Kurs AVR – lekcja 14

Rozwiązania zadań z ostatniego odcinka

Zadania domowe z ostatniej lekcji dotyczyły standardu 1-Wire. Pierwsze z nich polegało na wyświetlaniu na bieżąco liczby podłączonych układów 1-Wire. Aby je zrealizować, możemy skorzystać z funkcji oneWireSearchRom(), która służy do wyszukiwania układów na linii 1-Wire. Jak pamiętamy, po znalezieniu ostatniego układu funkcja zwraca 0. Jeśli nie ma żadnych układów, zwraca –1. Aby policzyć układy, trzeba tę funkcję uruchamiać tak długo, aż nie zwróci 0, a licznik układów zwiększać, gdy zwrócona wartość jest większa od –1. Przykładowa realizacja pokazana jest na **listingu 1**. Należy pamiętać, że program do swojej pracy wymaga podłączenia mikrokontrolera do linii 1-Wire, czyli de facto do rezystora podciągającego. Testując zachowanie programu bez podłączonego żadnego układu, trzeba o tym rezystorze pamiętać. Przy pinie wiszącym w powietrzu zamiast zera na wyświetlaczu pokaże się losowa liczba.

Zadaniem drugim był termometr z wyświetlaczem LED (**rysunek 1**). Było to dosyć proste ćwiczenie, bo wystarczyło wziąć listing 7 z lekcji 13 i zamiast biblioteki LCD wykorzystać bibliotekę LED. Dodatkowo dla wygody dobrze jest wynieść odczyt temperatury do oddzielnej funkcji, a w main() tylko ją wywoływać i wyświetlać wynik. Przykładowy kod zaprezentowano na **listingu 2**.

W trzecim zadaniu trzeba było dodać funkcję alarmu. Układy DS18B20 mają funkcję alarmu i polega ona na tym, że jeśli wykonywane jest wyszukiwanie układów 1-Wire, ale zamiast komendy Search ROM użyta zostanie Alarm Search (ECh), to zgłaszają się tylko te układy, w których wystąpiło przekroczenie temperatury podczas ostatniego pomiaru. Zamiast korzystać z tej funkcji, alarm można napisać zwyczajnie, porównując w kodzie programu bieżący wynik pomiaru z zadaną wartością. Wartość ta może być zapisana na stałe w kodzie, może być wczytywana z klawiatury lub też ustawiana w inny sposób. Można się tutaj wzorować na zmierzchowym wyłączniku światła z lekcji o komparatorze analogowym oraz o przetworniku analogowo-cyfrowym. Ze względu na ograniczoną ilość miejsca nie będziemy analizować konkretnego kodu.

```
#include <avr/io.h>
#include "lcd.h"
#include "1wire.h"

int main(void) {
 lcdInit();
 lcdInitPrintf();
 while(1){
  uint8_t rom[] = {0, 0, 0, 0, 0, 0, 0, 0};
  uint8_t devices = 0;
  int8_t lastDeviation = 0;
  do {
   lastDeviation = oneWireSearchRom(rom, lastDeviation);
   if (lastDeviation > -1) devices++;
  } while (lastDeviation > 0);
  lcdGotoXY(0, 0);
  printf("%3d", devices);
 }
}
```

Listing 1

Czwarte zadanie polegało na zmianie rozdzielczości termometru. Zgodnie z podpowiedzią w treści zadania, wykonujemy to komendą Write Scratchpad, po której wysyłamy trzy bajty. W trzecim bajcie trzeba wyzerować bity 5 i 6, aby ustawić rozdzielczość na 9 bitów. Można więc po prostu po komendzie Write Scratchpad wysłać trzy bajty o wartości zero. Aby jednak nasz kod był bardziej uniwersalny, napiszmy funkcję, która będzie ustawiała wszystkie cztery rozdzielczości (9–12 bitów). Przykład przedstawiony jest na **listingu 3**.

Aby przestawić nasz termometr na rozdzielczość 9 bitów, dodajemy przed główną pętlą wywołanie funkcji setResolution() z parametrem 9. Aby kod się skompilował, przed funkcją main() trzeba umieścić deklarację funkcji setResolution() oraz definicję makra komendy Write Scratchpad:

```
#define DS18B20_WRITE_SCRATCHPAD 0x4E
```

Testując program, zauważymy, że temperatura nie zmienia się ze skokiem 0,0625 stopnia, ale 0,5 stopnia. Należy pamiętać, że nasza funkcja setResolution() wysyłając zera, w pierwszym i drugim bajcie zeruje ustawienia alarmu. Gdyby ktoś chciał korzystać z funkcji alarmu w DS18B20, musi tak zmodyfikować funkcję, aby ustawiała też odpowiednio próg alarmu i nie nadpisywała go zerami.

W zadaniu ostatnim zastosowane miało być aktywne oczekiwanie na zakończenie pomiaru temperatury. Zamiast czekać z góry określony okres, program miał odpytywać układ DS18B20, czy już zakończył pomiar. Realizowane jest to przez ciągły odczyt jednego bitu. Jeśli uda się odczytać wartość 1, pomiar jest zakończony. Zmiana w kodzie programu sprowadza się do jednej linijki. Zamiast

```
_delay_ms(750);
```

piszemy:

```
while (!oneWireReceiveBit());
```

Konieczne może być ewentualnie dodanie deklaracji funkcji oneWireReceiveBit() do pliku 1wire.h, jeśli jej tam brakuje. Przy aktywnym oczekiwaniu szybkość odczytu temperatury będzie zmieniała się wraz z ustawioną rozdzielczością: mniejsza rozdzielczość da szybsze odczyty.

Rys. 1

Standard I²C

Kontynuujemy poznawanie standardów komunikacyjnych. Tym razem jest to I²C opracowany przez firmę Philips (obecnie NXP). W przeciwieństwie do 1-Wire, jest powszechnie obsługiwany sprzętowo przez mikrokontrolery, także przez ATmega32. Jeśli jednak zajrzymy do dokumentacji naszego procesora, nie znajdziemy nigdzie nazwy I²C. Do 2006 roku korzystanie z tego

```
#include <avr/io.h>
#include <util/delay.h>
#include "led.h"
#include "1wire.h"

#define DS18B20_CONVERT_T 0x44
#define DS18B20_READ_SCRATCHPAD 0xBE

float getTemperature();

struct scratchpad_struct {
 int16_t temperature;
 int16_t temperatureAlarm;
 uint8_t config;
 int8_t reserved[3];
 uint8_t crc;
};

int main(void){
 ledInit();
 while(1){
  float t = getTemperature();
  ledDispNumber(t, "%f", 0);
 }
}

float getTemperature() {
 oneWireReset();
 oneWireSendByte(OW_SKIP_ROM);
 oneWireSendByte(DS18B20_CONVERT_T);
 _delay_ms(750);
 oneWireReset();
 oneWireSendByte(OW_SKIP_ROM);
 oneWireSendByte(DS18B20_READ_SCRATCHPAD);
 struct scratchpad_struct scratchpad;
 uint8_t * byte = (uint8_t *) &scratchpad;
 for (uint8_t i = 0; i < sizeof(scratchpad); i++)
 byte[i] = oneWireReceiveByte();
 return scratchpad.temperature / 16.0;
}
```

Listing 2

```
void setResolution(uint8_t bits) {
 oneWireReset();
 oneWireSendByte(OW_SKIP_ROM);
 oneWireSendByte(DS18B20_WRITE_SCRATCHPAD);
 oneWireSendByte(0);
 oneWireSendByte(0);
 switch (bits) {
  case 9:
   oneWireSendByte(0b00000000);
   break;
  case 10:
   oneWireSendByte(0b00100000);
   break;
  case 11:
   oneWireSendByte(0b01000000);
   break;
  case 12:
   oneWireSendByte(0b01100000);
   break;
 }
}
```

Listing 3

standardu wymagało bowiem opłat licencyjnych, a opłaty za przyznawanie adresów producenci układów scalonych muszą ponosić do dzisiaj. W związku z tym Atmel wraz z innymi firmami stworzył standard o nazwie TWI (Two Wire Interface), który bazuje na I²C i niewiele się od niego różni. Informacje o obsłudze I²C znajdziemy więc w dokumentacji mikrokontrolerów Atmela pod hasłem TWI. Podobnie z I²C wywodzą się również standardy SMBus i PMBus. Ten pierwszy jest zapewne znany tym Czytelnikom, którzy uważnie zapoznawali się z instrukcjami do płyt głównych. Znajdziemy go tam, ponieważ używany jest do komunikacji z układami monitorującymi temperaturę, napięcia zasilające, zegar czy zamknięcie pokrywy laptopa.

Nazwa Two Wire Interface sugeruje nam, że oprócz masy do łączenia układów używane są dwa przewody. Są to SDA (Serial DAta) i SCL (Serial CLock). Pierwszym przesyłane są dane, drugim sygnał zegarowy. Podobnie jak w 1-Wire, obie linie są podciągnięte do plusa zasilania rezystorami o wartości kilku kiloomów, a podłączone układy mogą ściągać obie linie do masy za pomocą wyjść z otwartym drenem. Częstotliwość sygnału zegarowego mogła oryginalnie wynosić do 100 kHz, ale podnoszono ją w kolejnych wersjach standardu. Maksymalna częstotliwość z jaką można wymieniać dane z określonym układem scalonym, podana jest w jego dokumentacji. ATmega32 obsługuje transmisję I²C z częstotliwością do 400 kHz. W typowych zastosowaniach przez I²C przesyła się niewielkie ilości danych i duże prędkości nie są konieczne.

Podobnie jak w 1-Wire, rozróżnia się urządzenia master i slave. Master generuje sygnał zegarowy, rozpoczyna i kończy transmisję, wyznacza kierunek przesyłu danych oraz adresuje układy slave. Różnica jest taka, że układów master może być więcej niż jeden. Rodzi to oczywiście problem konfliktów, gdy dwa układy master rozpoczną transmisję w tej samej chwili. Na szczęście I²C pozwala rozwiązać ten problem, o czym za chwilę.

Rozpoczęcie i zakończenie transmisji jest sygnalizowane przez mastera tzw. sygnałami START i STOP. Po wystąpieniu sygnału START linia jest zajęta i do momentu wystąpienia sygnału STOP żaden inny master nie może kontrolować linii I²C. Oba sygnały generowane są poprzez zmianę stanu linii SDA w momencie, gdy SCL jest w stanie wysokim. Odwrotnie jest, gdy transmitowane są normalne bity: stan linii SDA może się zmieniać w momencie, gdy SCL jest w stanie niskim i musi zachowywać swój stan przez cały czas trwania stanu wysokiego na SCL.

Ponieważ tak jak w przypadku 1-Wire mamy linie podciągnięte do plusa oraz wyjścia z otwartym drenem, na obu liniach realizowana jest sprzętowo operacja logiczna AND. Wystarczy bowiem tylko jedno urządzenie ściągające w danej chwili linię do masy, aby wystąpił na niej niski stan logiczny. Jeśli więc jeden master wystawia wysoki stan logiczny, a mimo to na danej linii jest stan niski, wówczas master ten wie, że inny master jest właśnie aktywny. Musi wtedy zwolnić linię i poczekać na jej uwolnienie. Pozwala to na rozwiązanie konfliktów w sieciach z więcej niż jednym masterem. Proces decydowania, który master może korzystać z linii nazywany jest arbitrażem.

Po rozpoczęciu transmisji sygnałem START, master wysyła 9-bitowy pakiet adresowy. Najpierw transmitowane jest 7 bitów określających adres urządzenia slave, do którego master kieruje transmisję. Ponieważ bitów jest 7, daje to 128 możliwych adresów. Adres 0000000 jest adresem rozgłoszeniowym, oznaczającym, że transmisja jest kierowana do wszystkich urządzeń. W związku z tym na magistrali I²C może znajdować się do 127 urządzeń slave. Nowsze wersje standardu I²C dopuszczają adresowanie 10-bitowe, ale nie jest ono obsługiwane przez nasz mikrokontroler. Adres transmitowany jest od najstarszego bitu do najmłodszego. Po adresie master wysyła bit R/W, w którym informuje slave, czy będzie do niego wysyłać dane, czy też odbierać. R/W czytamy jako „read, not write", czyli jedynka oznacza odczyt/odbiór, a zero oznacza zapis/nadawanie. Ostatni transmitowany bit to ACK, czyli bit potwierdzenia. Master wysyła tu zawsze jedynkę, a urządzenie slave zgłasza się, ściągając linię SDA do masy. Gdy master widzi, że linia SDA jest w stanie niskim, choć wysyłał jedynkę, wówczas ma informację, że slave jest obecny na magistrali I²C, zareagował na swój adres i można kontynuować z nim wymianę danych. Jeśli slave się nie zgłosi, a master chce komunikować się z innym urządzeniem, może wysłać jeszcze raz sygnał START. Nie musi wysyłać sygnału STOP i zwalniać linii.

Adres urządzenia slave znajdziemy w jego dokumentacji. Ponieważ możemy chcieć mieć kilka takich samych urządzeń, wiele układów scalonych obsługujących I²C pozwala ustawiać swój adres w pewnym zakresie. Zwykle mają one w tym celu kilka pinów, które można zwierać do plusa lub do masy i w ten sposób zmienić adres względem adresu bazowego. Wówczas projektując płytkę drukowaną, trzeba zadbać, aby układy jednego typu miały różne kombinacje połączeń. Ewentualnie kombinacje mogą być ustawiane jumperami.

Ponieważ wraz z siedmioma bitami adresu wysyłany jest też bit R/W, można powiedzieć, że każde urządzenie slave ma dwa 8-bitowe adresy: parzysty (z zerem na najmłodszym bicie) do zapisu i nieparzysty (z jedynką na najmłodszym bicie) do odczytu. Należy zwrócić uwagę, że w tej sytuacji bit R/W powoduje przesunięcie bitów adresu w lewo. Jeśli np. układ slave ma swój podstawowy adres o wartości 1, to master chcąc zapisać do niego dane, wyśle bajt 00000010b (dziesiętnie 2), a chcąc odczytać dane, wyśle bajt 00000011b (dziesiętnie 3). Należy o tym pamiętać, bo może być to mylące.

Popatrzmy na konkretny przykład. Układ scalony WM8750 jest kodekiem audio konfigurowanym przez interfejs I²C. Układ ten ma nóżkę CSB pozwalającą przestawić adres układu, dzięki temu na jednej linii I²C mogą działać dwa układy WM8750. Producent w dokumentacji układu opisuje adresowanie za pomocą **tabeli 1.**

Tabela 1

CSB STATE	DEVICE ADDRESS
Low	0011010 (0x34h)
High	0011011 (0x36h)

Jaki adres ma WM8750, gdy nóżka CSB jest zwarta do masy? 7-bitowy adres to 0011010b, czyli dziesiętnie 26, a szesnastkowo 1Ah. W nawiasie jednak mamy podaną liczbę szesnastkową 34h, czyli dziesiętnie 52, a więc dwa razy większą. Liczba w nawiasie nie jest bowiem tylko zamianą podanej liczby dwójkowej na postać szesnastkową, ale uwzględnia też postać 8-bitową, z dodanym bitem R/W. By zaadresować WM8750 do zapisu, master wyśle bajt 00110100b (dziesiętnie 52, szesnastkowo 34h), a do odczytu bajt 00110101b (dziesiętnie 53, szesnastkowo 35h). Przy wysokim stanie logicznym na nóżce CSB, układ będzie można adresować odpowiednio bajtami 36h i 37h.

Przesyłanie danych wygląda podobnie jak przesyłanie adresu, transmitowane jest bowiem również 9 bitów. Tym razem jest to 8 bitów danych i bit potwierdzenia (ACK). Gdy master wysyła dane, slave potwierdza odbiór, zerując bit ACK. Gdy master odbiera dane, także potwierdza odbiór, ale z wyjątkiem ostatniego bajtu. Po zakończeniu odbierania ostatniego bajtu master nie ściąga linii SDA w dół.

Obsługa I²C w ATmega32

Jak wspomnieliśmy, nasz mikrokontroler ma sprzętową obsługę standardu I²C, a raczej jego odmiany zwanej TWI. Wyposażony jest w tym celu w dedykowane piny SDA i SCL, współdzielone z pinami PC1 i PC0. Do zarządzania I²C służy pięć rejestrów:

- TWBR – ustawienie prędkości transmisji (bit rate)
- TWCR – rejestr kontrolny
- TWSR – rejestr statusu i preskalera
- TWDR – rejestr danych
- TWAR – rejestr adresu

Do rejestru TWBR wpisujemy liczbę, która będzie decydowała o prędkości transmisji. Zależność prędkości od wartości w TWBR wyraża się wzorem:

$$\text{SCL frequency} = \frac{\text{CPU Clock frequency}}{16+2(\text{TWBR})\cdot 4^{\text{TWPS}}}$$

TWPS to liczba, jaką tworzą bity TWPS1 i TWPS0 w rejestrze TWSR. Jako że bity są dwa, liczba ta może mieć wartość od 0 do 3. Zgodnie z tą liczbą potęgowana jest liczba 4 **(tabela 2**). Ta potęga czwórki daje nam preskaler, przez który mnożymy podwójną wartość rejestru TWBR, a na końcu dodajemy 16. W ten sposób otrzymujemy mianownik, przez który dzielona jest częstotliwość taktowania mikrokontrolera.

Załóżmy, że taktujemy mikrokontroler kwarcem 16 MHz, do TWBR wpisaliśmy liczbę 100, a bity TWPS1 i TWPS0 ustawiliśmy na jedynki. W mianowniku otrzymamy więc 16 + 200 * 64 = 12816. Na linii SCL pojawi się więc częstotliwość ok. 1248Hz. Przy wyzerowanych bitach preskalera w mianowniku będzie wartość 16 + 200 = 216, co da częstotliwość 74074Hz. Przy kwarcu 16MHz częstotliwość 100kHz osiągniemy, zerując bity preskalera, a TWBR ustawiając na 72. 100kHz wystarczy do praktycznie każdego zastosowania. Jeśli byłaby potrzebna wyższa prędkość, np. 400kHz, dobrze jest wtedy zmniejszyć rezystory podciągające do 2,7kΩ.

Rejestr TWSR oprócz bitów preskalera zawiera także bity statusu, które informują o stanie kontrolera I²C. Ponieważ jest to 5 starszych bitów, noszą one nazwy TWS7... TWS3. Odczytując rejestr TWSR, możemy sprawdzić, w jakim stanie znajduje się aktualnie transmisja I²C, czy np. nie wystąpił błąd. Ze względu na obecność bitów preskalera w tym samym rejestrze, status trzeba odczytywać z maską 11111000b. Po nałożeniu maski (funkcją AND) otrzymamy liczbę będącą kodem aktualnego statusu. Możliwe kody statusów przedstawiono w **tabeli 3**.

Rejestr kontrolny TWCR zawiera następujące bity:

- TWINT – flaga przerwania, 1 oznacza zakończenie danej operacji. Uwaga: po zakończeniu operacji kontroler I²C czeka na wyczyszczenie tej flagi przez zapis jedynki. Do czasu wyczyszczenia flagi działanie kontrolera jest wstrzymane. Można wtedy modyfikować rejestry TWAR, TWSR i TWDR. Po ustawieniu bitu TWINT kontroler I²C rozpoczyna pracę i natychmiast zeruje ten bit.
- TWEA – włącza potwierdzanie odbioru danych przez zerowanie bitu ACK
- TWSTA – jedynka generuje sygnał START, po jego wygenerowaniu bit należy wyzerować
- TWSTO – jedynka generuje sygnał STOP, bit zeruje się automatycznie po zakończeniu generowania STOP-u
- TWWC – flaga kolizji, jest ustawiana, gdy nastąpi próba zapisu do rejestru TWDR, zanim wysłany zostanie poprzedni bajt
- TWEN – włącza obsługę I²C (TWI) w mikrokontrolerze, powoduje przejęcie przez podsystem TWI kontroli nad pinami SDA i SCL (PC1 i PC0)
- TWIE – włącza przerwania TWI

Gdy wysyłamy dane, zapisujemy je do rejestru TWDR. Z zapisem kolejnego bajtu czekamy, aż poprzedni zostanie wysłany. TWDR używany jest także przy odbiorze, znajdziemy w nim ostatni odebrany bajt. Uwaga: wysłanie bajtu umieszczonego w TWDR nie następuje w chwili zapisu do tego rejestru, ale po ustawieniu bitu TWINT w rejestrze TWCR.

Tabela 2

TWPS1	TWPS0	TWPS	Preskaler (4^{TWPS})
0	0	0	1
0	1	1	4
1	0	2	16
1	1	3	64

Biblioteka I²C

Wiedząc już, jak wygląda obsługa I²C w naszym mikrokontrolerze, możemy napisać bibliotekę do obsługi tego standardu. Zaczniemy od użytych w kodzie makr. Zawierają one definicje kodów statusów, maskę bitową dla statusów, wartość zwracaną przez funkcje przy braku błędu oraz ustawienie prędkości transmisji. Oto **listing 4**:

```
#define I2C_START 0x08
#define I2C_START_REPEATED 0x10
#define I2C_ADDR_WRITE_ACK 0x18
#define I2C_ADDR_WRITE_NACK 0x20
#define I2C_DATA_SENT_ACK 0x28
#define I2C_DATA_SENT_NACK 0x30
#define I2C_ARBITRATION_LOST 0x38
#define I2C_ADDR_READ_ACK 0x40
#define I2C_ADDR_READ_NACK 0x48
#define I2C_DATA_RECEIVED_ACK 0x50
#define I2C_DATA_RECEIVED_NACK 0x58

#define I2C_STATUS_MASK 0b11111000
#define I2C_OK 0
#define F_SCL 100000
```

Pierwsza funkcja z naszej biblioteki będzie służyć do włączenia kontrolera I²C. Jest to bardzo proste, bo sprowadza się do ustawienia bitu TWEN w rejestrze TWCR. Przy inicjalizacji ustawiamy też prędkość transmisji. Jeśli prędkość taktowania mikrokontrolera mamy zdefiniowaną w makrze F_CPU, a żądaną prędkość komunikacji I²C w makrze F_SCL, to w funkcji zostanie obliczona odpowiednia wartość rejestru TWBR. Funkcja nie modyfikuje domyślnych ustawień preskalera (TWPS=0), ponieważ praktycznie nigdy nie ma takiej konieczności. Nie ma też zwykle potrzeby definiować F_SCL jako wartości innej niż 100 000. Oto **listing 5**:

```
void i2cInit(void) {
 TWCR = _BV(TWEN);
 TWBR = (F_CPU/F_SCL - 16) / 2;
}
```

Kolejna funkcja, której potrzebujemy, to funkcja wysyłająca sygnał START (**listing 6**). Aby go wygenerować, musimy ustawić bity TWSTA i TWINT. Następnie czekamy na zakończenie sygnału, obserwując bit TWINT. Tutaj funkcja mogłaby się zakończyć, ale dobrze byłoby wiedzieć, czy wysłanie START się powiodło. W tym celu sprawdzamy, czy w rejestrze TWSR znajduje się kod oznaczający wysłanie START lub powtórzonego START. Jeśli będzie tam inny status, zostanie to potraktowane jako błąd. Przed sprawdzeniem kodu statusu, funkcją AND zerowane są 3 najmłodsze bity, aby nie przeszkadzały w sprawdzeniu wartości znajdującej się na 5 starszych bitach. O tym, czy funkcja wykonała się poprawnie, może ona informować na różne sposoby. Tutaj założono, że w przypadku sukcesu zwracane jest zero (makro I2C_OK), a w przypadku statusu innego niż oczekiwany, zwracany jest odczytany status.

```
uint8_t i2cStart(void) {
 TWCR = _BV(TWEN) | _BV(TWSTA) | _BV(TWINT);
 while (!(TWCR & _BV(TWINT)));
 uint8_t status = TWSR & I2C_STATUS_MASK;
 if (status == I2C_START || status == I2C_START_REPEATED) {
  return I2C_OK;
 } else {
  return status;
 }
}
```

Listing 6

Generowanie sygnału STOP jest prostsze, ponieważ jego poprawne wy generowanie nie jest sygnalizowane żadnym statusem. Funkcja więc ogranicza się do zainicjowania sygnału STOP ustawiając bity TWSTO i TWINT oraz do oczekiwania na jego zakończenie. Tutaj jest też taka różnica, że zakończenie STOP-u nie jest sygnalizowane bitem TWINT, ale

Tabela 3

Kod statusu	Opis
08h	Wysłano START
10h	Wysłano ponowny START
18h	Wysłano adres do zapisu i otrzymano ACK
20h	Wysłano adres do zapisu, ale nie otrzymano ACK
28h	Wysłano bajt, otrzymano ACK
30h	Wysłano bajt, ale nie otrzymano ACK
38h	Stracono arbitraż
40h	Wysłano adres do odczytu i otrzymano ACK
48h	Wysłano adres do odczytu, ale nie otrzymano ACK
50h	Odebrano bajt, odesłano ACK
58h	Odebrano bajt, ale nie odesłano ACK
F8h	Brak statusu: operacja jeszcze niezakończona lub wysłano STOP

wyzerowaniem się bitu TWSTO. Pokazuje to **listing 7**:

```
void i2cStop(void) {
 TWCR = _BV(TWEN) | _BV(TWSTO) | _BV(TWINT);
 while (TWCR & _BV(TWSTO));
}
```

Wysyłanie bajtów w ATmega32 wygląda tak samo, niezależnie od tego, czy wysyłamy adres, czy dane: wpisujemy bajt do TWDR, ustawiamy bit TWINT w TWCR i czekamy na pojawienie się jedynki w bicie TWINT. Jednak w zależności od tego, co wysłaliśmy, różne będą statusy. Aby sobie z tym poradzić, podzielimy wysyłanie na trzy funkcje: podstawową funkcję wysyłającą bajt oraz korzystające z niej funkcje do wysyłania adresu i danych (**listing 8**).

Funkcja i2cSend() będzie funkcją niższego poziomu, raczej nieprzeznaczoną do wywoływania z głównego programu. Oczywiście można z niej korzystać, ale nie będzie ona zwracać statusu. Ponadto korzystanie z dedykowanych funkcji i2cSendAddress() i i2cSendData() sprawi, że główny program będzie bardziej przejrzysty. Będzie bowiem widać, w którym miejscu wysyłany jest adres, a w którym zwykłe dane. Funkcja i2cSendAddress() po wywołaniu i2cSend() sprawdza, czy odebrany został status oznaczający potwierdzenie (ACK) od układu slave. Ponieważ są oddzielne statusy dla adresów zapisu i odczytu, sprawdzamy, czy adres był adresem do zapisu, czy do odczytu. Jak pamiętamy, decyduje o tym jedynka na najmłodszym bicie. Stosujemy więc operację AND na adresie i liczbie 1 (binarnie 00000001).

Do odbierania danych wystarczy nam jedna funkcja (**listing 9**). Ponieważ przy odbieraniu ostatniego bajtu master nie powinien wysyłać ACK, funkcja i2cReceive() w pierwszym parametrze pobiera informację o tym, czy odbierany bajt jest ostatnim, jaki master chce odebrać. Jeśli parametr ten jest różny od zera (odbierany jest ostatni bajt), nie jest ustawiany bit TWEA i ATmega32 nie potwierdza odbioru. Drugi parametr to wskaźnik na miejsce w pamięci, do którego ma być zapisana odebrana wartość. Co prawda funkcja mogłaby po prostu zwracać tę wartość za pomocą return, ale przyjęliśmy, że nasze funkcje zwracają kod błędu. Odebrana dana musi być więc przekazana do funkcji wywołującej w inny sposób.

Jak Czytelnicy być może zauważyli, w naszych funkcjach zapisujemy całą nową wartość rejestru TWCR, zamiast ustawiać wybrane bity, jak to zwykle robiliśmy. Jest np. za każdym razem ustawiany bit TWEN, choć gdy raz go ustawimy, to potem sam się nie wyzeruje i nie ma potrzeby go ustawiać. Wynika to z faktu, że TWCR jest nie tylko rejestrem konfiguracyjnym, ale też wyzwala określone czynności za pomocą bitu TWINT. Trzeba też odpowiednio manipulować bitami TWSTA i TWEA. Wygodniej jest więc ustawiać całą nową wartość niż na podstawie bieżącej wartości ustawiać lub zerować niektóre bity.

Układ MCP79410

MCP79410 to zegar czasu rzeczywistego (Real Time Clock – RTC). Dzięki podłączonemu rezonatorowi kwarcowemu oraz podtrzymaniu bateryjnemu może służyć jako niezależne źródło aktualnego czasu i daty. Współpracujący mikrokontroler nie musi więc samodzielnie odmierzać czasu i obliczać takich danych jak minuty czy miesiące, może po prostu odpytywać RTC o bieżący czas. Oczywiście RTC musi być najpierw zainicjowany aktualnym czasem, np. przez wpisanie go z klawiatury lub pobranie z innego źródła. Zasilanie bateryjne podtrzymuje działanie RTC w czasie braku zasilania podstawowego. Zapewnione jest wtedy odmierzanie czasu i zachowanie zawartości wewnętrznej pamięci SRAM, natomiast komunikacja I²C jest wtedy wyłączona.

Nasz układ ma też pamięć EEPROM. Jest to pamięć nieulotna, zachowująca swoją zawartość przy braku zasilania, także bateryjnego. EEPROM dostępny jest pod innym adresem I²C niż RTC. Można powiedzieć, że z punktu widzenia I²C układ MCP79410 to dwa układy w jednej obudowie. W konwencji 8-bitowej RTC adresujemy bajtem 1101111Xb, a EEPROM bajtem 1010111Xb (X – bit R/W). Innymi słowy, zapis RTC jest pod adresem DEh a odczyt pod DFh, natomiast zapis EEPROM pod AEh a odczyt pod AFh.

Niezależnie od tego, czy będziemy korzystać z RTC, czy z EEPROM-u, układ MCP79410 widoczny jest dla nas jako pamięć. Cała komunikacja z nim opiera się na zapisie i odczycie odpowiednich komórek/rejestrów, które mają swoje określone adresy. Rejestry kontrolujące pracę RTC mają adresy od 00h do 1Fh. Natomiast pod adresami od 20h do 5Fh znajduje się pamięć SRAM. Są to 64 bajty, które można wykorzystać do dowolnych celów. Pamięć ta jest podtrzymywana bateryjnie. Jeśli natomiast zaadresujemy na poziomie I²C funkcję EEPROM, pod adresami 00h – 7Fh będziemy mieć do dyspozycji 128 bajtów pamięci EEPROM, a więc nieulotnej, nawet bez baterii. Jest też 8 bajtów chronionych pod adresami F0h – F7h oraz rejestr statusowy pod adresem FFh. Bajty chronione mają zabezpieczenie przed zapisem. Aby móc zapisać do nich nowe wartości, trzeba najpierw wpisać bajt 55h i zaraz potem AAh do rejestru EEUNLOCK. Co ciekawe, rejestr ten jest rejestrem RTC, znajdującym się pod adresem 09h.

Przyjrzyjmy się rejestrom RTC. Mapa pierwszych 6 rejestrów przedstawiona jest w **tabeli 4**. Rejestr RTCSEC przechowuje liczbę sekund na bitach 6...0. Tak jak i w wielu innych układach RTC, liczba ta jest przechowywana w kodzie BCD (binary coded decimal), czyli jako liczba dziesiętna zapisana dwójkowo. Brzmi to groźnie, ale polega na tym, że oddzielnie zapisane są jedności a oddzielnie dziesiątki. Jedności zapisane są w bitach 3...0, noszących nazwy SECONE3...0, a dziesiątki w bitach 6...4, nazwanych SECTEN2...0. Jest to wygodne, gdy chcemy tylko wyświetlać czas, a nie potrzebujemy wykonywać na nim obliczeń. Nie trzeba wtedy z jednej wartości tworzyć dwóch cyfr, bo od razu mamy wartości dla cyfry dziesiątek i cyfry jedności. Analogicznie mamy przechowywane pozostałe wartości czasu: minuty w RTCMIN, godziny w RTCHOUR, dzień miesiąca w RTCDATE, miesiąc w RTCMTH i rok w RTCYEAR. Dzień tygodnia w rejestrze jest oczywiście jednocyfrowy i wystarczają dla niego 3 najmłodsze bity z rejestru RTCWKDAY. Choć nasz RTC dba o zmianę dnia tygodnia wraz ze zmianą dnia miesiąca o każdej północy, to nie oblicza dnia tygodnia. Jeśli więc chcemy odczytywać dzień tygodnia, to musimy go odpowiednio ustawić przy ustawianiu daty. Dzień tygodnia reprezentowany jest jako liczba z zakresu 1–7, przy czym nie ma przypisania konkretnej liczby do konkretnego dnia. Od użytkownika zależy np., czy 1 będzie traktować jako poniedziałek, czy jako

```
void i2cSend(uint8_t data) {
 TWDR = data;
 TWCR = _BV(TWEN) | _BV(TWINT);
 while (!(TWCR & _BV(TWINT)));
}
uint8_t i2cSendAddress(uint8_t address) {
 i2cSend(address);
 uint8_t status = TWSR & I2C_STATUS_MASK;
 if (address & 0b00000001) {
  return (status == I2C_ADDR_READ_ACK) ? I2C_OK : status;
 } else {
  return (status == I2C_ADDR_WRITE_ACK) ? I2C_OK : status;
 }
}
uint8_t i2cSendData(uint8_t data) {
 i2cSend(data);
 uint8_t status = TWSR & I2C_STATUS_MASK;
 return (status == I2C_DATA_SENT_ACK) ? I2C_OK : status;
}
```

Listing 8

```
uint8_t i2cReceive(uint8_t last, uint8_t * data){
 TWCR = last ? _BV(TWEN) | _BV(TWINT) : _BV(TWEN) | _BV(TWINT) | _BV(TWEA);
 while (!(TWCR & _BV(TWINT)));
 *data = TWDR;
 uint8_t status = TWSR & I2C_STATUS_MASK;
 if (last) {
  return (status == I2C_DATA_RECEIVED_NACK) ? I2C_OK : status;
 } else {
  return (status == I2C_DATA_RECEIVED_ACK) ? I2C_OK : status;
 }
}
```

Listing 9

niedzielę. Rok przechowywany jest na dwóch cyfrach. Jeśli chcemy wyświetlać cztery cyfry, musimy w naszym programie sami dodawać 20 z przodu.

Na mapie rejestrów zauważymy, że producent układu MCP79410 poupychał też kilka innych bitów w rejestrach czasu. I tak w rejestrze RTCSEC znajduje się bit ST. Służy on do uruchamiania/wstrzymywania odmierzania czasu. Gdy jest ustawiony na 0, RTC jest zatrzymany i wartości w rejestrach czasu nie zmieniają się. RTC dobrze jest zatrzymywać podczas ustawiania czasu, by uniknąć sytuacji, że jakiś rejestr zmieni się w połowie zapisywania nowych wartości.

Bit 12/24 w rejestrze RTCHOUR przełącza między formatem 12- i 24-godzinnym. Jeśli wybierzemy format 12-godzinny, wpisując do niego 1, bit 5 jest bitem AM/PM (0 – przed południem, 1 – po południu). Przy formacie 24-godzinnym bit 5 jest dodatkowym bitem dziesiątek godzin, dzięki któremu można przechowywać godziny 20–23.

Jedynka w bicie OSCRUN w rejestrze RTCWKDAY informuje o prawidłowej pracy rezonatora kwarcowego. Z kolei jedynka w bicie PWRFAIL informuje o wystąpieniu zaniku zasilania i zapisania tego momentu w rejestrach 18h–1Bh. Aby móc odnotować kolejny zanik zasilania, trzeba wyzerować ten bit. Bit VBATEN włącza korzystanie z podtrzymania bateryjnego. Należy pamiętać o ustawieniu go na 1, jeśli chcemy korzystać z baterii. Bit LPYR w rejestrze RTCMTH informuje, czy bieżący rok jest rokiem przestępnym.

Adres	Rejestr	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
00h	RTCSEC	ST	SECTEN2	SECTEN1	SECTEN0	SECONE3	SECONE2	SECONE1	SECONE0
01h	RTCMIN		MINTEN2	MINTEN1	MINTEN0	MINONE3	MINONE2	MINONE1	MINONE0
02h	RTCHOUR		12/24	AM/PM HRTEN1	HRTEN0	HRONE3	HRONE2	HRONE1	HRONE0
03h	RTCWKDAY			OSCRUN	PWRFAIL	VBATEN	WKDAY2	WKDAY1	WKDAY0
04h	RTCDATE			DATETEN1	DATETEN0	DATEONE3	DATEONE2	DATEONE1	DATEONE0
05h	RTCMTH			LPYR	MTHTEN0	MTHONE3	MTHONE2	MTHONE1	MTHONE0
06h	RTCYEAR	YRTEN3	YRTEN2	YRTEN1	YRTEN0	YRONE3	YRONE2	YRONE1	YRONE0

Tabela 4

Test komunikacji

Resztę funkcji układu MCP79410 omówimy w następnej lekcji, teraz za to skupimy się na praktycznym wykorzystaniu pozyskanych wiadomości o RTC. Jak wiemy, potrzebujemy odczytywać i zapisywać jego rejestry. Jak to wykonać z poziomu I²C? Żeby MCP79410 wiedział, do którego rejestru chcemy się odwołać, musimy najpierw wysłać jego adres. A więc niezależnie od tego, czy będziemy zapisywać, czy odczytywać, musimy wykonać operację zapisu/wysyłania adresu. Następnie, jeśli wykonujemy zapis do rejestru/ów, wysyłamy bajty danych. Układ MCP79410 będzie automatycznie zwiększał adres i kolejne bajty będą trafiały do kolejnych rejestrów. Kończymy sygnałem STOP. Natomiast jeśli chcemy odczytywać, to po wysłaniu adresu rejestru musimy przełączyć się na odczyt. Trzeba więc wysłać powtórny START, wysłać adres do odczytu a następnie odebrać potrzebną liczbę bajtów. Kończymy oczywiście sygnałem STOP. Na podstawie tego możemy napisać funkcje do odczytu i zapisu pojedynczych rejestrów oraz całych grup (**listing 10**).

```
void rtcReadRegister(uint8_t reg, uint8_t * data) {
 i2cStart();
 i2cSendAddress(MCP79410_RTC_WRITE_ADDRESS);
 i2cSendData(reg);
 i2cStart();
 i2cSendAddress(MCP79410_RTC_READ_ADDRESS);
 i2cReceive(1, data);
 i2cStop();
}
void rtcWriteRegister(uint8_t reg, uint8_t data) {
 i2cStart();
 i2cSendAddress(MCP79410_RTC_WRITE_ADDRESS);
 i2cSendData(reg);
 i2cSendData(data);
 i2cStop();
}
void rtcReadRegisters(uint8_t reg, uint8_t * data, uint8_t size) {
 i2cStart();
 i2cSendAddress(MCP79410_RTC_WRITE_ADDRESS);
 i2cSendData(reg);
 i2cStart();
 i2cSendAddress(MCP79410_RTC_READ_ADDRESS);
 for (uint8_t i = 0; i < size; i++) {
  i2cReceive(i == size - 1, data++);
 }
 i2cStop();
}
void rtcWriteRegisters(uint8_t reg, uint8_t * data, uint8_t size) {
 i2cStart();
 i2cSendAddress(MCP79410_RTC_WRITE_ADDRESS);
 i2cSendData(reg);
 for (uint8_t i = 0; i < size; i++) {
  i2cSendData(data[i]);
 }
 i2cStop();
}
```

Listing 10

Funkcje są dosyć proste i bazują na podanym wyżej algorytmie. Jako pierwszy parametr przekazywany jest numer rejestru, do którego chcemy się odwołać lub numer pierwszego rejestru, jeśli odczytujemy/zapisujemy grupę rejestrów. Drugi parametr to wskaźnik na dane. Wyjątkiem jest wysłanie pojedynczego bajtu, wtedy nie ma konieczności przekazywania go przez wskaźnik. Przy przesyłaniu grupy rejestrów podajemy jeszcze ich liczbę.

Na koniec napiszmy najprostszy program, który będzie włączał RTC i wyświetlał liczbę sekund na LCD. Aby włączyć RTC, musimy ustawić bit ST w rejestrze RTCSEC. Potem w pętli będziemy wyświetlać sekundy pobrane z tego samego rejestru. Kod przykładowej funkcji main() został przedstawiony na **listingu 11**.

Aby wyświetlić sekundy, pobierana jest najpierw wartość rejestru RTCSEC. Potem wydzielane są z niego dziesiątki i jedności sekund przy wykorzystaniu masek bitowych, aby wyciągnąć odpowiednie bity. Przy dziesiątkach stosujemy przesunięcie w prawo, ponieważ bity dziesiątek w rejestrze RTCSEC są na pozycjach 6–4, a więc za bardzo w lewo. Bez przesunięcia mielibyśmy wartość dziesiątek pomnożoną przez 16 (24). Cyfry dziesiątek i jedności są następnie wyświetlane na LCD. Oprócz wyświetlacza należy też oczywiście podłączyć nasz RTC, łącząc pin PC0 z pinem SCL, a PC1 z pinem SDA na złączu RTC.

Pojawiająca się na ekranie i zmieniająca co sekundę liczba sekund potwierdza poprawne działanie programu. W przypadku problemów warto sprawdzić statusy zwracane przez funkcje z naszej biblioteki I²C. Statusy można np. wyświetlić sobie na LCD lub obejrzeć pod debuggerem, jeśli mamy JTAG.

```
int main(void) {
 i2cInit();
 lcdInit();
 lcdInitPrintf();
 rtcWriteRegister(MCP79410_RTCSEC, 0b10000000);
 while(1){
  uint8_t rtcsec;
  rtcReadRegister(MCP79410_RTCSEC, &rtcsec);
  uint8_t secondsTens = (rtcsec & 0b01110000) >> 4;
  uint8_t secondsOnes = (rtcsec & 0b00001111);
  lcdGotoXY(0, 0);
  printf("%d%d", secondsTens, secondsOnes);
 }
}
```

Listing 11

Zadania

Jak zwykle zachęcam do jak największej liczby samodzielnych eksperymentów. W tej lekcji pomocne będą następujące ćwiczenia:

1. Wyszukiwarka podłączonych układów I²C
2. Zegar z kalendarzem ustawiany z klawiatury, z podtrzymaniem bateryjnym

W materiałach dodatkowych znajdują się listingi z rozwiązaniami zadań z poprzedniego odcinka, program testowy wyświetlający sekundy z RTC oraz karta katalogowa układu MCP79410.

Grzegorz Niemirowski
grzegorz@grzegorz.net