Kurs AVR – lekcja 18

Rozwiązania zadań z ostatniego odcinka

Pierwsze zadanie domowe z poprzedniej lekcji polegało na napisaniu programu mierzącego temperaturę i cyklicznie zapisującego wynik pomiaru do pamięci EEPROM. Program miał mieć możliwość przeglądania zapisanych danych. Przykładową realizację przedstawia **listing 1** (fragmenty poniżej, całość w Elportalu wśród materiałów dodatkowych do tego zadania).

Program musi jakoś przechowywać pomiary w EEPROM-ie. W przykładzie zdefiniowano w tym celu strukturę TemperatureRecord, która przechowuje informacje o dacie i czasie pomiaru oraz samą wartość temperatury. Struktury od kolejnych pomiarów są umieszczane jedna po drugiej w EEPROM-ie. Ponieważ struktura/rekord zajmuje 12 bajtów, w naszym EEPROM-ie zmieści się 85 pomiarów. Jeśli pomiar będzie zapisywany co 15 sekund, pamięć starczy na nieco ponad 21 minut. Jeśli jest to za mało, mamy trzy wyjścia: zapisywać rzadziej, zapisywać do innej pamięci (np. Flash lub zewnętrznej pamięci) albo zmniejszyć rozmiar rekordu. Częstotliwość zapisu zależy od konkretnych potrzeb, np. temperatura w pomieszczeniu nie zmienia się zbyt szybko i można ją mierzyć co kilka minut. Wybór innej pamięci to kwestia wydzielenia obszaru Flasha albo dodania zewnętrznej pamięci, co zwiększa koszt i komplikuje układ. Zmiana rozmiaru rekordu to kwestia już stricte programistyczna.

Listing 1

```
(...)
int main(void) { i2cInit(); keybInit(); lcdInit(); lcdInitPrintf();
 //odczyt adresu ostatnio zapisanego rekordu w EEPROM
 uint16_t lastRecordAddress;
 eepromReadBytesInternal(0, (uint8_t*) &lastRecordAddress,
 sizeof(lastRecordAddress));
 if (lastRecordAddress > 1023 - sizeof(TemperatureRecord))
 lastRecordAddress = 2;
 uint16_t browsingRecordAddress = lastRecordAddress;
 uint8_t recordWritten = 0;
 TemperatureRecord temperatureRecord;
 enum TemperatureState temperatureState = TEMP_IDLE;
 uint8_t browsing = 0;
 while(1){
  //odczyt temperatury
  getTemperature(&temperatureRecord.temperature);
  //odczyt daty i czasu
  DateTime * dateTime = &temperatureRecord.dateTime;
  rtcReadDateTime(dateTime);
  //zapis rekordu
  if ((dateTime->seconds % 15) == 0) {
   if (!recordWritten) {
    lastRecordAddress += sizeof(temperatureRecord);
    if (lastRecordAddress > 1023 - sizeof(temperatureRecord))
    lastRecordAddress = 2;
    browsingRecordAddress = lastRecordAddress;
    eepromWriteBytesInternal(lastRecordAddress, (uint8_t *)
    &temperatureRecord, sizeof(temperatureRecord));
    eepromWriteBytesInternal(0, (uint8_t*) &lastRecordAddress,
    sizeof(lastRecordAddress));
    recordWritten = 1;  }  } else {    recordWritten = 0;   }
  //wyświetlenie danych
  if (browsing) {
   TemperatureRecord historicRecord;
   eepromReadBytesInternal(browsingRecordAddress, (uint8_t *) &historicRecord,
   sizeof(temperatureRecord));
   displayRecord(&historicRecord);
  } else {    displayRecord(&temperatureRecord);   }
  //obsługa klawiatury
  if (readKeyboard()) {
   uint8_t key = getKey();
   if (browsing) {
    if (key == 1) {
     browsingRecordAddress += sizeof(temperatureRecord);    }
    if (key == 5) {
     browsingRecordAddress -= sizeof(temperatureRecord);    }
    if (browsingRecordAddress > 1023 - sizeof(temperatureRecord))
    browsingRecordAddress = 2;
    if (key == 16) browsing = 0;
   } else {
    if (key == 1) setDateTime();
    if (key == 2) browsing = 1;    }  }  }  }
(...)
void getTemperature(float * temperature) {
 static enum TemperatureState temperatureState = TEMP_IDLE;
 switch (temperatureState) {
  case TEMP_IDLE:
   oneWireReset();
   oneWireSendByte(OW_SKIP_ROM);
   oneWireSendByte(DS18B20_CONVERT_T);
   temperatureState = TEMP_IN_PROGRESS;
   break;
  case TEMP_IN_PROGRESS:
   if (oneWireReceiveBit()) temperatureState = TEMP_READY;
   break;
  case TEMP_READY:
   oneWireReset();
   oneWireSendByte(OW_SKIP_ROM);
   oneWireSendByte(DS18B20_READ_SCRATCHPAD);
   int16_t temp;
   uint8_t * byte = (uint8_t *) &temp;
   byte[0] = oneWireReceiveByte();
   byte[1] = oneWireReceiveByte();
   *temperature = temp / 16.0;
   temperatureState = TEMP_IDLE;
   break;  } }  (...)
```

W naszym przykładzie zapisujemy wszystkie dane oferowane przez RTC, w tym sekundy, dzień tygodnia oraz rok. Te dane często są zbędne i wystarczy np. godzina, minuta, dzień i miesiąc, co da 4 bajty. Temperaturę zapisujemy za pomocą zmiennoprzecinkowego typu float, który ma rozmiar 4 bajtów. Jeśli nie potrzebujemy części ułamkowej i wystarczy nam zakres od –128 do +128 stopni, wówczas wystarczy jeden bajt (typ int8_t). Optymalizując w ten sposób rozmiar rekordu, można go zmniejszyć z 12 do 5 bajtów. Warto przy tym zauważyć, że nadal dane o czasie i dacie zajmują dużo więcej miejsca niż sama temperatura. Plusem zapisywania daty i czasu w każdym rekordzie jest to, że urządzenie może działać z przerwami. Jeśli układ będzie działał w sposób ciągły, można zapisać tylko datę i czas pierwszego pomiaru. Wówczas przy przechowywaniu temperatury w jednym bajcie, w EEPROM-ie zmieści się ponad tysiąc pomiarów. Przy zapisie co minutę pojemność wyniesie ponad 16 godzin.

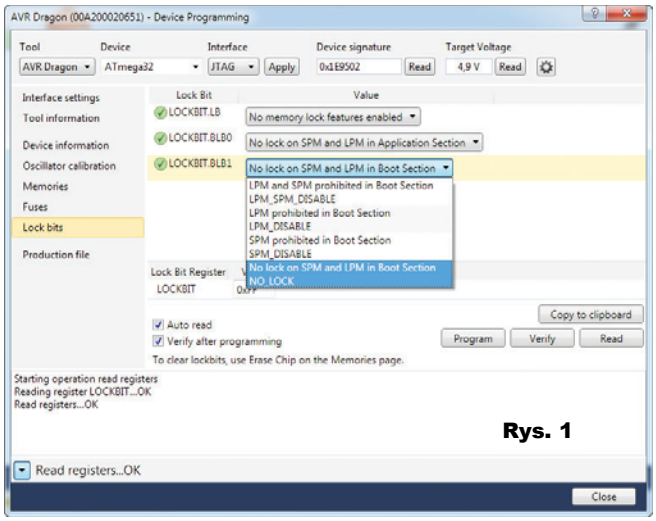
Ponieważ w przykładzie program miał być niewrażliwy na wyłączenie zasilania, w pierwszych dwóch bajtach EEPROM-u zapisywany jest adres ostatnio zapisanego rekordu. W ten sposób program nie rozpoczyna za każdym razem od początku pamięci, ale kontynuuje zapis od ostatnio zapisanego miejsca. Dlatego na początku programu, po inicjalizacji LCD, klawiatury i I²C, wykonywany jest odczyt pierwszych dwóch bajtów EEPROM-u. Jeśli nie wskazują na prawidłowy adres, przyjmowany jest adres domyślny wynoszący 2, a więc wskazujący na miejsce za owymi dwoma bajtami.

Główna pętla wykonuje pomiar temperatury oraz sprawdza stan klawiatury. Jest to ważne, ponieważ pomiar zajmuje kilkaset milisekund i nie możemy skorzystać z funkcji getTemperature() znanej z lekcji 14. Inaczej korzystanie z klawiatury będzie bardzo utrudnione, użytkownik musiałby przytrzymywać klawisze. Bezczynne czekanie na koniec pomiaru nie jest jednak konieczne. Mikrokontroler może zainicjować pomiar w układzie DS18B20 a następnie sprawdzać, czy się już zakończył. W międzyczasie może zajmować się innymi rzeczami, np. odczytem stanu klawiatury. Gdy pomiar się zakończy, wówczas mikrokontroler może odczytać zmierzoną wartość. Odczyt temperatury trzeba więc podzielić na trzy fazy: start pomiaru, sprawdzanie, czy pomiar się zakończył, odczyt wyniku. Na listingu widoczna jest zmodyfikowana funkcja getTemperature(), sprawdzająca, w jakiej

fazie znajduje się pomiar. W tym celu korzysta z typu wyliczeniowego TemperatureState. Przy pierwszym uruchomieniu funkcja inicjuje zmienną temperatureState wartością TEMP_IDLE. Jest to zmienna statyczna, więc przy kolejnych wywołaniach funkcji będzie pamiętać swoją wartość z poprzedniego wywołania. Przy stanie TEMP_IDLE następuje rozpoczęcie pomiaru temperatury i stan zmienia się na TEMP_IN_PROGRESS. Przy kolejnych wywołaniach funkcji jest ona w tym właśnie stanie i nie wykonuje już rozpoczęcia pomiaru, ale sprawdzenie, czy już się zakończył. Jeśli tak, stan zmienia się na TEMP_READY. Wówczas następne wywołanie spowoduje odczyt temperatury z dwóch pierwszych bajtów scratchpada kostki DS18B20 i modyfikację zmiennej wskazywanej przez przekazany do funkcji wskaźnik. W pozostałych stanach modyfikacja nie następuje. Następnie stan zostaje zmieniony na początkowy TEMP_IDLE.

W ten sposób funkcja getTemperature() przechodzi pomiędzy poszczególnymi stanami. Możliwe są też nieco inne podejścia. Np. zmienna przechowująca stan może być zadeklarowana w funkcji main(), ale w naszym przykładzie nie ma konieczności, aby główna funkcja znała stan naszej funkcji. Ważne, że pole temperature w strukturze temperatureRecord jest odpowiednio aktualizowane danymi z termometru DS18B20. Można też pójść dalej i funkcję getTemperature() rozbić na trzy funkcje, dla poszczególnych faz, i wywoływać je z głównej pętli, która będzie w instrukcji switch sprawdzać stan i wywoływać odpowiednią funkcję. Nie jest to jednak konieczne w tym przykładzie.

Po wywołaniu funkcji odpowiedzialnej za temperaturę główna pętla pobiera aktualny czas i sprawdza, czy nadszedł moment zapisu temperatury. W przykładzie założono, że zapis ma się odbywać co 15 sekund, sprawdzane jest więc operatorem modulo, czy reszta z dzielenia liczby sekund przez 15 daje zero. Jeśli tak, następuje obliczenie kolejnego adresu w EEPROM-ie, pod którym zostanie zapisany nowy rekord. Jeśli adres wypadnie poza obszarem pamięci, jest resetowany. Nowy rekord zostaje zapisany pod obliczonym adresem, a sam adres zostaje zapisany na początku pamięci, gdzie przechowywany jest adres najnowszego rekordu. Ponieważ pętla główna kręci się wiele razy na sekundę, fakt zapisu zapamiętywany jest w zmiennej recordWritten. Jest ona resetowana dopiero, gdy zmieni się liczba sekund.



Rys. 1

Na LCD wyświetlany jest albo aktualny czas i temperatura, albo dane historyczne. Służy do tego ta sama funkcja displayRecord(), do której w zależności od trybu przekazywany jest wskaźnik albo na zmienną temperatureRecord, albo na zmienną historicRecord. Zmienna historicRecord wypełniana jest danymi odczytanymi z EEPROM-u. W celu wyświetlenia symbolu stopni, do wyświetlacza wysyłany jest bajt DFh, ponieważ pod tym adresem wyświetlacz ma zdefiniowany symbol przypominający symbol stopnia. Jest to alternatywa dla samodzielnego definiowania symbolu.

W trybie przeglądania zapisanych danych, gdy zmienna browsing ustawiona jest na 1, przyciskami S1 i S5 można przechodzić między rekordami w przód i w tył. W zależności od przycisku zmieniany jest adres, z którego wczytywane są dane do zmiennej historicRecord. Przycisk S16 powoduje opuszczenie przeglądania i przejście do podglądu danych bieżących. W tym trybie przycisk S1 włącza ustawianie daty i czasu, a przycisk S2 włącza przeglądanie zapisanych danych. Ustawianie daty i czasu realizowane jest znaną już funkcją setDateTime(). Funkcja ta nie była modyfikowana i jest funkcją blokującą, podczas ustawiania daty i czasu nie są więc wykonywane pomiary temperatury i nie są zapisywane w EEPROM-ie. Nie jest to jednak duży problem, bo skoro przestawiamy datę i czas, to zwykle dlatego, że nie są prawidłowe, a więc nie ma sensu ich zapisywać. Potrzeba ustawiania zachodzi zresztą rzadko, szczególnie jeśli mamy założoną baterię podtrzymującą.

Fot. 1

Celem drugiego ćwiczenia było wyświetlanie godziny włączenia i wyłączenia płytki testowej oraz zapis tych danych w pamięci Flash. Czas włączenia i wyłączenia zasilania można pobrać z odpowiednich rejestrów RTC kostki MCP79410, omówionych w lekcji 15. Dla momentu zaniku zasilania są to cztery rejestry, począwszy od PWRDNMIN, a dla powrotu zasilania również cztery, od PWRUPMIN. Do naszej biblioteki układu MCP79410 możemy więc dopisać funkcje odczytujące te rejestry (**listing 2**). Przy funkcjach obsługujących bieżący czas korzystaliśmy z typu RtcDateTimeRegisters, tutaj analogicznie zdefiniowany jest typ RtcPowerRegisters.

Główny program przedstawia **listing 3**. Po inicjalizacji peryferii sprawdzane jest, czy wystąpił zanik zasilania. Sygnalizuje to bit PWRFAIL w rejestrze RTCWKDAY. Jeśli bit jest ustawiony, czasy zaniku i powrotu zasilania są odczytywane z RTC i zapisywane w strukturze powerTime. Następnie jest ona zapisywana w pamięci Flash, w stronie numer 100. Funkcja writeFlash() wygląda jak w lekcji poprzedniej. Należy pamiętać o umieszczeniu jej w sekcji bootloadera poprzez konfigurację ustawień linkera. W dalszej kolejności czyszczony jest bit PWRFAIL, aby kostka RTC mogła zapisać czasy przy kolejnym wyłączeniu i włączeniu zasilania.

Listing 2

```
typedef struct {
 uint8_t PWRMIN;
 uint8_t PWRHOUR;
 uint8_t PWRDATE;
 uint8_t PWRMTH;
} RtcPowerRegisters;

void rtcReadPowerDownDateTime(DateTime * dateTime) {
 RtcPowerRegisters rtcPowerRegisters;
 rtcReadRegisters(MCP79410_PWRDNMIN, (uint8_t *) &rtcPowerRegisters,
sizeof(rtcPowerRegisters));
 dateTime->minutes = fromBCD(rtcPowerRegisters.PWRMIN & 0b01111111);
 dateTime->hours = fromBCD(rtcPowerRegisters.PWRHOUR & 0b00111111);
 dateTime->dayOfMonth = fromBCD(rtcPowerRegisters.PWRDATE & 0b00111111);
 dateTime->month = fromBCD(rtcPowerRegisters.PWRMTH & 0b00011111);
}

void rtcReadPowerUpDateTime(DateTime * dateTime) {
 RtcPowerRegisters rtcPowerRegisters;
 rtcReadRegisters(MCP79410_PWRUPMIN, (uint8_t *) &rtcPowerRegisters,
sizeof(rtcPowerRegisters));
 dateTime->minutes = fromBCD(rtcPowerRegisters.PWRMIN & 0b01111111);
 dateTime->hours = fromBCD(rtcPowerRegisters.PWRHOUR & 0b00111111);
 dateTime->dayOfMonth = fromBCD(rtcPowerRegisters.PWRDATE & 0b00111111);
 dateTime->month = fromBCD(rtcPowerRegisters.PWRMTH & 0b00011111);
}
```

Pętla główna sprawdza stan klawiatury. Gdy wciśnięty zostanie dowolny klawisz, następuje przełączenie między wyświetlaniem bieżącego czasu a czasami zapamiętanymi we Flashu. Bieżący czas pobierany i wyświetlany jest tak, jak w poprzednich przykładach. Natomiast do odczytu Flasha została napisana funkcja readFlash(), która wczytuje dane z Flasha makrem pgm_read_byte(). Za jej pomocą wypełniona zostaje struktura powerTime, której zawartość wyświetlana jest następnie na LCD. Na **fotografii 1** widać, że płytka testowa została wyłączona 1 lipca o 11.48 a następnie włączona 4 minuty później.

Listing 3

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

#define FLASH_PAGE_SIZE_BYTES 128

void BOOTLOADER_SECTION writeFlash(uint16_t pageAddress, uint16_t data[], uint16_t size);
void readFlash(uint16_t pageAddress, uint8_t data[], uint16_t size);

struct PowerTime {
 DateTime down;
 DateTime up;
};

int main(void) {
 i2cInit();
 keybInit();
 lcdInit();
 lcdInitPrintf();
 //sprawdź, czy był zanik napięcia
 uint8_t rtcwkday;
 rtcReadRegister(MCP79410_RTCWKDAY, &rtcwkday);
 if (rtcwkday & _BV(MCP79410_PWRFAIL)) {
  //odczytaj czasy wyłączenia i włączenia
  struct PowerTime powerTime;
  rtcReadPowerDownDateTime(&powerTime.down);
  rtcReadPowerUpDateTime(&powerTime.up);
  //zapisz czasy we Flashu
  writeFlash(FLASH_PAGE_SIZE_BYTES * 100, (uint16_t*) &powerTime, (sizeof(powerTime) + 1) / 2);
  //wyczyść flagę zaniku zasilania
  rtcwkday &= ~_BV(MCP79410_PWRFAIL);
  rtcWriteRegister(MCP79410_RTCWKDAY, rtcwkday);
 }
 uint8_t showCurrentTime = 1;
 while(1){
  if (readKeyboard()) {
   getKey();
   showCurrentTime = !showCurrentTime;
   lcdWriteCommand(LCD_COMMAND_CLEAR);
  }
  if (showCurrentTime) {
   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);
  } else {
   struct PowerTime powerTime;
   readFlash(FLASH_PAGE_SIZE_BYTES * 100, (uint8_t*) &powerTime, sizeof(powerTime));
   lcdGotoXY(0, 0);
   printf("OFF: %02d:%02d %02d-%02d", powerTime.down.hours, powerTime.down.minutes, powerTime.down.dayOfMonth, powerTime.down.month);
   lcdGotoXY(0, 1);
   printf("ON:  %02d:%02d %02d-%02d", powerTime.up.hours, powerTime.up.minutes, powerTime.up.dayOfMonth, powerTime.up.month);
  }
 }
}

void writeFlash(uint16_t pageAddress, uint16_t data[], uint16_t size) {
 boot_page_erase_safe(pageAddress);
 for (uint16_t i = 0; i < size; i++) {
  boot_page_fill_safe(pageAddress + i *2, data[i]);
 }
 boot_page_write_safe(pageAddress);
 boot_rww_enable_safe();
}

void readFlash(uint16_t pageAddress, uint8_t data[], uint16_t size) {
 for (uint16_t i = 0; i < size; i++) {
  data[i] = pgm_read_byte(pageAddress + i);
 }
}
```

Bity blokujące (lock bits)

Oprócz tzw. fuse bitów, mikrokontroler ATmega32 ma też bity blokujące, które konfigurują ochronę pamięci Flash i EEPROM. Jest ich sześć: LB1, LB2, BLB01, BLB02, BLB11 i BLB12. Pierwsze dwa bity blokujące kontrolują dostęp do Flasha i EEPROM-u z poziomu programatorów ISP i JTAG. Domyślnie oba te bity są niezaprogramowane (są ustawione na 1) i możemy zarówno odczytywać, jak i zapisywać obie pamięci. Natomiast gdy zaprogramujemy (ustawimy na 0) bit LB1, stracimy możliwość programowania pamięci Flash i EEPROM za pomocą programatorów. Możemy w ten sposób zabezpieczyć się przed przypadkowym nadpisaniem tych pamięci. Jeśli następnie zaprogramujemy bit LB2, również odczyt Flasha i EEPROM-u zostanie zablokowany. Funkcja ta jest przydatna, jeśli piszemy program dla kogoś i nie chcemy, aby miał możliwość odczytania i dekompilacji naszego programu. Jest to więc sposób na zabezpieczenie własności intelektualnej.

BLB02	BLB01	Ustawienie w AtmelStudio	Tryb ochrony
1	1	NO_LOCK	Brak ochrony, sekcja aplikacyjna dostępna jest dla instrukcji SPM i LPM
1	0	SPM_DISABLE	Zablokowany zapis sekcji aplikacyjnej instrukcją SPM
0	1	LPM_DISABLE	Zablokowany odczyt sekcji aplikacyjnej z poziomu instrukcji LPM wykonywanej z sekcji bootloaderowej
0	0	LPM_SPM_DISABLE	Zablokowany zapis sekcji aplikacyjnej instrukcją SPM oraz zablokowany odczyt sekcji aplikacyjnej z poziomu instrukcji LPM wykonywanej z sekcji bootloaderowej

Tabela 1

Aby te zabezpieczenia miały sens, nie ma możliwości przestawiania zaprogramowanych bitów z powrotem w stan niezaprogramowany. Jeśli chcemy zdjąć zabezpieczenia, musimy skorzystać z funkcji wykasowania całego mikrokontrolera. Wykonujemy to, przechodząc do sekcji Memories i klikając Erase now przy Erase Chip.

Uwaga! Fuse bity są w rzeczywistości malutkim kawałkiem pamięci typu Flash. W związku z tym bity blokujące wyłączają dostęp również do fuse bitów. Jeśli więc chcemy włączyć zabezpieczenia za pomocą bitów blokujących, musimy wcześniej zaprogramować wszystko inne: Flash, EEPROM (jeśli potrzeba) i fuse bity.

Bity blokujące możemy ustawiać w okienku programowania, wybierając pozycję Lock bits (**rysunek 1**). Podobnie jak w przypadku niektórych fuse bitów, Atmel Studio nie daje możliwości ustawiania pojedynczych bitów, ale oferuje menu z gotowymi opcjami. Są to: NO_LOCK (zabezpieczenia wyłączone), PROG_DISABLED (zapis wyłączony) i PROG_VER_DISABLED (zarówno odczyt, jak i zapis są wyłączone).

Pozostałe cztery bity blokujące dotyczą programowego dostępu do Flash za pomocą instrukcji SPM (zapis) i LPM (odczyt). Są przydatne, gdy chcemy chronić nasz program nie tylko przed odczytem/modyfikacją za pomocą programatora, ale też ewentualnego bootloadera zapisanego w pamięci Flash. Mogą też chronić bootloader przed dostępem z głównego programu. Bity BLB02 i BLB01 odpowiedzialne są za sekcję aplikacyjną (**tabela 1**), a BLB12 i BLB11 za sekcję bootloaderową (**tabela 2**).

Monitorowanie napięcia zasilania

Nasz mikrokontroler ma funkcję monitorującą napięcie zasilania (Brown-out Detection, w skrócie BOD). Polega ona na tym, że jeśli wartość napięcia spadnie

Uwaga! W naszym mikrokontrolerze watchdog wymaga resetowania od momentu, gdy go włączymy. Do włączenia watchdoga nie musimy nim się przejmować, tak jak we wszystkich programach, w których go nie używaliśmy. Jest to logiczne i intuicyjne. Jednak w nowszych mikrokontrolerach AVR jest nieco inaczej: gdy watchdog zresetuje mikrokontroler, sam pozostaje nadal aktywny. Mimo resetu i mimo że program jeszcze nie zdążył go włączyć. Załóżmy, że po starcie programu inicjalizujemy różne rzeczy i po 100 ms od startu/resetu następuje włączenie watchdoga oraz ustawienie go na 65 ms. Następnie resetujemy go co 50ms. Wszystko jest w porządku. Jeśli jednak jest to nowy mikrokontroler i nastąpi reset, po 65ms od resetu watchdog znów spowoduje reset. Mimo że nie zdążyliśmy go włączyć, a co dopiero zresetować. Mikrokontroler wpadnie w pętlę ciągłego resetowania się, zupełnie niezrozumiałą dla niedoświadczonego programisty. W związku z tym, jeśli włączanie watchdoga i rozpoczęcie resetowania go odbywa się stosunkowo późno, trzeba na samym początku programu umieścić wyłączenie watchdoga. W skrajnym przypadku, gdy okres watchdoga jest bardzo krótki, może to nie wystarczyć. Przyczyną jest dodawany przez kompilator kod inicjalizacyjny, uruchamiany jeszcze przed funkcją main(), którego wykonanie też wymaga chwili. Wówczas konieczne jest umieszczenie funkcji wyłączającej watchdoga w specjalnej sekcji .init3.

```
void wdt_init(void) __attribute__((naked)) __attribute__((section(".init3")));
void wdt_init(void)
{
  MCUSR = 0;
  wdt_disable();
}
```

poniżej określonego progu, następuje reset mikrokontrolera. Zabezpiecza to przed zawieszeniem się mikrokontrolera spowodowanym skokiem napięcia. Oczywiście reset jest zaburzeniem pracy mikrokontrolera, ale z punktu widzenia niezawodności restart programu jest dużo lepszy od zawieszenia się i czekania na reset przez użytkownika. Urządzenie powinno być skonstruowane oraz zaprogramowane w taki sposób, aby w przypadku przerwy w zasilaniu mogło możliwie bezboleśnie wznowić pracę.

Funkcję BOD włączamy, programując (ustawiając na zero) fuse bit BODEN. Dostępny jest też fuse bit BODLEVEL wyznaczający napięcie progowe, przy którym nastąpi reset. Jeśli będzie on niezaprogramowany (ustawiony na 1), progiem będzie napięcie ok. 4V (3,6V – 4,2V). Zaprogramowanie fuse bitu obniży próg do ok. 2,7V (2,5V – 2,9V). Jak widać, napięcie przełączające nie jest dokładnie określone i zależy od konkretnego egzemplarza mikrokontrolera. W praktyce ta niedokładność nie ma większego znaczenia. Niezależnie od wartości progu, funkcja BOD ma histerezę o wartości 50mV. Jeśli np. dana kostka ma napięcie progowe wynoszące 3,90V, to mikrokontroler wejdzie w stan resetu, gdy napięcie zasilające spadnie poniżej 3,875V, a wyjdzie z resetu i zacznie pracę, gdy napięcie wzrośnie ponad 3,925V. W ten sposób mikrokontroler nie będzie się ustawicznie resetował, gdy napięcie zasilające będzie utrzymywało się w okolicach progu.

Tabela 3

WDP2	WDP1	WDP0	Timeout (3V)	Timeout (5V)
0	0	0	17,1ms	16,3ms
0	0	1	34,3ms	32,5ms
0	1	0	68,5ms	65ms
0	1	1	114ms	130ms
1	0	0	270ms	260ms
1	0	1	550ms	520ms
1	1	0	1,1s	1,0s
1	1	1	2,2s	2,1s

Watchdog

Nie tylko wahnięcia napięcia zasilającego mogą spowodować zawieszenie się mikrokontrolera. Zwykle wina leży po stronie programisty: w braku przewidywania określonych okoliczności, warunków brzegowych czy w nieumiejętnym korzystaniu ze wskaźników. Jednak jeśli nawet program jest w 100% poprawny, działanie mikrokontrolera może zostać zakłócone przez inne czynniki, jak np. silne zakłócenia elektromagnetyczne czy promieniowanie kosmiczne. W każdym razie jeśli urządzenie ma być niezawodne, szczególnie gdy będzie działać w trudno dostępnym miejscu, potrzebny jest mechanizm zwiększający odporność na zawieszenie się mikrokontrolera. Takim mechanizmem, obok BOD, jest watchdog.

Watchdog, czyli pies stróżujący, pilnuje pracy mikrokontrolera i wymaga okresowego karmienia. Gdy nie dostanie jedzenia na czas, stwierdza, że z mikrokontrolerem jest coś nie tak i resetuje go. Watchdog ma postać timera taktowanego z wewnętrznego generatora. Karmienie polega więc na okresowym resetowaniu licznika watchdoga. Jeśli nie zdążymy zrobić tego na czas, licznik przepełni się i watchdog spowoduje reset mikrokontrolera. Dobrą analogią jest też bomba zegarowa, którą trzeba odpowiednio szybko rozbroić. Ile mamy czasu? Watchdog taktowany jest sygnałem 1MHz, pochodzącym z wewnętrznego generatora. Sygnał zegarowy przechodzi po drodze przez konfigurowalny dzielnik, możemy więc wybrać czas, jaki będzie musiał upłynąć, zanim licznik watchdoga się przepełni. Stopień podziału dzielnika wyznaczają bity WDP2..0 w rejestrze WDTCR zgodnie z **tabelą 3**.

Ponieważ sygnał zegarowy pochodzi z generatora RC, jego częstotliwość zmienia się nieco wraz z napięciem zasilania i temperaturą, zależy też od konkretnego egzemplarza mikrokontrolera. Niemniej i tak resetowanie watchdoga nie powinno się odbywać w ostatniej chwili, ale z odpowiednim zapasem, np. w połowie okresu.

Jak jednak w praktyce zrealizować resetowanie watchdoga? Skoro ma być wykonywane cyklicznie, to może skonfigurować timer i resetować w przerwaniu z timera? Nie miałoby to jednak większego sensu, bo przerwanie uruchamiałoby się nawet, gdyby program np. wpadł w nieskończoną pętlę, w której nieprawidłowo sprawdzane są warunki wyjścia. Gdzie więc umieścić resetowanie watchdoga? Ogólnie w tych miejscach, w których program, zakładając normalną pracę, zatrzymuje się na dłuższy moment. Będzie to więc główna pętla programu oraz inne pętle, które na coś oczekują, np. na wciśnięcie klawisza. W związku z tym tworząc lub modyfikując program, trzeba pamiętać o watchdogu

Tabela 2

BLB12	BLB11	Ustawienie w AtmelStudio	Tryb ochrony
1	1	NO_LOCK	Brak ochrony, sekcja bootloaderowa dostępna jest dla instrukcji SPM i LPM.
1	0	SPM_DISABLE	Zablokowany zapis sekcji bootloaderowej instrukcją SPM
0	1	LPM_DISABLE	Zablokowany odczyt sekcji bootloaderowej z poziomu instrukcji LPM wykonywanej z sekcji aplikacyjnej
0	0	LPM_SPM_DISABLE	Zablokowany zapis sekcji bootloaderowej instrukcją SPM oraz zablokowany odczyt sekcji bootloaderowej z poziomu instrukcji LPM wykonywanej z sekcji aplikacyjnej

i w razie potrzeby dodawać kolejne miejsca resetujące. Może to być uciążliwe, bo jeśli zapomnimy o watchdogu, to po wprowadzeniu jakiejś zmiany w programie nagle pojawią się nam nieoczekiwane resety. Jeśli jednak watchdog ma pilnować naszego programu, to my też musimy się trochę wysilić i przeanalizować, które fragmenty mogą się wykonywać dłużej i dodać do nich wywoływanie resetowania watchdoga.

Aby włączyć watchdoga, również korzystamy z rejestru WDTCR, który oprócz bitów wyznaczających okres watchdoga zawiera też bity WDTOE i WDE. Do włączenia wystarczy wpisać jedynkę do bitu WDE. Żeby jednak watchdog nie został przypadkowo wyłączony, procedura wyłączania jest nieco bardziej skomplikowana. W tym celu należy wpisać jedynki do bitów WDTOE i WDE (mimo że chcemy wyłączyć watchdoga) a następnie, w ciągu czterech cykli zegara procesora, wpisać zero do WDE.

Wraz z kompilatorem otrzymujemy bibliotekę <avr/wdt.h>, dzięki której możemy łatwo obsługiwać watchdoga. Zawiera ona trzy przydatne makra:

- wdt_enable(timeout)
- wdt_disable()
- wdt_reset()

Służą one do włączania, wyłączania i resetowania watchdoga. Do makra wdt_enable() przekazujemy parameter określający, co ile maksymalnie trzeba resetować watchdoga. W zależności od potrzebnego czasu przekazujemy jedno z makr: WDTO_15MS, WDTO_30MS, WDTO_60MS, WDTO_120MS, WDTO_250MS, WDTO_500MS, WDTO_1S, WDTO_2S, WDTO_4S, WDTO_8S. Dwa ostatnie makra są dostępne dla mikrokontrolerów mających bit WDP3 w WDTCR.

Przeanalizujmy prosty przykład pokazujący działanie watchdoga. Został przedstawiony na poniższym **listingu 4**.

```
#include <avr/io.h>
#include <util/delay.h>
#include <avr/wdt.h>
#include "lcd.h"
#include "keyb.h"

int main(void) {
 wdt_enable(WDTO_120MS);
 keybInit();
 lcdInit();
 lcdInitPrintf();
 printf("Reset!");
 while(1){
  uint8_t key = 0;
  while (!(key = readKeyboard())) {
   wdt_reset();
  }
  while (readKeyboard()) {
   wdt_reset();
  }
  lcdWriteCommand(LCD_COMMAND_CLEAR);
  printf("%d", key);
  if (key == 16) {
   _delay_ms(250);
  }
 }
}
```

Najpierw inicjowany jest watchdog z limitem czasu wynoszącym 120 milisekund. Następnie inicjowana jest klawiatura i wyświetlacz LCD. Program wyświetla napis Reset! oznaczający zresetowanie się mikrokontrolera i rozpoczęcie wykonywania programu. W głównej pętli odczytywany jest numer wciśniętego przycisku na klawiaturze i wyświetlany na LCD. Odczyt stanu klawiatury realizowany jest za pomocą funkcji readKeyboard() i następuje w dwóch etapach: oczekiwanie na wciśnięcie klawisza i oczekiwanie na zwolnienie klawisza. Podczas oczekiwania resetowany jest licznik watchdoga. Nie można było zastosować wykorzystywanej zwykle funkcji getKey(), gdyż nie resetuje ona watchdoga i nastąpiłby niepożądany reset mikrokontrolera. Jeśli wciśnięty zostanie klawisz S16, wykonywane jest opóźnienie wynoszące 250 milisekund. Jest to symulacja zbyt długo wykonującej się funkcji. Ponieważ watchdog ustawiony jest na 120 milisekund, następuje reset. Na wyświetlaczu zobaczymy więc przez krótki moment liczbę 16 i zaraz potem napis Reset! świadczący o zadziałaniu watchdoga i wykonanym resecie mikrokontrolera.

Nasz prosty przykład pozwala nie tylko przetestować działanie watchdoga, ale także zwraca uwagę na konieczne modyfikacje w programie wynikające z użycia watchdoga. Tutaj jest to zastąpienie wywołania funkcji getKey() dwiema pętlami sprawdzającymi stan klawiatury, a przy okazji resetującymi licznik watchdoga. Alternatywnie możemy zmodyfikować funkcję getKey(), umieszczając w niej wywołania wdt_reset(). W ten sposób resetowanie watchdoga zniknie nam z głównego programu i trafi do biblioteki klawiatury. Ma to swoje plusy, główny kod będzie wyglądał bardziej elegancko. Z drugiej strony przy bardziej rozbudowanym kodzie resetowanie watchdoga i tak będzie trzeba dodać w innych miejscach. Ponadto z resetowaniem watchdoga ukrytym w bibliotekach łatwiej zapomnieć, gdzie resetowanie jest wykonywane a gdzie nie.

Zadanie

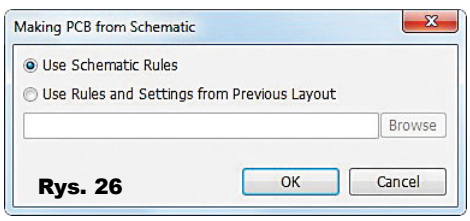
Watchdog może służyć nie tylko do kontroli stabilności programu, ale również jako programowe resetowanie mikrokontrolera. Jak można zmodyfikować przykład z listingu 4, aby klawisz S16 również powodował reset, ale bez włączania watchdoga na cały czas działania programu?

W materiałach dodatkowych do tego numeru EdW znajdują się projekty rozwiązań zadań domowych oraz przykład obsługi watchdoga z listingu 4.

Grzegorz Niemirowski
grzegorz@grzegorz.net

Ciąg dalszy ze strony 30

Wybór formatu pliku, w jakim ma być zapisany eksportowany w ten sposób schemat, odbywa się w oknie zapisu pliku z rozwijanej listy widocznej na **rysunku 25**. Nie różni się to znacząco od zapisu innych plików w systemie Windows i nie ma potrzeby obszerniejszego opisywania tej czynności.



Rys. 26

Przejście do edytora PCB Layout

Przejście do edytora schematów PCB Layout odbywa się skrótem klawiaturowym Ctrl+B. Tym razem otworzy się małe okienko widoczne na **rysunku 26** z pytaniem o użycie reguł ze schematu lub z poprzedniego projektu. Pozostawiamy domyślny wybór reguł ze schematu i zatwierdzamy przyciskiem OK. Otworzy się edytor PCB Layout, a na jego planszy projektowej pojawią się rozmieszczone elementy płytkowe w sposób zbliżony do rozmieszczenia na planszy schematów. Projektowaniem płytki zajmiemy się w kolejnej części tego kursu.

Krzysztof Kawa
kaawa@wp.pl