8_t), ale struktura daje większą przejrzystość i można ją wykorzystać także w innych funkcjach operujących na alarmach.

Kolejny konieczny kawałek kodu to wczytywanie czasu alarmu z klawiatury. Odpowiednią funkcję możemy napisać na bazie funkcji ustawiającej bieżący czas i datę. Po wczytaniu godziny i minuty wywoływana jest opisana przed chwilą funkcja konfigurująca alarm a następnie ustawiany jest bit ALM0EN w rejestrze CONTROL, aby włączyć alarm. Wczytany czas alarmu zwracany jest za pomocą wskaźników (**listing 2**).

Wystąpienie alarmu sygnalizowane jest za pomocą nóżki MFP kostki MCP79410. Mikrokontroler musi więc obserwować jej stan. W tym celu możemy wykorzystać przerwania zewnętrzne, np. INT2. Gdy alarm się pojawi, trzeba go jakoś zasygnalizować. Elementem sygnalizacyjnym może być brzęczyk piezo z generatorem, zaś na potrzeby testów wystarczy LED. Załóżmy, że na pinie mikrokontrolera będziemy generować sygnał prostokątny, aby uzyskać dźwięk przerywany lub miganie diody. Zrealizować tę funkcję możemy za pomocą Timer1 pracującego w trybie CTC. Obsługa przerwań zewnętrznych i timera przedstawiona jest na **listingu 3**:

```
ISR(TIMER1_COMPA_vect) {
 if (PORTB & _BV(PORTB0)) {
  PORTB &= ~_BV(PORTB0);
 } else {
  PORTB |= _BV(PORTB0);
 }
}

ISR(INT2_vect) {
 alarmActivated = 1;
}

void startBlink() {
 //CTC, preskaler 1024
 TCCR1B = _BV(CS12) | _BV(CS10) | _BV(WGM12);
}

void stopBlink() {
 TCCR1B = 0;
 PORTB &= ~_BV(PORTB0);
}
```

Przerwanie z timera zmienia stan pinu 0 portu B na przeciwny. Do pinu tego podłączamy piezo lub LED. Przerwanie INT2 ustawia zmienną sygnalizującą przerwanie alarmu, jest ona odczytywana w funkcji main(). Na listingu mamy też dwie funkcje do włączania i wyłączania sygnalizacji alarmu. Pierwsza włącza timer, a druga wyłącza. W funkcji stopBlink() ustawiany jest też stan niski na pinie sygnalizacyjnym, na wypadek gdyby wyłączenie timera nastąpiło akurat wtedy, gdy pin jest w stanie wysokim. Bez tej operacji sygnał pozostawałby włączony, nie byłby jedynie przerywany.

Ostatnim fragmentem kodu jest funkcja main(), przedstawiona na listingu 4. Program rozpoczyna się inicjalizacją kontrolera I²C, klawiatury i wyświetlacza. Dla tego ostatniego definiowany jest znak przedstawiający dzwonek. Wygląd tego znaku określony jest w tablicy customCharBell, zgodnie z zasadami przedstawionymi w odcinku 6. Po co nam dzwonek na wyświetlaczu? Będzie informował o tym, czy alarm jest ustawiony.

Dalej konfigurowane jest przerwanie INT2. Ponieważ pin MFP ma wyjście z otwartym drenem, na połączonym z nim pinie 2 portu B włączamy podciągnięcie do plusa. Dzięki temu zostanie poprawnie wykryte zbocze opadające, gdy włączy się alarm. Jako pin sygnalizacyjny służy pin 0 portu B, ustawiamy go jako wyjście. Następnie ustawiana jest szybkość generowania przerwań z funkcji CTC w Timer1 oraz włączone zostaje przerwanie dla tejże funkcji. Ponieważ nie wiemy, jaki jest stan alarmu w kostce RTC, operacją wykonywaną w części inicjalizacyjnej jest wykasowanie alarmu (**listing 4**).

Tak jak w przykładzie z zegarem RTC, w głównej pętli wyświetlany jest aktualny czas i data. Dodatkowo sprawdzany jest bit ALM0EN w rejestrze CONTROL. Jeśli ustawiony, to znaczy, że alarm jest włączony (ustawiony) i sygnalizujemy to, wyświetlając ikonkę dzwonka. W tym celu do LCD wysyłany jest bajt o kodzie 1, ponieważ pod tym numerem nasz symbol został uprzednio zdefiniowany. Aby wstawić bajt o danej wartości do łańcucha znaków, użyta jest notacja \xhh, gdzie hh oznacza wartość bajtu w postaci szesnastkowej. Jeśli nie ma alarmu, wyświetlamy spacje.

Użytkownik steruje zegarem/budzikiem za pomocą czterech pierwszych klawiszy klawiatury: 1 – ustawianie czasu i daty, 2 – ustawianie czasu alarmu, 3 – wyłączenie sygnalizacji alarmu, 4 – wyłączenie alarmu w RTC. Kod jest tutaj prosty, ale zastanawiające może być wyłączanie alarmu. Widać bowiem, że pisząc do rejestru CONTROL, nie ustawiamy bitu ALM0EN, czyli alarm zostaje wyłączony. Ale po co jest ustawiany bit OUT? Otóż jak może pamiętamy z poprzedniej lekcji, gdy nie są włączone alarmy i nie jest ustawiony bit

Listing 1

```
void rtcWriteMinuteAlarm(uint8_t alarmNumber) {
 RtcAlarmRegisters rtcAlarmRegisters;
 rtcAlarmRegisters.ALM0SEC = 0;
 rtcAlarmRegisters.ALM0MIN = 0;
 rtcAlarmRegisters.ALM0HOUR = 0;
 //dopasowanie sekund
 rtcAlarmRegisters.ALM0WKDAY = 0;
 rtcAlarmRegisters.ALM0DATE = 0;
 rtcAlarmRegisters.ALM0MTH = 0;
 //zapisz rejestry
 rtcWriteRegisters(alarmNumber ? MCP79410_ALM1SEC : MCP79410_ALM0SEC, (uint8_t *) &rtcAlarmRegisters, sizeof(rtcAlarmRegisters));
}
```

Listing 2

```
void setAlarmTime(uint8_t * alarmHour, uint8_t * alarmMinute) {
 //włącz kursor
 lcdWriteCommand(LCD_COMMAND_ON_OFF | LCD_PARAM_ON_OFF_CURSOR | LCD_PARAM_ON_OFF_DISPLAY);
 //wyczyść wyświetlacz
 lcdWriteCommand(LCD_COMMAND_CLEAR);
 //wczytaj czas
 lcdGotoXY(0, 0);
 printf("  :  ");
 lcdGotoXY(0, 0);
 const uint8_t timeFieldsSizes[] = {2, 2};
 uint16_t time[2];
 readSeparatedNumbers(timeFieldsSizes, 2, time);
 *alarmHour = time[0];
 *alarmMinute = time[1];
 //zapisz nowy czas i datę
 rtcWriteMinuteAlarm(0);
 //włącz alarm
 rtcWriteRegister(MCP79410_CONTROL, _BV(MCP79410_ALM0EN));
 //wyłącz kursor
 lcdWriteCommand(LCD_COMMAND_ON_OFF | LCD_PARAM_ON_OFF_DISPLAY);
}
```

SQWEN, stan nóżki MFP zależy od bitu OUT. W naszym kodzie nie ustawiamy bitu ALMPOL, co powoduje, że gdy alarm jest włączony, nóżka MFP jest w stanie wysokim. Gdy alarm się uaktywni, na MFP pojawia się stan niski. Opadające zbocze informuje mikrokontroler o wystąpieniu alarmu. Jeśli teraz wyłączymy alarm (chodzi o wyłączenie funkcji alarmu, nie o skasowanie aktywnego alarmu za pomocą zapisu do ALM0WKDAY), funkcja alarmu przestanie kontrolować MFP i na nóżce tej pojawi się stan niski, co spowoduje wykrycie nieistniejącego alarmu przez mikrokontroler. Trzeba więc zapisać jedynkę do bitu OUT, aby na MFP nadal utrzymywał się stan wysoki.

Pozostała jeszcze obsługa alarmu. Ponieważ alarm wyzwala się co minutę, po jego wyzwoleniu musimy sprawdzić, czy bieżący czas jest zgodny z ustawionym czasem alarmu, przechowywanym w zmiennych alarmHour i alarmMinute. Jeśli stwierdzona zostanie zgodność, włączana jest sygnalizacja.

Niektórzy z Czytelników zapewne zapytają, dlaczego właściwie używane jest przerwanie zewnętrzne, skoro program może po prostu sprawdzać stan pinu w miejscu sprawdzania zmiennej alarmActivated. Owszem, można się tutaj obejść bez przerwania, ale zapewnia ono większą elastyczność. Łatwo np. wprowadzić usypianie mikrokontrolera w celu zmniejszenia poboru prądu i wybudzanie alarmem. Tryby obniżonego poboru prądu w ATmega32 zostaną poruszone w jednej z przyszłych lekcji.

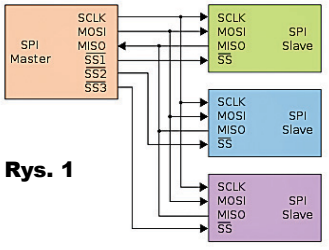
Drugie i trzecie zadanie domowe polegały na napisaniu kodu testującego pamięć SRAM i EEPROM w MCP79410. Można to zrealizować w jednym programie przedstawionym na **listingu 5**.

Kod jest dosyć prosty i nie wymaga szczegółowego omawiania. Są dwie funkcje testujące oba rodzaje pamięci. Zwracają 0 w przypadku błędu i 1 przy sukcesie. Test polega na przygotowaniu testowej tablicy bajtów wypełnionej losowymi bajtami (input), zapisie do pamięci, odczycie do drugiej tablicy (output) i porównaniu tablic. Przy dostępie do SRAM widać wartość 0x20, ponieważ pamięć ta zaczyna się od adresu 20h.

Test EEPROM-u jest nieco bardziej złożony, ponieważ pamięć ta jest podzielona na szesnaście 8-bajtowych stron, które trzeba zapisywać oddzielnie. W związku z tym wprowadzona została dodatkowa pętla, która iteruje po stronach. Natomiast adresem początku każdej strony jest numer tejże strony pomnożony przez 8. Ponadto po zapisie strony dodane jest sprawdzanie, czy układ MCP79410 zakończył wewnętrzną operację zapisu.

Listing 4

```
int main(void) {
 i2cInit();
 keybInit();
 lcdInit();
 lcdInitPrintf();
 lcdDefineChar(customCharBell, 1);
 uint8_t alarmMinute = 0;
 uint8_t alarmHour = 0;
 //podciągnięcie PORTB2
 PORTB |= _BV(PORTB2);
 //włączenie przerwania INT2
 GICR |= _BV(INT2);
 //włącz przerwania globalnie
 sei();
 //PB0 jako wyjście
 DDRB |= _BV(DDB0);
 //szybkość migania
 OCR1A = 4000;
 TIMSK = _BV(OCIE1A);
 //skasuj ewentualny alarm
 rtcWriteRegister(MCP79410_ALM0WKDAY, 0);
 while(1){
  //pobranie aktualnego czasu
  DateTime dateTime;
  rtcReadDateTime(&dateTime);
  //wyświetlenie czasu
  lcdGotoXY(0, 0);
  printf("%02d:%02d:%02d", dateTime.hours, dateTime.minutes, dateTime.seconds);
  //wyświetlenie daty
  lcdGotoXY(0, 1);
  printf("%02d-%02d-%4d", dateTime.dayOfMonth, dateTime.month, dateTime.year);
  printf(" %s", WeekDays[dateTime.dayOfWeek]);
  //wyświetlenie ikonki alarmu jeśli włączony
  uint8_t control;
  rtcReadRegister(MCP79410_CONTROL, &control);
  if (control & _BV(MCP79410_ALM0EN)) {
   printf(" \x01");
  } else {
   printf("  ");
  }
  //odczyt klawiatury
  uint8_t key = 0;
  if ((key = readKeyboard())) {
   while(readKeyboard());
   if (key == 1) setDateTime();
   if (key == 2) setAlarmTime(&alarmHour, &alarmMinute);
   if (key == 3) {
    stopBlink();
   }
   //wyłączenie alarmu, MFP=1
   if (key == 4) rtcWriteRegister(MCP79410_CONTROL, _BV(MCP79410_OUT));
  }
  if (alarmActivated) {
   alarmActivated = 0;
   //skasowanie alarmu
   rtcWriteRegister(MCP79410_ALM0WKDAY, 0);
   //pobranie aktualnego czasu
   rtcReadDateTime(&dateTime);
   if (dateTime.hours == alarmHour && dateTime.minutes == alarmMinute) {
    startBlink();
   }
  }
 }
}
```



Rys. 1

Standard SPI

Serial Peripheral Interface (SPI) to popularny standard wymiany danych między układami scalonymi. Tak jak w poprzednio poznanych standardach, wykorzystywany jest model master – slave, w którym jest jeden układ nadrzędny sterujący transmisją, zwykle mikrokontroler, oraz jeden lub więcej układów podrzędnych. SPI w stosunku do poznanego ostatnio I²C różni się dwiema zasadniczymi cechami: może przesyłać dane w trybie dupleks oraz nie ma adresowania.

Dupleks oznacza, że dany układ może jednocześnie nadawać i odbierać dane. W związku z tym obecne są dwie linie danych: MOSI przesyłająca dane z mastera do slave'a oraz MISO przesyłająca dane z slave'a do mastera.

Skąd jednak wiadomo, z którym układem scalonym master wymienia dane, jeśli nie ma adresowania? Otóż każdy układ podrzędny ma wejście /SS, za pomocą którego master informuje go, że został wybrany i transmisja będzie odbywać się właśnie z nim. Jak podpowiada nam ukośnik, aktywnym stanem jest niski stan logiczny. Jeśli na wejściu /SS będzie stan wysoki, slave ignoruje sygnały pojawiające się na liniach SCLK, MISO oraz MOSI, a jego nóżka MOSI jest w stanie wysokiej impedancji, aby nie przeszkadzać w transmisji innym układom (**rysunek 1**).

Żeby wybierać układy podrzędne, układ mikrokontrolera musi mieć przeznaczoną odpowiednią liczbę nóżek do sterowania wejściami /SS układów podrzędnych, dla każdego układu po jednej nóżce. Jeśli w urządzeniu będzie większa liczba układów scalonych na szynie SPI, a mikrokontroler będzie miał mało wolnych nóżek, można skorzystać z multipleksera.

Ogólnie więc w SPI wykorzystywane są cztery sygnały:

- SCLK: Serial Clock – sygnał zegarowy generowany przez mastera
- MOSI: Master Output

Slave Input – wyjście mastera, wejście slave'a
- MISO: Master Input Slave Output – wejście mastera, wyjście slave'a
- /SS: Slave Select – wybór slave'a

Można spotkać również nazwy alternatywne. SLCK może mieć krótszą formę SCK, a zamiast /SS może być /CS (chip select). Ponadto ukośnik może być opuszczany lub zastępowany literką n (nSS). Z kolei wyprowadzenia danych mogą być oznaczone jako SDI i SDO (Serial Data Input, Serial Data Output), przy czym dla mastera SDO będzie równoważne z MOSI, a SDI z MISO, natomiast dla slave'a odwrotnie.

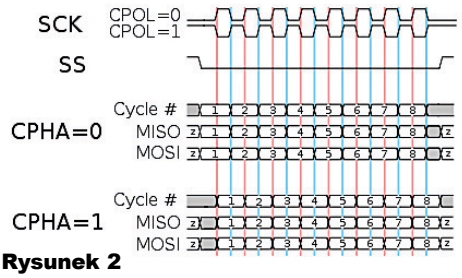
Gdy układ nadrzędny wysteruje linię /SS układu podrzędnego, może rozpocząć się wymiana danych. Master generuje sygnał zegarowy i w jego takt przesyłane są kolejne bity, jednocześnie na liniach MISO i MOSI. Dane są zwykle zorganizowane w bajty, czyli po 8 bitów, ale czasami można spotkać też inną liczbę bitów. Bity transmitowane są od najstarszego do najmłodszego. Ponieważ na szynie SPI dane są przesyłane jednocześnie w dwie strony, nie ma odrębnych operacji wysyłania i odbierania. Jest tylko operacja wymiany/transferu danych. Jeśli mówimy np. o wysłaniu jakiegoś bajtu, to dotyczy to już konkretnego protokołu danego układu scalonego. Zwykle w takim kontekście wysyłanie bajtu oznacza wysłanie tego bajtu i zignorowanie bajtu przychodzącego. Natomiast odebranie oznacza wysłanie dowolnego bajtu i wzięcie do dalszego przetwarzania bajtu odebranego. De facto transmisja danych odbywa się więc wtedy w półdupleksie.

Podobnie jak w przypadku I²C, częstotliwość zegara można zmieniać w szerokich granicach. Nie ma ustalonych standardowych prędkości, natomiast każdy układ ma podaną przez producenta maksymalną szybkość transmisji. Zwykle jest ona dosyć wysoka, przynajmniej w porównaniu do I²C i wynosi kilka megabitów na sekundę.

Oprócz prędkości, ważnymi parametrami konfiguracyjnymi szyny SPI jest polaryzacja zegara oraz faza. Parametry te oznaczane są odpowiednio jako CPOL oraz CPHA i mogą przyjmować wartości 0 lub 1. Polaryzacja mówi o tym, jaki stan ma linia SCLK w spoczynku. Jeśli CPOL=0, w czasie gdy nie są transmitowane dane, SCLK pozostaje w stanie niskim. Gdy CPOL=1, SCLK w spoczynku jest w stanie wysokim. Z kolei CPHA mówi, w jakim momencie układ odbierający ma odczytywać oraz ustawiać stan linii danych. Gdy CPHA=CPOL, bity są odczytywane z linii na narastającym zboczu sygnału zegarowego, a wystawiane na zboczu opadającym. Gdy CPHA ma inną wartość niż CPOL, bity są odczytywane na opadającym zboczu, a wystawiane na narastającym (**rysunek 2**). Ponieważ mamy dwa jednobitowe parametry, daje to razem cztery kombinacje. Są one nazywane trybami SPI i zestawione zostały w **tabeli 1**. Większość układów scalonych SPI działa w trybie 0.

Tabela 1

Tryb	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1



Rysunek 2

Obsługa SPI w ATmega32

Nasz mikrokontroler ma sprzętową obsługę standardu SPI. Piny /SS, MOSI, MISO i SCK to odpowiednio PB4, PB5, PB6 i PB7. Od strony programowej korzystamy z trzech rejestrów:
- SPCR – rejestr konfiguracyjny
- SPSR – rejestr statusowy
- SPDR – rejestr danych

W SPCR znajdują się następujące bity:
- SPIE – włącza generowanie przerwania na koniec transmisji
- SPE – włącza kontroler SPI
- DORD – gdy ustawiony jest na zero, bity są transmitowane od najstarszego, gdy na 1 – od najmłodszego
- MSTR – jedynka przełącza kontroler w tryb master, zero w slave
- CPOL – polaryzacja
- CPHA – faza
- SPR1 – konfiguracja prędkości
- SPR0 – konfiguracja prędkości

Korzystając z SPI, na pewno ustawimy bit SPE. Jeśli zajdzie potrzeba generowania przerwania na koniec transferu bajtu, ustawimy też bit SPIE. Bity DORD, CPOL i CPHA będą zwykle wyzerowane, chyba że trafi się nam jakiś nietypowy układ slave. Praktycznie zawsze nasz mikrokontroler będzie pełnił funkcję mastera, bit MSTR będzie więc ustawiony. Z kolei prędkość transmisji wyznaczana jest przez bity SPR1 i SPR0, ale też SPI2X z rejestru SPSR. Wszystkie trzy bity określają, ile razy częstotliwość taktowania szyny SPI będzie mniejsza od częstotliwości taktowania mikrokontrolera. Ilustruje to **tabela 2**. Jak widać, bit SPI2X zmniejsza dwukrotnie stopień podziału, a więc przyspiesza szynę SPI dwukrotnie. Jeśli przy procesorze taktowanym zegarem 16 MHz chcemy uzyskać prędkość transmisji na poziomie 1 Mb/s, ustawimy bit SPR0.

W rejestrze SPSR oprócz wspomnianego bitu SPI2X znajdują się bity SPIF i WCOL. Pozostałe 5 bitów są nieużywane. SPIF to flaga przerwania generowanego, gdy zakończy się transfer bajtu.

Tabela 2

SPI2X	SPR1	SPR0	Podział przez
0	0	0	4
0	0	1	16
0	1	0	64
0	1	1	128
1	0	0	2
1	0	1	8
1	1	0	32
1	1	1	64

```
#include <stdlib.h>
#include <avr/io.h>
#include <util/delay.h>
#include "i2c.h"
#include "lcd.h"
#include "MCP79410.h"

uint8_t testSRAM();
uint8_t testEEPROM();

int main(void) {
 lcdInit();
 lcdInitPrintf();
 i2cInit();
 if (testSRAM()) printf("SRAM OK"); else printf("RAM ERROR");
 lcdGotoXY(0, 1);
 if (testEEPROM()) printf("EEPROM OK"); else printf("EEPROM ERROR");
 while (1) { }
}

uint8_t testSRAM() {
 uint8_t input[64];
 uint8_t output[64];
 for (uint8_t i = 0; i < sizeof(input); i++) {
  input[i] = rand();
 }
 rtcWriteRegisters(0x20, input, sizeof(input));
 rtcReadRegisters(0x20, output, sizeof(input));
 for (uint8_t i = 0; i < sizeof(input); i++) {
  if (output[i] != input[i]) {
   return 0;
  }
 }
 return 1;
}

uint8_t testEEPROM() {
 for (uint8_t page = 0; page < 16; page++) {
  uint8_t input[8];
  uint8_t output[8];
  for (uint8_t i = 0; i < sizeof(input); i++) {
   input[i] = rand();
  }
  eepromWriteBytes(page * 8, input, sizeof(input));
  while (eepromIsBusy()) _delay_ms(1);
  eepromReadBytes(page * 8, output, sizeof(input));
  for (uint8_t i = 0; i < sizeof(input); i++) {
   if (output[i] != input[i]) {
    return 0;
   }
  }
 }
 return 1;
}
```

Listing 5

Flagę czyści się przez odczyt rejestru SPRS a następnie rejestru SPDR. WCOL to flaga kolizji, którą mikrokontroler ustawia, jeśli wykonany zostanie zapis do rejestru SPDR, zanim nie skończy się transmisja poprzednio wpisanego bajtu. Zerowanie tej flagi odbywa się tak samo jak SPIF.

SPDR to rejestr transmitowanych danych. Zapis do tego rejestru powoduje rozpoczęcie transmisji wpisanego bajtu. Gdy transmisja zakończy się, z rejestru można odczytać odebrany bajt.

Omówienia wymaga jeszcze pin /SS. Gdy ATmega32 jest skonfigurowana jako slave, /SS działa zawsze jako wejście. Pojawienie się stanu niskiego włącza odbieranie danych na nóżce MOSI a MISO staje się wyjściem i nadaje dane. Gdy panuje stan wysoki, kontroler SPI wyłącza się, a nóżka MISO może być ustawiona jako wyjście przez użytkownika.

W przypadku pracy mikrokontrolera jako master, co zwykle będzie miało miejsce, użytkownik może określić kierunek pinu /SS. Jeśli zostanie ustawiony jako wyjście, będzie zachowywać się jak zwykły pin, niezwiązany z SPI. Natomiast po ustawieniu jako wejście, musi być na niego podawany z zewnątrz stan wysoki, aby kontroler SPI mógł działać. Gdy pojawi się stan niski, zostanie to zinterpretowane jako pojawienie się innego mastera na szynie SPI. Następuje wtedy wyzerowanie bitu MSTR i ATmega32 staje się układem podrzędnym SPI. Dodatkowo zostaje ustawiona flaga SPIF. W typowych zastosowaniach /SS będzie pracować jako wyjście i sterować wejściem /SS jednego z układów podrzędnych.

Biblioteka SPI

Jak mogliśmy się przekonać, SPI jest bardzo prostym standardem, a jego obsługa w naszym mikrokontrolerze bardzo łatwa. Bibliotekę obsługującą SPI można stworzyć praktycznie z zaledwie dwóch krótkich funkcji – **listing 6**.

```
void spiInit(void) {
 // /SS, MOSI, SCK jako wyjścia
 DDRB |= _BV(PORTB4) | _BV(PORTB5) | _BV(PORTB7);
 // /SS w stanie wysokim
 SS_HIGH;
 //SPI włączone, tryb master, podział przez 16
 SPCR = _BV(SPE) | _BV(MSTR) | _BV(SPR0);
}

uint8_t spiTransfer(uint8_t data) {
 //rozpocznij transmisję
 SPDR = data;
 //poczekaj na koniec
 while(!(SPSR & _BV(SPIF)));
 //zwróć odebraną wartość
 return SPDR;
}
```

Listing 6

Inicjalizacja to skonfigurowanie odpowiednich pinów jako wyjścia, włączenie kontrolera SPI w trybie master i ustawienie prędkości transmisji. Pin /SS jest ustawiany w stanie spoczynkowym, czyli wysokim. Natomiast aby wykonać transfer na szynie SPI, trzeba zapisać wysyłany bajt do rejestru SPDR, poczekać na koniec transmisji sygnalizowany flagą SPIF oraz zwrócić odebrany bajt. Jeśli zdarzyłoby się, że procesor miałby pracować jako slave, dobrze jest łapać przerwanie SPI, aby mieć powiadomienie o nadchodzących danych (obsługa przerwania w procedurze SPI_STC_vect).

SS_HIGH wraz z SS_LOW to makra zdefiniowane podobnie jak w bibliotece dla LCD:

```
#define SS_LOW PORTB &= ~_BV(PORTB4)
#define SS_HIGH PORTB |= _BV(PORTB4)
```

Układ 25LC080

Układem znajdującym się na naszej płytce testowej, który pomoże nam zaznajomić się ze standardem SPI, jest kostka 25LC080 będąca pamięcią EEPROM o pojemności 1kB. Maksymalne obsługiwane przez nią taktowanie szyny SPI wynosi 2MHz. EEPROM jest zorganizowany w 64 strony po 16 bajtów. Istnieje też rejestr statusowy, pełniący także funkcję rejestru konfiguracyjnego. Struktura tego rejestru przedstawiona jest w **tabeli 3**.

7	6	5	4	3	2	1	0
WPEN	-	-	-	BP1	BP0	WEL	WIP

Tabela 3

Bit WPEN włącza ochronę rejestru przed zapisem. Stan tego bitu jest brany pod uwagę pod warunkiem, że pin /WP jest w stanie niskim. Jeśli pin /WP ustawimy w stan wysoki, zapis do rejestru statusowego będzie możliwy, nawet jeśli WPEN będzie ustawiony na 1. Bity BP1 i BP0 pozwalają włączyć ochronę przed zapisem dla części EEPROM-u, zgodnie z **tabelą 4**.

Bity WEL i WIP są bitami statusowymi, tylko do odczytu. WEL przedstawia stan tzw. zatrzasku zapisu (Write Enable Latch), który musi być ustawiony przed wykonaniem operacji zapisu. WIP informuje, czy trwa wewnętrza operacja zapisu (Write-In-Process).

Tabela 4

BP1	BP0	Obszar chroniony
0	0	brak
0	1	górna ćwiartka 0300h - 03FFh
1	0	górna połowa 0200h - 03FFh
1	1	całość 0000h - 03FFh

Układ posiada następujące wyprowadzenia, dostępne na złączu EEPROM:

- /CS – /SS
- SO – MISO
- /WP – ochrona przed zapisem
- /HOLD – wstrzymywanie transmisji
- SCK – SCLK
- SI – MOSI

Omówienia wymaga jeszcze /HOLD. Pin ten pozwala na wstrzymanie transmisji przez mastera i wznowienie jej później. Gdy jest w stanie niskim, układ 25LC080 ignoruje sygnał zegarowy oraz nie transmituje i nie odbiera danych. Funkcja ta została pomyślana pod kątem mikrokontrolerów pracujących jako master, które w trakcie transmisji SPI muszą obsłużyć przerwanie i na chwilę wstrzymać obsługę szyny SPI. Zazwyczaj nie ma potrzeby korzystania z tej funkcji i pin /HOLD powinien być podłączony do plusa zasilania.

Obsługa układu 25LC080

Użytkownik ma do dyspozycji sześć funkcji, jakie może wykonać na układzie 25LC080 (**tabela 5**).

Aby odczytać dane z układu 25LC080, należy najpierw ustawić jego pin /CS w stan niski. Następnie należy wydać komendę READ, czyli wysłać bajt o wartości 3. Następnie trzeba wysłać 16-bitowy adres komórki, od której ma być rozpoczęty odczyt. Potem można odczytać dowolną liczbę bajtów, układ będzie zwracał bajty z kolejnych adresów. Jeśli będziemy kontynuować odczyt po odczytaniu ostatniego bajtu (z adresu 3FFh), wewnętrzny licznik adresowy zresetuje się i będziemy otrzymywać bajty od początku pamięci (od adresu 000h). Odczyt kończy się ustawieniem pinu /CS w stan wysoki.

Komendy WRDI i WREN nie mają żadnych parametrów. Ich wysłanie polega więc na ustawieniu /CS w stan niski, wysłaniu jednego bajtu i ustawieniu /CS w stan wysoki. Należy o tym pamiętać, bo /CS nie tylko wybiera/adresuje układ, ale też rozdziela komendy. Nawet jeśli kostka 25LC080 będzie jedynym układem slave na szynie SPI, nie możemy jej nóżki /CS podłączyć na stałe do masy.

Zapis danych komendą WRITE wygląda tak samo jak odczyt komendą READ. Trzeba jednak pamiętać o dwóch rzeczach. Po pierwsze najpierw trzeba wydać komendę WREN, aby włączyć zatrzask zapisu. Zatrzask ten bowiem jest zresetowany po włączeniu zasilania oraz resetuje się po każdym zapisie EEPROM-u oraz po odczycie i zapisie rejestru statusowego. Po drugie EEPROM jest stronicowany, podobnie jak było to przy omawianej w poprzednich dwóch lekcjach kostce MCP79410. W jednej operacji zapisu nie

można więc zapisać więcej, niż wynosi rozmiar strony (16 bajtów). Zapisanie większej liczby bajtów spowoduje reset wewnętrznego licznika i zapis od początku strony.

Po zakończeniu wysyłania danych i ustawieniu /CS w stan wysoki układ 25LC080 potrzebuje do 5ms, aby dokończyć zapis. Stąd też konieczne jest sprawdzanie stanu bitu WIP, żeby nie rozpoczynać nowego zapisu przed zakończeniem poprzedniego.

Instrukcje RDSR i WRSR odczytują i zapisują jeden bajt – status rejestru. Po wysłaniu więc numeru instrukcji wykonywany jest transfer jednego bajtu.

Instrukcja	Numer instrukcji	Opis
READ	3	Odczyt danych
WRITE	2	Zapis danych
WRDI	4	Wyłączenie zatrzasku zapisu (blokada zapisu)
WREN	6	Włączenie zatrzasku zapisu (odblokowanie zapisu)
RDSR	5	Odczyt rejestru statusowego
WRSR	1	Zapis rejestru statusowego

Tabela 5

Biblioteka dla 25LC080

Na podstawie przedstawionych informacji o działaniu naszego układu możemy napisać bibliotekę do wygodnego korzystania z niego w naszych programach. Znajdą się w niej podobne funkcje jak w bibliotece dla MCP79410, czyli służące do odczytu i zapisu pojedynczych bajtów lub tablic, a także funkcja do sprawdzania, czy układ jest zajęty przeprowadzaniem wewnętrznej operacji zapisu. Dojdą też funkcje do odczytu i zapisu rejestru statusowego oraz tzw. zatrzasku zapisu. Kod biblioteki przedstawiony jest na **listingu 7**.

Funkcje odczytu i zapisu danych wyglądają podobnie do siebie. Po ustawieniu /CS w stan niski wysyłana jest odpowiednia komenda, następnie starsze 8 bitów adresu, młodsze 8 bitów adresu i w końcu dane. W przypadku zapisu na samym początku wywoływana jest jeszcze funkcja eepromWriteEnable(), która wysyła instrukcję WREN. Odczyt i zapis rejestru statusu wygląda jak odczyt i zapis bajtu, ale nie jest wysyłany adres. Kod wykorzystuje makra dla poszczególnych instrukcji. Nazwy makr zaczynają się od podkreślenia, ponieważ zgodnie z zasadami języka C nie mogą zaczynać się od cyfr. Funkcja eepromIsBusy() odczytuje rejestr statusu i zwraca 1, jeśli jest ustawiony bit WIP.

Korzystając z utworzonej biblioteki, możemy napisać prosty kod, który będzie ją testował. Przykład znajduje się na **listingu 8**.

Po inicjalizacji SPI i LCD program zapisuje litery „Tes” oraz znak NULL w komórkach 0–3. Następnie odczytuje te komórki i wyświetla ich zawartość na LCD. Potem do komórki 789 zapisywana jest literka t. Zawartość komórki również trafia na LCD. Jeśli wszystko jest w porządku, na ekranie zobaczymy słowo Test. Adresy 0–3 i 789 są tylko przykładem. Wybierając adresy, trzeba pamiętać, że pamięć jest podzielona na 16-bajtowe strony. Ponadto ze względu na rozmiar EEPROM-u, wynoszący 1kB, maksymalny adres to 1023.

Aby uruchomić program, trzeba odpowiednio połączyć piny złącza EEPROM na płytce testowej: /CS z PB4, SI z PB5, SO z PB6, SCK z PB7, /WB z GND i /HOLD z VCC.

Listing 7

```
#include "25LC080.h"
uint8_t eepromReadByte(uint16_t address) {
 SS_LOW;
 spiTransfer(_25LC080_READ);
 spiTransfer(address >> 8);
 spiTransfer(address);
 uint8_t data = spiTransfer(0);
 SS_HIGH;
 return data;
}
void eepromReadBytes(uint16_t address, uint8_t data[], uint16_t size) {
 SS_LOW;
 spiTransfer(_25LC080_READ);
 spiTransfer(address >> 8);
 spiTransfer(address);
 for (uint16_t i = 0; i < size; i++) {
  data[i] = spiTransfer(0);
 }
 SS_HIGH;
}
void eepromWriteByte(uint16_t address, uint8_t data) {
 eepromWriteEnable();
 SS_LOW;
 spiTransfer(_25LC080_WRITE);
 spiTransfer(address >> 8);
 spiTransfer(address);
 spiTransfer(data);
 SS_HIGH;
}
void eepromWriteBytes(uint16_t address, uint8_t data[], uint16_t size) {
 eepromWriteEnable();
 SS_LOW;
 spiTransfer(_25LC080_WRITE);
 spiTransfer(address >> 8);
 spiTransfer(address);
 for (uint16_t i = 0; i < size; i++) {
  spiTransfer(data[i]);
 }
 SS_HIGH;
}
uint8_t eepromReadStatus(void) {
 SS_LOW;
 spiTransfer(_25LC080_RDSR);
 uint8_t data = spiTransfer(0);
 SS_HIGH;
 return data;
}
void eepromWriteStatus(uint8_t data) {
 eepromWriteEnable();
 SS_LOW;
 spiTransfer(_25LC080_WRSR);
 spiTransfer(data);
 SS_HIGH;
}
void eepromWriteEnable(void) {
 SS_LOW;
 spiTransfer(_25LC080_WREN);
 SS_HIGH;
}
void eepromWriteDisable(void) {
 SS_LOW;
 spiTransfer(_25LC080_WRDI);
 SS_HIGH;
}
uint8_t eepromIsBusy(void) {
 if (eepromReadStatus() & _BV(_25LC080_WIP)) return 1; else return 0;
}
```

Zadania

Zachęcam jak zwykle do wymyślania własnych programów. Przykładowe zadania w tym odcinku:

1. Program testujący ochronę obszarów pamięci EEPROM,
2. Zamek szyfrowy z kodem przechowywanym w EEPROM.

W materiałach dodatkowych znajdują się kompletne projekty rozwiązań zadań domowych oraz projekt zawierający bibliotekę SPI wraz z programem testującym komunikację z 25LC080. Zamieszczona jest także karta katalogowa układu 25LC080.

Grzegorz Niemirowski
grzegorz@grzegorz.net

Listing 8

```
#include <avr/io.h>
#include <util/delay.h>
#include "spi.h"
#include "25LC080.h"
#include "lcd.h"

int main(void) {
 spiInit();
 lcdInit();
 lcdInitPrintf();
 //test zapisu tablicy
 uint8_t testString[] = "Tes";
 eepromWriteBytes(0, testString, sizeof(testString));
 while (eepromIsBusy()) _delay_ms(1);
 uint8_t buffer[sizeof(testString)];
 eepromReadBytes(0, buffer, sizeof(testString));
 printf((char*)buffer);
 //test zapisu bajtu
 eepromWriteByte(789, 't');
 while (eepromIsBusy()) _delay_ms(1);
 printf("%c", eepromReadByte(789));
 while(1){ }
}
```