Kurs AVR – lekcja 15

Rozwiązania zadań z ostatniego odcinka

W poprzednim odcinku pierwsze zadanie domowe polegało na napisaniu wyszukiwarki układów I²C. Ponieważ na linii I²C może znajdować się do 127 układów, możemy po prostu odpytywać kolejne adresy. Jeśli po wysłaniu adresu odbierzemy bit ACK, będzie to świadczyło o tym, że mamy podłączony układ slave o tym adresie. Nasz program będzie więc bardzo prosty (**listing 1**).

```
#include <avr/io.h>
#include "i2c.h"
#include "lcd.h"
int main(void) {
 i2cInit();
 lcdInit();
 lcdInitPrintf();
 for (uint8_t i = 1; i < 128; i++) {
  i2cStart();
  if (i2cSendAddress(i << 1) == I2C_OK) {
   printf("%d ", i);
  }   }
 i2cStop();
 while(1){
 }      }
```

Przed wysłaniem każdego adresu wysyłany jest sygnał START, a po wysłaniu wszystkich adresów wysyłany jest sygnał STOP. Gdy adres zostanie wysłany, sprawdzamy, czy odebrany został bit ACK. Jeśli tak, funkcja i2cSendAddress() zwraca status I2C_OK i adres zostaje wyświetlony na LCD. Ponieważ funkcja i2cSendAddress() pobiera adres w notacji 8-bitowej, standardowy adres 7-bitowy jest przesuwany o 1 bit w lewo za pomocą operatora <<. Najmłodszy bit zostaje tym samym wyzerowany i wysyłane adresy stają się adresami do zapisu. Jeśli do mikrokontrolera mamy podłączony znajdujący się na płytce testowej układ MCP79410, na wyświetlaczu zobaczymy liczby 87 i 111. Jak pamiętamy z poprzedniej lekcji, jest on dostępny do zapisu pod 8-bitowymi adresami DEh (RTC) i AEh (EEPROM). Czy wyświetlone wartości są więc poprawne? Zamieńmy adresy 87 i 111 na 8-bitową postać adresów I²C. Przesunięcie o jeden bit w lewo jest równoznaczne z pomnożeniem przez 2, otrzymamy więc liczby 174 i 222. Szesnastkowo to AEh i DEh, a więc wszystko się zgadza. Adresy możemy wyświetlić od razu w postaci 8-bitowej, szesnastkowej. Wystarczy, że wywołanie funkcji printf() będzie miało postać:

```
printf("%X ", i << 1);
```

Wtedy na wyświetlaczu pojawi się napis AE DE potwierdzający poprawne działanie programu. Uwaga: nasz program nie uwzględnia zawijania linijek tekstu na LCD i przewijania listy, nie wyświetli więc poprawnie więcej niż 5 układów I²C. Jeśli mielibyśmy więcej niż 5 układów, trzeba dopisać kod zapewniający prawidłowe wyświetlanie listy znalezionych układów i przewijanie jej np. za pomocą klawiatury.

Celem drugiego zadania domowego było napisanie zegara z kalendarzem, z podtrzymaniem bateryjnym. Zacznijmy od wyświetlania aktualnego czasu.

W poprzedniej lekcji napisaliśmy kod wyświetlający sekundy, wystarczy więc rozszerzyć go o odczyt pozostałych rejestrów czasu. Najpierw jednak zajmijmy się uporządkowaniem kodu. Mamy bowiem napisane cztery funkcje do zapisu i odczytu rejestrów układu MCP79410. Warto byłoby umieścić je w oddzielnej bibliotece stworzonej specjalnie dla tej kostki. Znalazłyby się tam również funkcje do zarządzania nią, w tym do odczytywania i zapisywania czasu. Utwórzmy więc pliki MCP79410.c i MCP79410.h. Do pliku .c przenieśmy nasze funkcje do zarządzania rejestrami, a do pliku .h ich nagłówki. Definicje makr z adresami MCP79410 także przeniesiemy do pliku nagłówkowego.

Sekundy odczytywaliśmy z pierwszego rejestru (RTCSEC). Aby mieć pełne dane o czasie i dacie, potrzebujemy także sześciu kolejnych rejestrów. Możemy je wczytać do tablicy bajtów:

```
uint8_t registers[7];
rtcReadRegisters(MCP79410_RTCSEC, registers, sizeof(registers));
```

a następnie odwoływać się do nich za pomocą indeksów, np. registers[3], aby uzyskać dostęp do dnia tygodnia. Dla ułatwienia można sobie stworzyć makra dla indeksów, aby do odebranych rejestrów odwoływać się w sposób bardziej czytelny (registers[MCP79410_RTCMTH]). Należy zwrócić uwagę, że w omawianym przypadku odczytujemy rejestry od pierwszego, mającego indeks 0. Jeśli zaczynalibyśmy od innego rejestru, to elementy tablicy miałyby przesunięte indeksy względem indeksów rejestrów i nie moglibyśmy używać tych samych makr. Np. rozpoczynając odczyt od RTCMTH, wartość tego rejestru otrzymalibyśmy w registers[0], co nie byłoby już równoważne registers[MCP79410_RTCMTH], bo makro MCP79410_RTCMTH zdefiniowane byłoby jako 5.

Inny sposób to wykorzystanie zmiennej strukturalnej, w której zadeklarowane zostaną pola odpowiadające poszczególnym rejestrom. Jak każda zmienna jest ona de facto tablicą bajtów i można jej adres przekazać do funkcji rtcReadRegisters(), aby została wypełniona wartościami z rejestrów układu MCP79410.

```
rtcReadRegisters(MCP79410_RTCSEC, (uint8_t *) &rtcDateTimeRegisters, sizeof(rtcDateTimeRegisters));
```

Zadeklarujmy więc w pliku MCP79410.h typ strukturalny:

```
struct RtcDateTimeRegisters {
 uint8_t RTCSEC;
 uint8_t RTCMIN;
 uint8_t RTCHOUR;
 uint8_t RTCWKDAY;
 uint8_t RTCDATE;
 uint8_t RTCMTH;
 uint8_t RTCYEAR;     };
```

Jest to znany już, klasyczny sposób deklaracji typu strukturalnego, gdzie po słowie kluczowym struct podajemy nazwę naszego nowego typu strukturalnego, a następnie w nawiasach klamrowych definiujemy pola struktury. Po nawiasach można jeszcze podać nazwę zmiennej, wtedy od razu oprócz definicji strutkury będziemy mieć też utworzoną zmienną tego typu. Zwykle jednak się tego nie stosuje, bo oddziela się definicję typu od deklaracji zmiennej.

Nasza definicja struktury jest w porządku, ale warto tutaj wspomnieć o możliwościach, jakie daje słowo kluczowe typedef. Służy ono do definiowania nowych typów danych na podstawie już istniejących. Dzięki niemu możemy np. używać typu uint8_t, który normalnie w języku C nie występuje, a został utworzony na podstawie typu unsigned char. W przypadku struktur słowo kluczowe typedef pozwala usunąć pewną uciążliwość: konieczność stosowania słowa kluczowego struct. Używamy go bowiem, nie tylko definiując strukturę, ale też deklarując zmienne strukturalne i funkcje operujące na nich. Zmodyfikujmy definicję naszej struktury:

```
typedef struct {
 uint8_t RTCSEC;
 uint8_t RTCMIN;
 uint8_t RTCHOUR;
 uint8_t RTCWKDAY;
 uint8_t RTCDATE;
 uint8_t RTCMTH;
 uint8_t RTCYEAR;
} RtcDateTimeRegisters;
```

Definiujemy tutaj strukturę bez nazwy, po słowie kluczowym struct od razu zaczyna się definicja jej pól. Następnie za pomocą słowa kluczowego typedef staje się ona nowym typem o nazwie RtcDateTimeRegisters. Zmienną strukturalną zadeklarujemy następująco:

```
RtcDateTimeRegisters rtcDateTimeRegisters;
```

Nie musimy już tutaj pisać struct. W tym przykładzie zmienna została nazwana tak jak typ, z wyjątkiem małej litery na początku, ale oczywiście nazwa może być inna.

Pozostaje jeszcze sprawa przekazywania struktury do funkcji. Przykładowo funkcja rtcReadRegisters() oczekuje typu uint8_t *, czyli wskaźnika na typ uint8_t, operuje bowiem na tablicy bajtów. Musimy więc wykonać rzutowanie:

Operatorem & pobierany jest adres zmiennej rtcDateTimeRegisters, a następnie

przekazywany do funkcji, jakby był adresem zmiennej typu uint8_t. Bez podania typu docelowego w nawiasach kompilator zgłosiłby ostrzeżenie o niezgodności typów. Co prawda ostrzeżenie to nie błąd, a więc kompilator kontynuuje swoją pracę i otrzymujemy skompilowany kod, ale niezgodność typów świadczy często jeśli nie o błędzie, to o sytuacji potencjalnie niebezpiecznej. Tutaj np. funkcja otrzymuje w parametrze typ danych, którego się nie spodziewa. Wykonując rzutowanie przez wstawienie nawiasu z typem docelowym, w pewnym sensie potwierdzamy, że wiemy, co robimy i zadbaliśmy o prawidłowe wywołanie funkcji. Tutaj zadbanie polega na pobraniu rozmiaru struktury operatorem sizeof, dzięki czemu pobierzemy tyle bajtów, ile zajmuje struktura. Jeśli odczytalibyśmy więcej, nadmiarowe bajty zostałyby zapisane poza obszarem struktury, co miałoby nieprzewidziane skutki dla działania programu.

Odczyt rejestrów daty i czasu można sobie opakować w funkcję, która będzie pobierać tylko adres struktury:

```
void readDateTimeRegisters(RtcDateTimeRegisters * rtcDateTimeRegisters) {
 rtcReadRegisters(MCP79410_RTCSEC, (uint8_t *) &rtcDateTimeRegisters, sizeof(RtcDateTimeRegisters));
}
```

Należy zwrócić tutaj uwagę, że operatorem sizeof() pobierany jest rozmiar typu, a nie zmiennej. W tej funkcji mamy bowiem tylko zmienną wskaźnikową, czyli adres miejsca w pamięci, i ma on oczywiście inny rozmiar niż zmienna, na którą wskazuje. Jako że zmienna docelowa nie jest w tej funkcji widoczna, pobierany jest rozmiar typu. Wynosi on tyle samo co rozmiar zmiennej tego typu.

Gdy mamy odczytane rejestry, możemy wyświetlić je na LCD. Jednak jak pamiętamy, zapisane w nich liczby są w formacie BCD, a więc podzielone są na dziesiątki i jedności. Można co prawda obsługiwać je tak jak w przykładzie z ostatniej lekcji, wygodniej jednak będzie najpierw skonwertować je do normalnej postaci. Szczególnie że dobrze byłoby mieć funkcję zwracającą te wartości do dalszego przetwarzania, a nie tylko wyświetlającą je na wyświetlaczu.

Sprawy nie ułatwia fakt, że rejestry daty i godziny zawierają różne inne bity. Przed konwersją z formatu BCD trzeba je więc najpierw wyczyścić za pomocą masek bitowych. Pomocna tutaj będzie tabela 4 z poprzedniej lekcji. Zastosujemy operację logiczną AND i maski zerujące wszystko, co nie dotyczy czasu. Przykładowo wyrażenie pobierające sekundy będzie miało postać:

```
rtcDateTimeRegisters.RTCSEC & 0b01111111
```

Zwróconą przez nie wartość będzie można poddać konwersji z formatu BCD. Konwersję ułatwia nam to, że wszystkie rejestry przechowujące cyfrę dziesiątek mają ją zapisaną na czterech najstarszych bitach. Można więc napisać jedną funkcję konwertującą z BCD:

```
uint8_t fromBCD(uint8_t reg) {
 return ((reg & 0b11110000) >> 4) * 10 + (reg & 0b00001111); }
```

Dziesiątki zapisane są na 4 starszych bitach. Pobieramy je więc, zerując bity jednostek i przesuwając bity dziesiątek na swoje „normalne" miejsce. W tym momencie mamy liczbę dziesiątek, jaka jest w docelowej wartości. Jako że są to dziesiątki, musimy pomnożyć je przez 10, aby móc uzyskać wartość docelową. Pozostają jeszcze jedności. Pobieramy je z bajtu, zerując bity dziesiątek. Otrzymaną liczbę dodajemy do pomnożonych dziesiątek. W ten sposób z zapisu BCD odzyskiwania jest oryginalna liczba.

Możemy już napisać kod odczytujący aktualny czas. Jednak aby kod ten był użyteczny, powinien mieć postać funkcji zwracającej w jakiś sposób odczytane dane. Ponieważ potrzebujemy zwrócić kilka liczb, nie możemy zwrócić ich przez return. Z tym problemem zetknęliśmy się już w lekcji 7, kiedy również zwracaliśmy czas, z tym że nie pochodził on z RTC, ale był wczytywany z klawiatury. Wówczas funkcja pobierała trzy wskaźniki: na godziny, minuty i sekundy. Dzięki wskaźnikom mogliśmy zmodyfikować przekazywane zmienne i w ten sposób zwrócić kilka wartości jednocześnie. Tutaj jednak mamy jeszcze datę i dzień tygodnia, co daje 7 wartości. Oczywiście możemy stworzyć funkcję z siedmioma parametrami, ale nie będzie to zbyt eleganckie. Jako że są to wszystko dane powiązane ze sobą, dotyczące bieżącego czasu, nasuwa się użycie struktury. Jednak przed chwilą zdefiniowaliśmy taką strukturę, czy nie moglibyśmy z niej skorzystać? Ogólnie mówiąc, tak. Moglibyśmy też zamiast struktur posługiwać się tablicami typu uint8_t. Jednakże w programowaniu chodzi nie tylko o to, żeby kod działał, ale też żeby był czytelny, uporządkowany oraz łatwy w utrzymaniu i rozwoju. Struktura, którą zdefiniowaliśmy, opisuje fizyczne rejestry kostki MCP79410. Tymczasem w głównym fragmencie kodu chcielibyśmy skupić się bardziej na samych danych niż na organizacji rejestrów w MCP79410. Zdefiniujmy więc nową strukturę dla danych o czasie:

```
typedef struct {
 uint8_t seconds;
 uint8_t minutes;
 uint8_t hours;
 uint8_t dayOfWeek;
 uint8_t dayOfMonth;
 uint8_t month;
 uint16_t year;
} DateTime;
```

Nasz nowy typ strukturalny przypomina poprzedni, ale służy do przechowywania czasu w normalnej postaci, a nie do komunikacji z układem RTC. Nazwy pól są więc bardziej naturalne i przeznaczone do przechowywania liczb w zwykłej formie zamiast BCD. Pole roku zostało rozszerzone do 16 bitów, aby mogło przechowywać rok czterocyfrowy. Mając definicję nowej struktury, możemy zakończyć pisanie funkcji pobierającej aktualny czas. Będzie ona wyglądała jak na **listingu 3**. Dzięki niej możemy napisać kod wyświetlający aktualny czas (**listing 4**).

Jest on dosyć prosty. Deklarujemy na początku zmienną strukturalną dla czasu i przekazujemy ją do naszej funkcji pobierającej czas z RTC. Następnie wyświetlamy poszczególne pola na LCD: czas na pierwszej linii, datę z dniem tygodnia na drugiej. Jak wyświetlany jest dzień tygodnia? Korzystamy z tablicy wskaźników na ciągi:

```
const char * WeekDays[] = {" ", "pn", "wt", "sr", "cz", "pt", "so", "nd"};
```

W ten sposób w ostatniej linijce poprzedniego kodu możemy zaindeksować tablicę WeekDays i pobrać z niej wskaźnik na ciąg odpowiadający bieżącemu dniowi tygodnia, a następnie przekazać go do funkcji printf(). Tablica zawiera stałe, więc deklarujemy ją jako const. Oczywiście dwuliterowe skróty są tylko przykładem, można użyć np. pełnych słów. Jeśli chcemy użyć polskich liter, musimy zdefiniować je w pamięci wyświetlacza, zgodnie z opisem w lekcji 6. Pierwszy element tablicy zawierający spacje pełni funkcję wypełniacza. Nie będziemy go używać, bo nasz RTC jako dni tygodnia przyjmuje liczby 1–7, ale musi on istnieć, bo tablica musi zawierać element o indeksie zero.

Listing 3

```
void rtcReadDateTime(DateTime * dateTime) {
 RtcDateTimeRegisters rtcDateTimeRegisters;
 rtcReadRegisters(MCP79410_RTCSEC, (uint8_t *) &rtcDateTimeRegisters, sizeof(rtcDateTimeRegisters));
 dateTime->seconds = fromBCD(rtcDateTimeRegisters.RTCSEC & 0b01111111);
 dateTime->minutes = fromBCD(rtcDateTimeRegisters.RTCMIN & 0b01111111);
 dateTime->hours = fromBCD(rtcDateTimeRegisters.RTCHOUR & 0b01111111);
 dateTime->dayOfWeek = fromBCD(rtcDateTimeRegisters.RTCWKDAY & 0b00000111);
 dateTime->dayOfMonth = fromBCD(rtcDateTimeRegisters.RTCDATE & 0b00111111);
 dateTime->month = fromBCD(rtcDateTimeRegisters.RTCMTH & 0b00011111);
 dateTime->year = 2000 + fromBCD(rtcDateTimeRegisters.RTCYEAR & 0b11111111);    }
```

Listing 4

```
DateTime dateTime;
rtcReadDateTime(&dateTime);
lcdGotoXY(0, 0);
printf("%02d:%02d:%02d", dateTime.hours, dateTime.minutes, dateTime.seconds);
lcdGotoXY(0, 1);
printf("%02d-%02d-%4d", dateTime.dayOfMonth, dateTime.month, dateTime.year);
printf(" %s", WeekDays[dateTime.dayOfWeek]);
```

Nasz zegar musi mieć możliwość ustawiania czasu i daty. Wczytywanie tych danych z klawiatury przerabialiśmy w lekcji 7. Wczytywaliśmy wtedy tylko godzinę, minutę i sekundę, teraz natomiast potrzebujemy też daty. W tym celu możemy rozszerzyć tamtą funkcję o datę. Możemy też napisać prawie identyczną funkcję do wczytywania daty. Nie są to jednak najlepsze rozwiązania, są mało elastyczne i mało uniwersalne. Dobrze byłoby napisać funkcję służącą do wczytywania danych rozdzielonych separatorami (np. dwukropkami dla czasu i myślnikami dla daty). Byłaby ona przydatna nie tylko do wczytywania daty i godziny, ale też np. adresów IP czy MAC. Przykład takiej funkcji przedstawiono na **listingu 5**. Jest to rozbudowana wersja funkcji readTime() z lekcji 7. Funkcja pobiera trzy parametry: listę rozmiarów pól, liczbę elementów listy oraz tablicę zwracanych wartości. Jeśli będziemy wczytywać datę lub godzinę, w obu przypadkach użytkownik będzie wprowadzał trzy wartości. Będą one dwucyfrowe z wyjątkiem roku, który będzie czterocyfrowy. Podobnie jak w funkcji readTime() wciśnięte cyfry zapisujemy do tablicy o nazwie input. Tym razem jednak zapamiętywane są kody ASCII cyfr, a nie same liczby. Otrzymujemy więc na koniec w tablicy input ciągi znaków oddzielone spacjami, przedstawiające liczby, np. „12 34 56". Łatwo je wtedy skonwertować do liczb funkcją atoi(). Ponieważ przed wywołaniem funkcji mamy już wyświetlone separatory, np. dwukropki oddzielające godziny od minut i minuty od sekund, nasza funkcja musi je omijać. Tak jak w funkcji readTime() wykonujemy to, przesuwając kursor, ale dodatkowo też przeskakując bajty w tablicy input. Żeby to zrobić, musimy znać pozycje separatorów na wyświetlaczu a tym samym też te spacje w tablicy input, których nie chcemy nadpisać, bo mają oddzielać liczby. W tym celu obliczane są pozycje separatorów i zapamiętywane w tablicy separatorsPositions.

Listing 5

```
void readSeparatedNumbers(const uint8_t fieldsSizes[],
                          uint8_t fieldsNumber, uint16_t numbers[]) {
 if (!fieldsNumber) return;
 //oblicz długość ciągu wejściowego
 uint8_t inputStringLength = 0;
 for (uint8_t i = 0; i < fieldsNumber; i++) {
        inputStringLength += fieldsSizes[i];
        inputStringLength += 1;     }
 //oblicz pozycje separatorów
 uint8_t lastSeparatorPosition = 0;
 uint8_t separatorsPositions[fieldsNumber - 1];
 for (uint8_t i = 0; i < sizeof(separatorsPositions); i++) {
  separatorsPositions[i] = lastSeparatorPosition + fieldsSizes[i];
  lastSeparatorPosition = separatorsPositions[i] + 1;        }
 //wypełnij ciąg spacjami
 char input[inputStringLength];
 for (uint8_t i = 0; i < sizeof(input); i++) input[i] = ' ';
 uint8_t inputIndex = 0;
 uint8_t key = 0;
 while(1) {
  key = getKey();
  if (key == 15) {
   if (inputIndex > 0) {
    if (inputIndex == sizeof(input) - 2) {
     printf("%c", ' ');
    lcdWriteCommand(LCD_COMMAND_SHIFT | LCD_PARAM_SHIFT_LEFT); }
    inputIndex--;
    for (uint8_t i = 0; i < sizeof(separatorsPositions); i++) {
     if (inputIndex == separatorsPositions[i]) {
       lcdWriteCommand(LCD_COMMAND_SHIFT | LCD_PARAM_SHIFT_LEFT);
       inputIndex--;                }          }
    lcdWriteCommand(LCD_COMMAND_SHIFT | LCD_PARAM_SHIFT_LEFT);
    printf("%c", ' ');
    lcdWriteCommand(LCD_COMMAND_SHIFT | LCD_PARAM_SHIFT_LEFT); }
   continue;          }
  if (key < 10) input[inputIndex] = '0' + key;
  if (key == 10) input[inputIndex] = '0';
  if (key == 16) {
   break;          }
  printf("%c", input[inputIndex]);
  if (inputIndex < sizeof(input) - 2) {
   inputIndex++;          } else {
  lcdWriteCommand(LCD_COMMAND_SHIFT | LCD_PARAM_SHIFT_LEFT);    }
  for (uint8_t i = 0; i < sizeof(separatorsPositions); i++) {
   if (inputIndex == separatorsPositions[i]) {
    lcdWriteCommand(LCD_COMMAND_SHIFT | LCD_PARAM_SHIFT_RIGHT);
    inputIndex++;          }      }     }
 //wpisane ciągi cyfr skonwertuj do liczb
 uint8_t offset = 0;
 for (uint8_t i = 0; i < fieldsNumber; i++) {
  numbers[i] = atoi(input + offset);
  offset += fieldsSizes[i] + 1;     }      }
```

Aby ustawić czas w RTC, musimy zapisać rejestry przechowujące czas za pomocą funkcji odwrotnej do rtcReadDateTime(). Można ją obejrzeć na **listingu 6**.

Najpierw zerujemy rejestr RTCSEC, aby wyzerować bit ST i zatrzymać RTC. Potem przygotowywane są wartości rejestrów – z pól struktury daty i czasu pobierane są wartości i konwertowane do BCD. Przy okazji ustawiany jest bit VBATEN, aby włączyć podtrzymanie bateryjne. Po zapisaniu rejestrów zapisujemy jeszcze raz rejestr RTCSEC, tym razem z ustawionym bitem ST, aby wznowić odmierzanie czasu. Konwersja do BCD jest bardzo prosta. Dana liczba jest dzielona przez 10, aby uzyskać dziesiątki. Są one umieszczane na 4 starszych bitach. Następnie operatorem modulo wyciągane ją jedności i umieszczane na 4 młodszych bitach. Funkcja wygląda następująco:

```
uint8_t toBCD(uint8_t val) {
 return ((val / 10) << 4) | (val % 10);
}
```

Omówiliśmy wczytywanie danych z klawiatury i ich zapis do RTC. Jak jednak połączone są funkcje readSeparatedNumbers() i rtcWriteDateTime()? Przedstawia to funkcja setDateTime(), która wszystko spina razem (**listing 7**).

Najpierw komendą on/off włączamy kursor i wyświetlacz. Co prawda wyświetlacz był już włączony – nieustawienie bitu włączenia wyświetlacza spowodowałoby jego wyłączenie. Zaczynamy przygotowanie do wczytania czasu. W pierwszej linii wyświetlacza wyświetlone zostają dwukropki, pomiędzy którymi użytkownik będzie wpisywał godzinę, minutę i sekundę. Deklarowana jest tablica mówiąca, że wczytane zostaną 3 liczby dwucyfrowe. Wczytujemy je funkcją readSeparatedNumbers() i trafiają one do tablicy time. Analogicznie robimy z datą, ale wyświetlana jest ona w drugiej linijce. Na koniec dane z tablic time i date trafiają do struktury, którą potem przekazujemy do funkcji rtcWriteDateTime() zapisującej nowy czas i datę do rejestrów RTC.

A skąd bierze się dzień tygodnia? Nie ma potrzeby, aby użytkownik wpisywał go ręcznie, może zostać obliczony. Jest wiele sposobów obliczania dnia tygodnia, został wykorzystany jeden z wielu gotowych kodów dostępnych w Internecie. Kto ma ochotę, może przeanalizować jego działanie (**listing 8**).

Po uruchomieniu program wyświetla aktualny czas. Jeśli RTC byłby zatrzymany lub wyzerowany, trzeba wcisnąć dowolny klawisz i ustawić czas. Jeśli mamy ustawiony czas i włożoną baterię do oprawki na płytce testowej, to możemy odłączyć zasilanie płytki i po ponownym włączeniu zegar powinien nadal działać i pokazywać poprawny czas (**rysunek 1**).

Wyjście MFP

Kostka MCP79410 ma nóżkę MFP, która w zależności od konfiguracji może pełnić różne funkcje:

- Wyjście ogólnego przeznaczenia
- Sygnalizacja alarmu
- Źródło sygnału zegarowego

MFP jest wyjściem z otwartym drenem i wymaga podciągnięcia do plusa za pomocą rezystora 10kΩ. Jeśli nóżka działa jako ogólne wyjście, możemy sterować jej stanem logicznym, ustawiając

Rys. 1

Listing 7

```
void setDateTime() {
 //włącz kursor
 lcdWriteCommand
(LCD_COMMAND_ON_OFF |
   LCD_PARAM_ON_OFF_CURSOR | LCD_PARAM_ON_OFF_DISPLAY);
 //wczytaj czas
 lcdGotoXY(0, 0);
 printf("  :  :  ");
 lcdGotoXY(0, 0);
 const uint8_t timeFieldsSizes[] = {2, 2, 2};
 uint16_t time[3];
 readSeparatedNumbers(timeFieldsSizes, 3, time);
 //wczytaj datę
 lcdGotoXY(0, 1);
 printf("  -  -    ");
 lcdGotoXY(0, 1);
 const uint8_t dateFieldsSizes[] = {2, 2, 4};
 uint16_t date[3];
 readSeparatedNumbers(dateFieldsSizes, 3, date);
 DateTime newDateTime;
 newDateTime.hours = time[0];
 newDateTime.minutes = time[1];
 newDateTime.seconds = time[2];
 newDateTime.dayOfMonth = date[0];
 newDateTime.month = date[1];
 newDateTime.year = date[2];
 newDateTime.dayOfWeek = getWeekDay(&newDateTime);
 //zapisz nowy czas i datę
 rtcWriteDateTime(&newDateTime);
 //wyłącz kursor
 lcdWriteCommand(LCD_COMMAND_ON_OFF | LCD_PARAM_ON_OFF_DISPLAY); }
```

Listing 8

```
uint8_t getWeekDay(DateTime * dateTime) {
 int d = dateTime->dayOfMonth;
 int m = dateTime->month;
 int y = dateTime->year;
 uint8_t weekday  = (d += m < 3 ? y-- : y - 2, 23*m/9 + d + 4 + y/4- y/100 + y/400)%7;
 if (!weekday) weekday = 7;
 return weekday;
}
```

odpowiednią wartość bitu OUT w rejestrze CONTROL (rejestr numer 07h). Stan na nóżce MFP może być też ustawiany automatycznie i sygnalizować wystąpienie alarmu (włączenie się budzika). Wreszcie na MFP można wyprowadzić podzielony sygnał zegarowy (32768Hz). Stopień podziału wyznaczają bity SQWFS1..0 z rejestru CONTROL zgodnie z **tabelą 1**.

Tabela 1

SQWFS1	SQWFS0	Podział	Częstotliwość
0	0	1:1	32768 Hz
0	1	1:4	8192 Hz
1	0	1:8	4096 Hz
1	1	1:32768	1 Hz

Sygnał zegarowy można wykorzystać do taktowania innych układów scalonych, take mikrokontrolera. Jak pamiętamy z lekcji 4, Timer1 może być taktowany zewnętrznym sygnałem podawanym na nóżki T0 i T1 (PB0 i PB1). Mając RTC, możemy więc nie tylko zwolnić mikrokontroler z odmierzania czasu, ale też w niektórych sytuacjach zapewnić stabilny sygnał zegarowy dla innych operacji związanych z czasem, co pozwoli np. zrezygnować z rezonatora kwarcowego dla mikrokontrolera.

Konfigurację nóżki MFP wyznaczają bity SQWEN, ALM1EN i ALM0EN z rejestru CONTROL. Ogólnie, jeśli nie jest włączone generowanie sygnału zegarowego ani żaden z dwóch alarmów, to nóżka działa jako wyjście dowolnego przeznaczenia. Jeśli włączony jest co najmniej jeden alarm, to wówczas nóżka kontrolowana jest przez funkcję alarmu. Natomiast przy włączonym wyprowadzeniu sygnału zegarowego na MFP jest właśnie sygnał zegarowy, niezależnie od konfiguracji alarmów. Opcje konfiguracji zebrane są w **tabeli 2**. Z kolei **tabela 3** przedstawia bity rejestru CONTROL.

Tabela 2

SQWEN	ALM0EN	ALM1EN	Tryb pracy
0	0	0	Wyjście ogólne
0	0	1	Wyjście alarmowe
0	1	0	
0	1	1	
1	x	x	Sygnał zegarowy

Tabela 3

OUT	SQWEN	ALM1EN	ALM0EN	EXTOSC	CRSTRIM	SQWFS1	SQWFS0

Przy podtrzymaniu bateryjnym nóżka MFP działa tylko jako wyjście alarmowe. Działanie jako wyjście ogólnego przeznaczenia lub jako źródło sygnału zegarowego wymaga normalnego zasilania.

Alarmy

Jak wspomniano wyżej, nasz układ RTC potrafi nie tylko odmierzać czas, ale może też generować alarmy w określonych momentach. Jak to działa? Alarmy są dwa i konfigurujemy je za pomocą rejestrów bardzo podobnych do rejestrów bieżącego czasu. Ustawiamy więc sekundy, minuty, godziny, dzień tygodnia, dzień miesiąca i miesiąc. Nie ma rejestru roku. **Tabela 4** przedstawia rejestry alarmu pierwszego. Drugi alarm ma analogiczne rejestry pod adresami 11–16h. Różnią się one tylko oznaczeniami, zamiast cyfry 0 mają w nazwach cyfrę 1. To samo dotyczy bitów: rejestr ALM1WKDAY ma bity ALM1MSK2, ALM1MSK1, ALM1MSK0 i ALM1IF w miejscu bitów ALM0MSK2, ALM0MSK1, ALM0MSK0 i ALM0IF.

Ogólnie zasada działania jest taka, że alarm następuje gdy bieżący czas stanie się zgodny z czasem ustawionym w rejestrach danego alarmu. Aby jednak alarm nie wypadał tylko raz w roku, możliwe jest maskowanie niektórych składowych czasu. Definiują to bity ALM0MSK2, ALM0MSK1 i ALM0MSK0 dla pierwszego alarmu i analogiczne bity dla drugiego alarmu. Możliwe kombinacje przedstawia **tabela 5.**

Tabela 5

ALMxMSK2..0	Dopasowywany element
000	Sekundy
001	Minuty
010	Godziny
011	Dzień tygodnia
100	Dzień miesiąca
101	-
110	-
111	Wszystkie, łącznie z miesiącem

Jak widać, możliwości są tutaj ograniczone. Podczas gdy niektóre układy RTC mają oddzielne bity maskujące dla każdej składowej czasu, co pozwala tworzyć różne kombinacje, w MCP79410 można wybrać tylko pojedyncze składowe (i to z wyjątkiem miesiąca) albo wszystkie. Założeniem producenta było łatwe tworzenie alarmów okresowych. Przykładowo wartość 000 da alarm co minutę, a 010 co dobę. Jeśli natomiast chcielibyśmy mieć alarm codziennie o 07.15, mamy dwa wyjścia. Jedno to wpisanie liczby 15 do rejestru ALMxMIN i wybranie maski 001. Alarm będzie występował co godzinę, 15 minut po pełnej godzinie. Za każdym wystąpieniem alarmu trzeba będzie sprawdzać, czy bieżąca godzina to 7. Jeśli tak, wówczas podjąć określoną akcję. Drugie rozwiązanie to maska 111 i ustawienie czasu alarmu na 07:15:00 oraz datę najbliższego alarmu (bieżący dzień, jeśli jest przed 7.15, kolejny, jeśli już minęła). Wówczas przy każdy wyzwoleniu alarmu trzeba przestawiać jego datę na kolejny dzień. Ewentualnie można też ustawić maskę 000, aby alarm wyzwalał się co minutę i całą obsługę zrobić już po stronie mikrokontrolera. Wtedy alarm z RTC będzie de facto źródłem sygnału 1/60 Hz, a całą logikę alarmów będziemy musieli zawrzeć w naszym

Listing 6

```
void rtcWriteDateTime(DateTime * dateTime) {
 rtcWriteRegister(MCP79410_RTCSEC, 0);
 RtcDateTimeRegisters rtcDateTimeRegisters;
 rtcDateTimeRegisters.RTCSEC = toBCD(dateTime->seconds);
 rtcDateTimeRegisters.RTCMIN = toBCD(dateTime->minutes);
 rtcDateTimeRegisters.RTCHOUR = toBCD(dateTime->hours);
 rtcDateTimeRegisters.RTCWKDAY = toBCD(dateTime->dayOfWeek) | _BV(MCP79410_VBATEN);
 rtcDateTimeRegisters.RTCDATE = toBCD(dateTime->dayOfMonth);
 rtcDateTimeRegisters.RTCMTH = toBCD(dateTime->month);
 rtcDateTimeRegisters.RTCYEAR = toBCD(dateTime->year - 2000);
 rtcWriteRegisters(MCP79410_RTCSEC, (uint8_t *) &rtcDateTimeRegisters, sizeof(rtcDateTimeRegisters));
 rtcWriteRegister(MCP79410_RTCSEC, rtcDateTimeRegisters.RTCSEC | _BV(MCP79410_ST));  }
```

Tabela 4

Adres	Rejestr	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0Ah	ALM0SEC	-	SECTEN2	SECTEN1	SECTEN0	SECONE3	SECONE2	SECONE1	SECONE0
0Bh	ALM0MIN	-	MINTEN2	MINTEN1	MINTEN0	MINONE3	MINONE2	MINONE1	MINONE0
0Ch	ALM0HOUR	-	12/24	AM/PM HRTEN1	HRTEN0	HRONE3	HRONE2	HRONE1	HRONE0
0Dh	ALM0WKDAY	ALMPOL	ALM0MSK2	ALM0MSK1	ALM0MSK0	ALM0IF	WKDAY2	WKDAY1	WKDAY0
0Eh	ALM0DATE	-	-	DATETEN1	DATETEN0	DATEONE3	DATEONE2	DATEONE1	DATEONE0
0Fh	ALM0MTH	-	-	-	MTHTEN0	MTHONE3	MTHONE2	MTHONE1	MTHONE0

Wyrównanie pól struktury

W tej lekcji oraz poprzedniej korzystaliśmy ze struktur, które odwzorowywały rejestry w podłączanych układach. Wskaźniki na te struktury były potem traktowane jako wskaźniki na tablice bajtów. Dzięki temu można było wygodnie przesyłać zmienne strukturalne z/do mikrokontrolera. Na naszym 8-bitowym mikrokontrolerze nie sprawia to problemów, ale jeśli przyjdzie nam pisać kod na komputer PC lub mikrokontroler ARM, sprawa będzie nieco bardziej złożona. Na tych architekturach kompilator rozkłada pola struktury tak, aby znalazły się w pamięci pod adresami podzielnymi przez ich rozmiar.

Rozważmy poniższą strukturę:

```
struct s1 { uint32_t x;  uint8_t y;  }
```

Pole x znajdzie się pod adresem A, natomiast pole y pod adresem A + 4, zaraz za 4-bajtowym polem x. Rozłóżmy jednak pola odwrotnie:

```
struct s2 { uint8_t y;  uint32_t x;  }
```

Pole y zostało przesunięte na początek, więc to ono znajdzie się pod początkowym adresem A. Do tego miejsca nie ma niespodzianek. Co się jednak stanie z polem x? Trafi ono pod adres A + 4, mimo że poprzedzające je pole y zajmuje tylko 1 bajt. Kompilator wyrównując (ang. alignment) położenie 4-bajtowego pola x, wprowadzi 3-bajtową dziurę (ang. padding) między polami x i y. W ten sposób struktura s1 zajmie w pamięci 5 bajtów, a struktura s2 osiem, mimo że zawierają takie same pola.

Programując mikrokontrolery AVR, nie musimy się tym przejmować, ale o wyrównaniu pól trzeba pamiętać, aby uniknąć niemiłych niespodzianek przy programowaniu innych mikrokontrolerów lub pod Windows/Linuksem. Jeśli wówczas będzie nam potrzebne wyłączenie wyrównania, trzeba skorzystać z tzw. pakowania struktur. W przypadku użycia kompilatora GCC używa się atrybutu packed:

```
struct __attribute__((__packed__)) s2 {
uint8_t y;      uint32_t x;  }
```

Ta struktura zajmie 5 bajtów.

kodzie. Wybór danego rozwiązania będzie zależał już od konkretnych okoliczności i funkcji, jakie będzie miał mieć nasz program. Uwaga! RTC sprawdza warunek wystąpienia alarmu co sekundę i uruchamia go za każdym razem gdy warunek jest spełniony. W związku z tym alarm ustawiony np. na daną minutę wyzwoli się 60 razy podczas trwania tej minuty. Trzeba to uwzględnić w programie korzystającym z alarmów RTC.

Alarmy włączamy za pomocą bitów ALM0EN i ALM1EN w rejestrze CONTROL. Jak pamiętamy, jeśli nie jest włączona funkcja sygnału zegarowego, wówczas włączenie przynajmniej jednego alarmu powoduje przejęcie kontroli nad nóżką MFP. Za pomocą bitów ALMPOL w rejestrze ALMxWKDAY ustawiamy, jaki stan logiczny ma pojawić się na tym wyprowadzeniu w momencie alarmu. 0 oznacza stan niski, 1 stan wysoki. Stan alarmu możemy w każdej chwili sprawdzić, odczytując bity ALM0IF i ALM1IF. Jedynka oznacza wystąpienie alarmu. Po zgłoszeniu alarmu na danym bicie mikrokontroler powinien go wyzerować, aby mógł być ustawiony przy kolejnym alarmie. Przy tym nie trzeba konkretnie ustawiać go na zero, każdy bowiem zapis do rejestru ALMxWKDAY spowoduje wyzerowanie flagi alarmu.

Jak zachowa się nóżka MFP, jeśli włączone są dwa alarmy? Alarmy są przecież niezależne i w danej chwili może np. uaktywnić się tylko jeden z nich. Czy nóżka powiadomi wtedy o alarmie? Tak, wystarczy tylko jeden aktywny alarm, aby nóżka była w stanie wskazującym na wystąpienie alarmu. Jeśli ALMPOL = 0, wówczas przy braku alarmu na MFP będzie 1, a wystąpienie przynajmniej jednego alarmu da 0. Natomiast przy ALMPOL = 1 w spoczynku na nóżce MFP będzie 0, a po wystąpieniu przynajmniej jednego alarmu pojawi się 1.

Adres	Rejestr	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
18h	PWRDNMIN	-	MINTEN2	MINTEN1	MINTEN0	MINONE3	MINONE2	MINONE1	MINONE0
19h	PWRDNHOUR	-	12/24	AM/PM HRTEN1	HRTEN0	HRONE3	HRONE2	HRONE1	HRONE0
1Ah	PWRDNDATE	-	-	DATETEN1	DATETEN0	DATEONE3	DATEONE2	DATEONE1	DATEONE0
1Bh	PWRDNMTH	WKDAY2	WKDAY1	WKDAY0	MTHTEN0	MTHONE3	MTHONE2	MTHONE1	MTHONE0

Tabela 6

W tym momencie uważny Czytelnik powie: chwila, przecież bity ALMPOL są dwa! Który będzie więc brany pod uwagę? Otóż tak naprawdę jest tylko jeden bit ALMPOL, w rejestrze pierwszego alarmu. W rejestrze drugiego alarmu jest jego kopia tylko do odczytu. Stan, jaki ma wystąpić na nóżce MFP w przypadku alarmu, konfiguruje się więc tylko w jednym miejscu. Kopiami są również bity 12/24, zawierają wartość analogicznego bitu z rejestru RTCHOUR.

Timestamp zaniku/powrotu zasilania

Ciekawą funkcją układu MCP79410 jest zapisywanie momentu (timestampu) zaniku zasilania oraz jego powrotu. Oczywiście, aby ta funkcja działała, konieczne jest podtrzymanie bateryjne. Gdy odłączone zostanie główne zasilanie, bieżący czas zostaje zapisany w rejestrach PWRDNMIN, PWRDNHOUR, PWRDNDATE i PWRDNMTH znajdujących się po adresami 18h–1Bh. Analogicznie ponowne podłączenie zostanie zapisane w rejestrach PWRUPMIN, PWRUPHOUR, PWRUPDATE i PWRUPMTH, które znajdziemy pod adresami 1Ch–1Fh. Rejestry dla odłączenia zasilania przedstawiono w **tabeli 6**. Rejestry dla powrotu zasilania mają identyczną strukturę. Fakt zaniku zasilania odnotowywany jest w bicie PWRFAIL rejestru RTCWKDAY (adres 03h). Gdy włączone zostanie główne zasilanie i swoją pracę rozpocznie mikrokontroler, powinien on sprawdzić stan tego bitu i wyzerować go, aby funkcja mogła zadziałać kolejny raz. Oczywiście jest to zbędne, gdy nie zamierzamy korzystać z funkcji timestampu zaniku zasilania.

BP1	BP0	STATUS	Obszar chroniony
0	0	00h	Brak ochrony
0	1	04h	Górna ćwiartka (60h–7Fh)
1	0	08h	Górna połowa (40h–7Fh)
1	1	0Ch	Całość (00h–7Fh)

Tabela 7

Pamięć SRAM

Jeśli zaadresujemy kostkę MCP79410 jako RTC, wówczas oprócz rejestrów związanych z odmierzaniem mamy też dostęp do 64 bajtów pamięci SRAM. Adresacja, odczyt i zapis przeprowadzane są tak samo jak rejestrów RTC, możemy więc korzystać z naszych funkcji takich jak rtcReadRegisters() czy rtcWriteRegisters(). Jedyna różnica jest taka, że komórki SRAM nie są związane z odmierzaniem czasu i można je wykorzystać zupełnie dowolnie. Obszar pamięci SRAM znajduje się pod adresami 20h–5Fh. Jako że jest to pamięć RAM, jest to pamięć ulotna, ale może ona korzystać z podtrzymania bateryjnego i zachowa ona swoją zawartość tak długo jak bateria będzie podłączona i ustawiony będzie bit VBATEN.

Pamięć EEPROM

W przypadku gdy MCP79410 zostanie zaadresowany jako EEPROM, mamy do dyspozycji 128 bajtów nieulotnej pamięci EEPROM pod adresami 00h–7Fh, 8 bajtów chronionej pamięci EEPROM pod adresami F0h–F7h oraz rejestr STATUS pod adresem FFh.

Główna pamięć EEPROM, mająca rozmiar 128 bajtów, podzielona jest na 8-bajtowe strony. Co to oznacza? W przypadku odczytu – nic. Możemy odczytywać EEPROM tak samo jak SRAM czy rejestry RTC, pobierając jeden lub więcej kolejnych bajtów. W przypadku zapisu sytuacja jest już nieco inna. W pamięci EEPROM oraz bardzo podobnej do niej pamięci Flash nie można zapisywać sobie dowolnych komórek w dowolnym momencie. Ze względu na sposób działania tych pamięci trzeba najpierw wykasować cały obszar zwany stroną i dopiero wtedy można coś w nim zapisać.

W przypadku naszej kostki sytuacja wygląda trochę inaczej, ponieważ w jej przypadku nie ma specjalnych komend wykonujących jakąś akcję, jest tylko zapis i odczyt rejestrów/komórek pamięci. Rozwiązano to w ten sposób, że wysyłane przez mikrokontroler bajty trafiają do bufora. Gdy mikrokontroler wyśle na koniec transmisji na szynie I²C sygnał STOP, układ MCP79410 kasuje stronę EEPROM-u właściwą dla aktualnego adresu i zapisuje do niej zawartość bufora. Jeśli będziemy zapisywać pojedynczy bajt, wszystko odbędzie się automatycznie i zapis będzie wyglądał tak samo, jak zapis komórki SRAM czy rejestru. W przypadku zapisu większej liczby bajtów ogranicza nas rozmiar strony: nie możemy za jednym razem zapisać więcej niż 8 bajtów. Ponadto dany zapis nie może obejmować więcej niż jednej strony. Oznacza to, że możemy zapisać 8 bajtów od adresu 00h do 07h, ale nie możemy zapisać ich od adresu 01h do 08h. Ostatni bajt wypada bowiem już na kolejnej stronie, a MCP79410 nie może skasować i zapisać więcej niż jednej strony w danym cyklu. Co się więc stanie z „wystającymi" bajtami? Czy zostaną utracone? Otóż gdy wysyłamy kolejne bajty do zapisu, nasz układ po każdym bajcie zwiększa wskaźnik (indeks) w swoim wewnętrznym buforze. W ten sposób kolejne bajty trafiają w kolejne komórki bufora. Gdy zostanie zapisana ostatnia, ósma komórka bufora, wskaźnik zostaje zresetowany i pokazuje na pierwszą komórkę strony. Zatem w przypadku zapisu ośmiu bajtów pod adresy 01h–08h ostatni bajt trafi pod adres 00h, a komórka o adresie 08h pozostanie nienaruszona. Jeśli więc potrzebujemy zapisać dane na obszarze obejmującym więcej niż jedną stronę, musimy zrobić to etapami.

Uwaga! Zapis do pamięci EEPROM zajmuje trochę czasu. Nasz układ musi bowiem skasować stronę pamięci i zapisać do niej zawartość bufora. Potrzebuje na to znacznie więcej czasu niż na zapis do pamięci SRAM, który następuje praktycznie natychmiastowo i nie wymaga większej uwagi. Gdy trwa zapis do EEPROM-u nie możemy wykonywać żadnych operacji na tej pamięci. Działa natomiast część RTC układu MCP79410 i trwający zapis EEPROM-u nie przeszkadza w dostępie do rejestrów RTC. Jeśli jednak chcemy coś zapisać lub odczytać z EEPROM-u, musimy poczekać na koniec zapisu. Mamy więc tutaj sytuację podobną jak w lekcji 13, gdy czekaliśmy na zakończenie pomiaru temperatury przez układ DS18B20. Tak jak wtedy mamy dwa wyjścia: dodać w naszym programie odpowiednio duże opóźnienie lub też odpytywać okresowo układ, czy zakończył swoje wewnętrzne operacje. Producent nie podaje, jak długo może maksymalnie trwać zapis, ale eksperymenty pokazują, że opóźnienie wynoszące 5 milisekund powinno wystarczyć. Natomiast aby sprawdzić, czy zapis jeszcze trwa, możemy wysłać sygnał START i adres do zapisu. Jeśli kostka odpowie bitem ACK, oznacza to, że zapis się już zakończył i jest gotowa do wymiany danych. W przypadku braku ACK należy spróbować później, np. po milisekundzie. Przykładowa funkcja sprawdzająca, czy układ jest zajęty, może wyglądać następująco:

```
uint8_t eepromIsBusy(void) {
 i2cStart();
 uint8_t status = i2cSendAddress(MCP79410_EEPROM_WRITE_ADDRESS);
 i2cStop();
 return status != I2C_OK;
}
```

Funkcja zwraca 1, jeśli funkcja wysyłająca adres zwróciła status różny niż I2C_OK. Oczekiwanie na koniec zapisu będzie wyglądało tak:

```
while (eepromIsBusy()) _delay_ms(1);
```

Wspomnieliśmy na początku, że w obszarze adresowym EEPROM-u znajduje się rejestr STATUS. Jego nazwa jest nieco myląca, bo służy on do zabezpieczania EEPROM-u przed zapisem. Domyślnie ochrony nie ma. Możemy jednak ją włączyć dla ¼, ½ lub całej pamięci, zgodnie z **tabelą 7**. Ponieważ rejestr STATUS ma aktywne tylko dwa bity: BP1 i BP0, znajdujące się na pozycjach 3 i 2, zarządzanie ochroną sprowadza się do wyboru jednej z 4 wartości: 00h, 04h, 08h lub 0Ch.

Oprócz tego, że możemy w pewnym stopniu chronić główną pamięć EEPROM, układ MCP79410 ma jedną stronę (8 bajtów) z jeszcze większym stopniem ochrony przed zapisem. Strona ta zajmuje obszar o adresach F0h–F7h. Żeby do niej zapisać, musimy wpisać specjalne wartości do rejestru EEUNLOCK, który jest rejestrem RTC o adresie 09h. Te wartości to 55h i AAh. A więc procedura zapisu polega na zaadresowaniu kostki MCP79410 jako RTC, zapisu bajtu 55h do rejestru 09h, następnie znów zaadresowania jako RTC i zapisu bajtu AAh do tego samego rejestru, następnie zaadresowaniu kostki jako EEPROM i wpisaniu żądanych wartości do komórek F0h–F7h. Wygląda to skomplikowanie, ale kod zapisujący chroniony obszar będzie bardzo prosty:

```
void eepromWriteProtected(uint8_t reg, uint8_t * data, uint8_t size) {
 rtcWriteRegister(MCP79410_EEUNLOCK, 0x55);
 rtcWriteRegister(MCP79410_EEUNLOCK, 0xAA);
 eepromWriteBytes(reg, data, size);
}
```

Zadania

W tym odcinku proponuję zastanowić się nad następującymi zadaniami:

1. Zegar z budzikiem, wykorzystać alarm RTC
2. Tester pamięci SRAM
3. Tester pamięci EEPROM

W materiałach dodatkowych znajduje się listing 1 oraz kompletny projekt zegara RTC z podtrzymaniem bateryjnym.

Grzegorz Niemirowski
grzegorz@grzegorz.net