Kurs AVR – lekcja 17

Rozwiązania zadań z ostatniego odcinka

W poprzedniej lekcji pierwsze zadanie domowe dotyczyło sprawdzenia działania programowego zabezpieczenia EEPROM-u przed zapisem. Możemy to wykonać, sprawdzając powodzenie zapisu różnych bajtów przed i po włączeniu blokady poprzez ustawienie bitów BP0 i BP1 w rejestrze statusowym. Przykład przedstawia **listing 1.**

Listing 1

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

uint8_t testWrite(uint8_t expected);

int main(void) {
 spiInit();
 lcdInit();
 lcdInitPrintf();
 //wyłącz ochronę
 eepromWriteStatus(0);
 while (eepromIsBusy()) _delay_ms(1);
 printf(testWrite(12) ? "OK" : "ERR");
 lcdGotoXY(0, 1);
 //włącz ochronę
 eepromWriteStatus(_BV(_25LC080_BP1) | _BV(_25LC080_BP0));
 while (eepromIsBusy()) _delay_ms(1);
 printf(testWrite(34) ? "OK" : "ERR");
 while(1){ }
}

uint8_t testWrite(uint8_t expected) {
 eepromWriteByte(0, expected);
 while (eepromIsBusy()) _delay_ms(1);
 return (eepromReadByte(0) == expected);
}
```

Zadaniem funkcji pomocniczej testWrite() jest zapisanie bajtu, a następnie jego odczyt i sprawdzenie, czy odczytany bajt ma tę samą wartość co zapisany. Jeśli odczytana wartość będzie taka sama jak zapisana, funkcja zwraca 1, w przeciwnym wypadku 0. W funkcji main() wołamy funkcję testującą dwa razy: po wyłączeniu blokady i po zablokowaniu całego obszaru EEPROM-u. Jeśli zapis powiedzie się, wyświetlany jest napis OK, jeśli nie – napis ERR. Oczywiście w drugim zapisie musi być użyta inna wartość niż przy pierwszym, aby można było stwierdzić, że rzeczywiście nastąpiło zmodyfikowanie pamięci. Ponieważ przy drugim zapisie ochrona ma być włączona, spodziewanym wynikiem działania programu jest napis OK w pierwszej linijce wyświetlacza, a ERR w drugiej linijce.

Drugie zadanie bazowało na zamku szyfrowym z lekcji 10. W tamtym przykładzie kod/PIN był na stałe zaszyty w programie mikrokontrolera. Dysponując pamięcią EEPROM, możemy stworzyć zamek programowany, z możliwością zmiany PINu. Jak ma wyglądać zmiana? Możliwości jest wiele. Przyjmijmy, że będzie wywoływana przyciskiem S16. W rzeczywistym urządzeniu musiałby być on odpowiednio ukryty, np. wewnątrz zamkniętej obudowy. Wykorzystywane i tak są tylko przyciski S1–S10. Po wciśnięciu przycisku S16 można wpisać nowy PIN, który zapisywany jest w EEPROM-ie. Kod rozwiązania przedstawiono na **listingu 2**.

Listing 2

```
#include <avr/io.h>
#include <util/delay.h>
#include "keyb.h"
#include "led.h"
#include "25LC080.h"

int main(void) {
 spiInit();
 ledInit();
 keybInit();
 DDRC |= _BV(DDC0);
 uint8_t PIN[4];
 eepromReadBytes(0, PIN, sizeof(PIN));
 while(1) {
  ledClear();
  uint8_t match = 1;
  uint8_t newPIN[4];
  uint8_t newPINEntered = 0;
  //wczytanie PINu
  for (uint8_t i = 0; i < 4; i++) {
   uint8_t input = getKey();
   if (input == 16) {
    newPINEntered = 1;
    input = getKey();     }
   if (input == 10) input = 0;
   if (input != PIN[i]) match = 0;
   newPIN[i] = input;
   ledDispSymbol(LED_MINUS, 3 - i);  }
  //zapisz nowy PIN
  if (newPINEntered) {
   newPINEntered = 0;
   eepromWriteBytes(0, newPIN, sizeof(newPIN));
   for (uint8_t i = 0; i < sizeof(PIN); i++) { PIN[i] = newPIN[i]; } }
  //otwarcie zamka
  if (match) {
   PORTC |= _BV(PORTC0);
   _delay_ms(1000);
   PORTC &= ~_BV(PORTC0);
  }  }  }
```

Ogólnie szkielet programu pozostaje taki sam. Na początku odbywa się inicjalizacja peryferii. Nowością jest wczytanie PIN-u z EEPROM-u zawartego w kostce 25LC080. W głównej pętli pojawiają się nowe zmienne: newPIN oraz newPINEntered. Dalej znajduje się lekko zmodyfikowana pętla wczytująca cyfry. Dodane zostało sprawdzanie, czy naciśnięty został klawisz S16. Jeśli tak, ponownie wywoływana jest funkcja getKey(), aby wczytać pierwszą cyfrę nowego PIN-u. Pierwszą cyfrą nie może bowiem być liczba 16. Kolejne cyfry wczytywane są jak dotychczas, ale dodatkowo zapamiętywane są w tablicy newPIN. Dzieje się to niezależnie od tego, czy wciśnięty został przycisk S16, ale w niczym to nie przeszkadza, bowiem dalej i tak znajduje się if sprawdzający zmienną newPINEtered. Jeśli jest ona ustawiona na 1 dzięki klawiszowi S16, następuje ustawienie nowego PIN-u. Zostaje on zapisany do EEPROM-u oraz skopiowany do podstawowej tablicy PIN. W przykładzie dioda sygnalizująca otwarcie zamka została przeniesiona na port C, ponieważ port B jest w całości wykorzystany na potrzeby wyświetlacza LED oraz szyny SPI.

Wbudowany EEPROM

Kontynuujemy temat pamięci nieulotnych. W tym odcinku omówimy pamięć EEPROM zawartą w naszym mikrokontrolerze ATmega32. Ma ona rozmiar 1024 bajtów (1 kB), a jej żywotność wynosi 100 tysięcy cykli zapisu. W przeciwieństwie do pamięci zawartych w układach scalonych przedstawionych w poprzednich lekcjach, EEPROM w ATmega32 nie jest podzielony na strony. Możemy więc odczytywać i zapisywać niezależnie dowolne bajty przy wykorzystaniu kilku rejestrów:

- EEARH i EEARL – rejestry adresu
- EEDR – rejestr danych
- EECR – rejestr kontrolny

Ponieważ rozmiar EEPROM-u to 1024 bajty, potrzeba 10 bajtów, aby móc adresować poszczególne komórki ($2^{10} = 1024$). Z tego względu do adresowania wymagane są dwa rejestry: EEARL przechowujący 8 młodszych bitów adresu oraz EEARH przechowujący 2 najstarsze bity. Taka sytuacja nie jest nowością, spotkaliśmy się z nią np. przy przetworniku analogowo-cyfrowym. Tak jak wtedy, tak i teraz biblioteki dostarczone z Atmel Studio zapewniają nam makro ułatwiające dostęp do tego typu rejestrów. Makro adresowe dla EEPROM-u nosi nazwę EEAR i z punktu widzenia programistycznego możemy je traktować jako jeden 10-bitowy rejestr.

Rejestr EEDR pośredniczy w wymianie danych z EEPROM-em. Przy zapisie do pamięci umieszczamy w nim żądaną wartość, a przy odczycie pobieramy.

W rejestrze EECR znajdują się cztery bity:

- EERIE – włącza generowanie przerwania gotowości EEPROMu
- EEMWE – włącza działanie bitu EEWE
- EEWE – zapis jedynki rozpoczyna procedurę zapisu bajtu
- EERE – zapis jedynki rozpoczyna procedurę odczytu bajtu

Bit EEMWE jest swego rodzaju zabezpieczeniem przed przypadkowym zapisem do EEPROM-u. Po ustawieniu zeruje się on automatycznie w ciągu czterech cykli zegarowych. W związku z tym, aby móc przeprowadzić zapis, bit EEWE trzeba ustawić od razu po ustawieniu EEMWE. Jeśli

minie zbyt wiele czasu między ustawieniem tych dwóch bitów, zapis sie nie powiedzie. Jest to szczególnie ważne, gdy używamy przerwań. Jeśli między ustawieniem bitów EEMWE i EEWE uruchomiona zostanie procedura obsługi przerwania, to nim się ona zakończy, bit EEMWE zostanie wyzerowany i ustawienie bitu EEWE nie uruchomi zapisu. Konieczne jest więc chwilowe wyłączenie przerwań.

Aby zapisać bajt do EEPROM-u trzeba:
1. Poczekać na zakończenie ewentualnego poprzedniego zapisu, sprawdzając stan bitu EEWE
2. Zapisać adres do EEAR
3. Zapisać daną do EEDR
4. Ustawić bit EEMWE
5. Ustawić bit EEWE

Procedura odczytu jest podobna:
1. Poczekać na zakończenie ewentualnego poprzedniego zapisu, sprawdzając stan bitu EEWE
2. Zapisać adres do EEAR
3. Ustawić bit EERE
4. Odczytać rejestr EEDR

Obsługa EEPROM-u w naszym mikrokontrolerze zamknie się więc w dwóch prostych funkcjach (**listing 3**). Z ich wykorzystaniem możemy napisać funkcje do odczytu i zapisu tablic bajtów (**listing 4**):

Nasze funkcje mają takie same nazwy i argumenty jak funkcje do obsługi kostki 25LC080, które napisaliśmy w poprzedniej lekcji. Możemy używać ich praktycznie w ten sam sposób, różnica polega tylko na braku funkcji eepromIsBusy(). W związku z tym do przetestowania działania EEPROM-u możemy wykorzystać ten sam program testowy, z niewielkimi tylko zmianami. Aby jednak nie powtarzać przykładów, napiszmy prosty program testowy z wykorzystaniem struktur (**listing 5**).

Funkcje do obsługi EEPROM-u zostały przeniesione do oddzielnego pliku eeprom.c. Ich nagłówki są w pliku eeprom.h. W głównym programie inicjujemy LCD oraz definiujemy prostą strukturę. Następnie deklarowana jest struktura in

Listing 5

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

int main(void) {
 lcdInit();
 lcdInitPrintf();
 struct test {
  uint8_t a;
  uint32_t b;
  char c[10];  };
 struct test in = {7, 1000000, "EdW"};
 eepromWriteBytes(0, (uint8_t*)&in, sizeof(in));
 struct test out;
 eepromReadBytes(0, (uint8_t*)&out, sizeof(out));
 printf("a=%d b=%ld", out.a, out.b);
 lcdGotoXY(0, 1);
 printf("c=%s", out.c);
 while(1){ }  }
```

inicjowana danymi testowymi. Adres struktury przekazujemy do funkcji eepromWriteBytes(), która traktuje ją jak tablicę bajtów i zapisuje do EEPROM-u. Stąd rzutowanie na typ uint8_t*. Następnie deklarowana jest struktura out, do której wczytujemy zapisane dane. Wyświetlenie zawartości struktury out na LCD pozwala na zweryfikowanie poprawności zapisu i odczytu. Przy wyświetlaniu pola typu uint32_t używamy specyfikatora %ld, ponieważ sam specyfikator %d służy do wyświetlania typu int, który na platformie AVR jest 16-bitowy (typy int16_t, uint16_t). Dodajemy więc literkę l (long), aby poprawnie wyświetlić typ 32-bitowy.

```
uint8_t eepromReadByte(uint16_t address) {
 //zaczekaj na zakończenie poprzedniej operacji
 while(EECR & _BV(EEWE));
 //ustaw adres
 EEAR = address;
 //rozpocznij odczyt
 EECR |= _BV(EERE);
 //zwróć daną
 return EEDR; }

void eepromWriteByte(uint16_t address, uint8_t data) {
 //zaczekaj na zakończenie poprzedniej operacji
 while(EECR & _BV(EEWE));
 //ustaw adres i daną
 EEAR = address;
 EEDR = data;
 //odblokuj zapis
 EECR |= _BV(EEMWE);
 //rozpocznij zapis
 EECR |= _BV(EEWE); }
```

Listing 3

```
void eepromReadBytes(uint16_t address, uint8_t data[], uint16_t size) {
 for (uint16_t i = 0; i < size; i++) {
  data[i] = eepromReadByte(address + i); }  }

void eepromWriteBytes(uint16_t address, uint8_t data[], uint16_t size) {
 for (uint16_t i = 0; i < size; i++) {
  eepromWriteByte(address + i, data[i]); } }
```

Listing 4

Wbudowana pamięć Flash

Z pamięci Flash korzystamy od początku kursu. To w niej za pomocą programatora zapisywane są nasze skompilowane programy, które mikrokontroler następnie odczytuje i wykonuje. Dotychczas jednak nie przyglądaliśmy się bliżej tej pamięci. Do czego jeszcze możemy ją wykorzystać?

Flash, jak wiemy, jest pamięcią nieulotną. Przechowywane w nim dane nie znikają po odłączeniu zasilania. Może więc służyć jako zamiennik wbudowanego EEPROM-u jeśli jego rozmiar jest niewystarczający. W ATmega32 EEPROM ma 1 kB, a Flash ma 32 kB. Flash jest więc znacznie większy, przy czym oczywiście jego część będzie zajęta przez działający na mikrokontrolerze program. Z drugiej strony Flash szybciej się zużywa: producent gwarantuje 10 tysięcy cykli zapisu dla każdej komórki pamięci. Jest to więc dziesięciokrotnie mniej niż w przypadku EEPROM-u. Wybierając więc między tymi dwiema pamięciami, trzeba wziąć pod uwagę potrzebną pojemność oraz największą liczbę zapisów do tego samego miejsca w pamięci, jaka wystąpi podczas pracy urządzenia. Decydując się na wbudowaną pamięć Flash, trzeba też pamiętać, że została przeznaczona przede wszystkim na kod wykonywalny, a nie na dane użytkownika. Dostęp do Flasha w ATmega32 jest zorganizowany właśnie pod kątem zapisywania w nim programów. Na czym to jednak polega, skoro program wgrywamy po prostu programatorem ISP lub JTAG?

Bootloader i programowy zapis Flasha

Programator jest narzędziem, z którego korzystamy jako programiści/konstruktorzy urządzenia z mikrokontrolerem. Dzięki niemu łatwo i szybko możemy wgrywać różne programy i testować poprawki. Jeśli jednak urządzenie z mikrokontrolerem budujemy dla kogoś i chcemy, aby miało ono możliwość aktualizacji oprogramowania mikrokontrolera, to raczej nie możemy oczekiwać, że osoba ta będzie miała odpowiedni programator i będzie umiała z niego korzystać. Aktualizacja oprogramowania powinna być jak najprostsza dla użytkownika. Jak więc załadować program do mikrokontrolera bez programatora? Musiałby na nim działać inny program, który przez jeden z interfejsów komunikacyjnych, np. port szeregowy, pobierze właściwy program i zapisze go we Flashu. Taki program nazywamy bootloaderem.

Bootloader znajduje się w wydzielonym obszarze pamięci Flash. Może zostać uruchomiony przez skok z programu głównego lub też automatycznie przy resecie. Ta druga możliwość konfigurowana jest za pomocą fusebitu BOOTRST. Jeśli jest niezaprogramowany (ustawiony na 1), procesor po wyjściu ze stanu resetu rozpoczyna wykonywanie kodu od początku aplikacyjnego obszaru pamięci Flash, czyli od adresu 0000h. Tak jest domyślnie i z tego ustawienia korzystamy – po resecie uruchamia się nasz program. Jeśli jednak BOOTRST zostanie zaprogramowany (ustawiony na 0), po resecie procesor skoczy do obszaru bootloadera. Adres początku bootloadera, będący jednocześnie granicą między obszarem aplikacji a obszarem bootloadera, jest konfigurowalny za pomocą fuse bitów BOOTSZ1 i BOOTSZ0 zgodnie z **tabelą 1**.

Jak widać w tabeli, występują dwa rodzaje adresowania Flasha. Wynika to z faktu, że pamięć Flash w naszym mikrokontrolerze nie jest zorganizowana w postaci komórek jednobajtowych, ale dwubajtowych (16-bitowych). Są to najmniejsze elementy pamięci Flash i nazywane słowami. Ponieważ słowa

są dwubajtowe, ich adresy będą dwa razy mniejsze od adresów bajtów. Jeśli np. obszar bootloadera będzie się zaczynał pod adresem 7C00h, to jego pierwsze dwa bajty znajdą się w słowie pamięci Flash o adresie 3E00h. Należy o tym pamiętać, ponieważ w niektórych miejscach stosowane jest bajtowe adresowanie Flasha, a w innych adresowanie słowami. Z adresowaniem słowami możemy się zetknąć, np. ustawiając fusebity BOOTSZ1 i BOOTSZ0 z poziomu Atmel Studio (**rysunek 1**). Obszar, jaki może zajmować bootloader, nie jest duży, to niewielki ułamek (1/64–1/8) pamięci Flash (32kB). Bootloader ma być tylko programem pomocniczym. Po włączeniu zasilania mikrokontrolera (wyjściu z resetu) zostaje uruchomiony i albo od razu uruchamia główny program, albo rozpoczyna pobieranie nowego programu i zapisywanie go we Flashu.

Aby bootloader mógł zapisywać kod programu we Flashu, mikrokontroler musi udostępniać odpowiednią instrukcję. Jest to instrukcja SPM (Store Program Memory). Teraz najważniejsze: instrukcja SPM może być wywołana tylko z kodu znajdującego się w obszarze bootloadera, wyznaczonego fuse bitami BOOTSZ1 i BOOTSZ0. Oznacza to, że jeśli nawet nie potrzebujemy bootloadera, tylko po prostu chcemy napisać program zapisujący dane do Flasha, to musi być on podzielony na dwie części. Główna część programu znajdzie się w części aplikacyjnej, a część odpowiedzialna za zapis Flasha w części bootloaderowej.

Zmiana położenia kodu we Flashu

O tym, w jakim miejscu pamięci znajdzie się dana funkcja z naszego programu, decyduje linker. Zwykle nie poświęcamy mu zbyt wiele uwagi. Klikamy polecenie Build i linker uruchamia się automatycznie, gdy tylko kompilator zakończy swoją pracę. Po co właściwie jest linker? Kompilator tłumaczy kod w języku C na kod maszynowy, czyli bajty rozumiane przez mikrokontroler jako instrukcje i ich argumenty. Przetłumaczone funkcje trzeba potem odpowiednio poukładać w pamięci, a w miejscach, w których jedna funkcja wywołuje drugą, wstawić skoki pod odpowiednie adresy – za to odpowiada linker.

Tabela 1

BOOTSZ1	BOOTSZ0	Rozmiar obszaru bootloadera	Adres bootloadera (adresacja bajtowa)	Adres bootloadera (adresacja słowami)
1	1	512 B / 256 słów	7E00h	3F00h
1	0	1 kB / 512 słów	7C00h	3E00h
0	1	2 kB / 1024 słowa	7800h	3C00h
0	0	4 kB / 2048 słów	7000h	3800h

W swojej pracy linker opiera się na konfiguracji tzw. sekcji pamięci. Gdy w Atmel Studio kompilujemy (budujemy) nasz program, w okienku Output pojawia się m.in. podsumowanie rozmiarów sekcji:

```
   text    data     bss     dec     hex filename
   5026      10      17    5053    13bd test.elf
```

Sekcja .text (nazwy sekcji zwykle poprzedzamy kropką) to skompilowany kod wykonywalny naszego programu. W przykładzie widzimy, że składające się na niego instrukcje zajęły 5026 bajtów. Sekcje .data i .bss to zmienne globalne oraz statyczne zmienne lokalne. Różnią się tym, że .data przechowuje zmienne zainicjowane (np. int a = 5;), a .bss niezainicjowane. W pamięci Flash znajdzie się więc sekcja .text oraz dane inicjalizacyjne dla sekcji .data, która sama w sobie, podobnie jak .bss, znajdzie się w SRAM. W tabelce widzimy też kolumny dec i hex. Jest to po prostu suma rozmiarów sekcji, zapisana dziesiętnie i szesnastkowo.

Aby przenieść nasz kod w wybrane miejsce pamięci Flash, mamy dwa wyjścia. Jedno to zmiana położenia sekcji .text. Wówczas cały kod wykonywalny naszego programu zostanie przeniesiony. Jest to rozwiązanie dobre, gdy piszemy bootloader lub mały program, który w całości zmieści się w obszarze bootloadera. Jeśli jednak nie piszemy bootloadera, a tylko chcemy wzbogacić nasz program o zapis danych we Flashu, lepiej będzie przenieść tylko te funkcje, które na tej pamięci operują. W tym celu trzeba stworzyć nową sekcję, poinformować linker, pod jakim ma się ona znaleźć adresem oraz wskazać, które funkcje mają trafić do tejże sekcji. Aby oznaczyć wybrane funkcje, stosujemy tzw. atrybuty. Wygląda to w ten sposób:

```
void __attribute__ ((section (".bootloader"))) writeFlash(void);
```

W przykładzie tym nadaliśmy funkcji writeFlash() atrybut section o treści „.bootloader”. Aby zapis był nieco krótszy, możemy skorzystać z makra BOOTLOADER_SECTION zdefiniowanego w bibliotece <avr/boot.h>. Wówczas deklaracja będzie wyglądała następująco:

```
void BOOTLOADER_SECTION writeFlash(void);
```

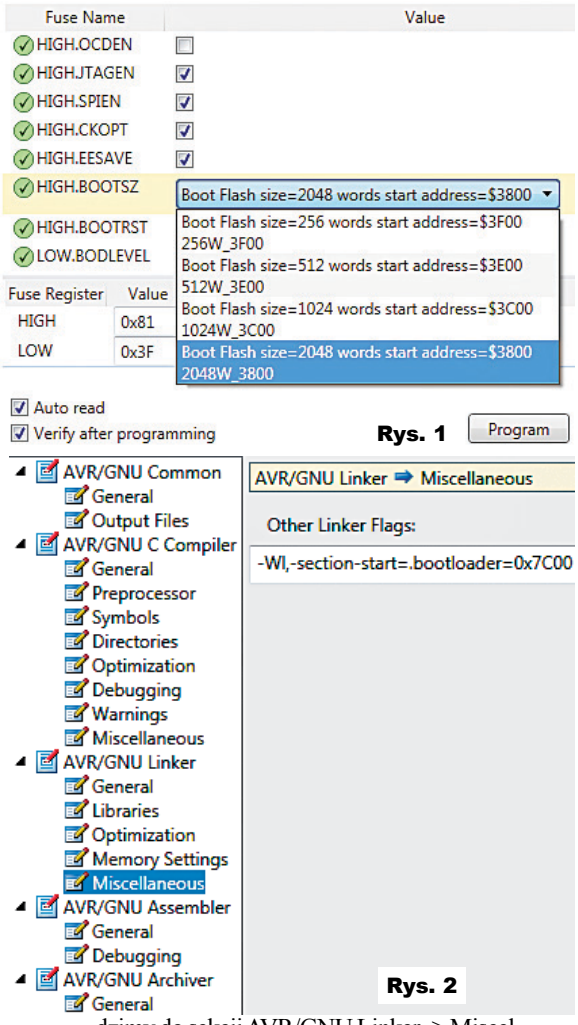
Pozostaje jeszcze przekazać linkerowi nowy adres sekcji. W tym celu otwieramy właściwości projektu i przechodzimy do sekcji AVR/GNU Linker -> Miscellaneous (**rysunek 2**). Znajduje się tam pole, do którego możemy dopisać dodatkowe parametry. Korzystaliśmy już z niego, gdy chcieliśmy wyświetlać na LCD liczby ułamkowe i konieczne było dolinkowanie odpowiedniej biblioteki. Załóżmy, że za pomocą fuse bitów ustawiliśmy początek obszaru bootloadera na 7C00h. Dopisujemy więc:

```
-Wl,-section-start=.bootloader=0x7C00
```

W ten sposób przekazujemy linkerowi adres początku sekcji o nazwie .bootloader. Atmel Studio daje nam jeszcze jedną drogę zdefiniowania adresu sekcji: wybranie pozycji Memory Settings. Mamy tutaj trzy listy (dla Flasha, SRAM i EEPROM), do których możemy wpisać, gdzie mają leżeć dane sekcje. Uwaga! Lista Flash segment operuje na adresacji słowami. Żeby nasza sekcja .bootloader znalazła się pod bajtowym adresem 7C00h, musimy więc wpisać (**rysunek 3**):

```
.bootloader=0x3E00
```

Wówczas do opcji linkera zostanie dodana linijka dokładnie taka, jak przedstawiona wcześniej, z adresem 7C00h. Możemy to sprawdzić, klikając w główną sekcję AVR/GNU Linker i patrząc na okienko All



Rys. 1

Rys. 2

Options. W każdym razie oba sposoby są równoważne i można wybrać sobie jeden lub drugi.

Funkcje biblioteczne do obsługi Flasha

Jak wiemy, do obsługi Flasha służy instrukcja SPM. Z poziomu języka C nie możemy jednak bezpośrednio wywoływać instrukcji procesora, konieczne są wstawki w języku asemblera. Na szczęście nie musimy uczyć się asemblera ani jak wstawiać kod asemblerowy do kodu C. Biblioteka <avr/boot.h> dostarcza nam gotowe makra, z których możemy korzystać jak z funkcji. Są to:

- boot_page_erase(address)
- boot_spm_busy_wait()
- boot_page_fill(address, word)
- boot_page_write(address)
- boot_rww_enable()

Flash w ATmega32 podzielony jest na 256 stron po 64 słowa (128 bajtów). Zapis odbywa się właśnie stronami. Aby zapisać dane do strony, najpierw kasujemy ją za pomocą makra boot_page_erase(). Następnie za pomocą boot_page_fill() wypełniamy słowo po słowie specjalny bufor danymi, które chcemy zapisać. Gdy strona jest wykasowana a dane czekają w buforze, możemy wykonać zapis makrem boot_page_write(). Na koniec trzeba wywołać makro boot_rww_enable(). Bez niego nie będzie możliwy odczyt zapisanych danych. Makro boot_spm_busy_wait() służy do oczekiwania na zakończenie operacji na Flashu, trzeba je wywoływać po kasowaniu i zapisie (po makrach boot_page_erase() i boot_page_write()). Alternatywnie można korzystać z pozostałych makr w wersjach bezpiecznych, czyli mających już wbudowane oczekiwanie na zakończenie poprzednich operacji na Flashu. Są to makra z dopiskiem _safe w nazwie: boot_page_erase_safe(address), boot_page_fill_safe(address, data), boot_page_write_safe(address) i boot_rww_enable_safe().

Napiszmy przykładową funkcję zapisującą dane w ustalonej stronie Flasha:

```
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();
}
```

Funkcja pobiera adres początku strony, dane w tablicy typu uint16_t oraz liczbę elementów tablicy. Jaki powinien być adres strony? Musimy zmieścić się między naszym programem a obszarem bootloadera. Jeśli program ma kilka kilobajtów, to możemy np. skorzystać z setnej strony pamięci Flash. Strony mają po 128 bajtów, więc adres będzie wynosił 12800 (3200h). Dla ułatwienia możemy sobie zadeklarować makro z rozmiarem strony:

```
#define FLASH_PAGE_SIZE_BYTES 128
```

Przykładowe wywołanie może wyglądać:

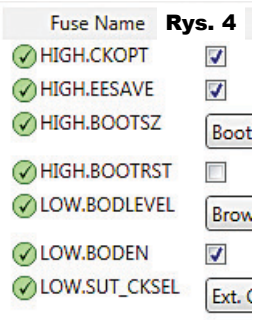
```
uint16_t testData[] = {1, 2, 3, 4};
writeFlash(FLASH_PAGE_SIZE_BYTES * 100, testData, 4);
```

Funkcja przyjmuje dane w postaci tablicy typu uint16_t. Jeśli chcielibyśmy zapisywać dane innego typu, mamy dwa wyjścia. Jedno to dostosować funkcję. Przykładowo, jeśli przekazywana miała być tablica typu uint8_t, wówczas trzeba będzie pobierać po 2 bajty i przekazywać do boot_page_fill_safe() jako jedno słowo 16-bitowe. Drugie rozwiązanie to zmiana sposobu przekazywania danych. Trzymając się przykładu z tablicą typu uint8_t, możemy jej adres zapisać we wskaźniku na typ uint16_t. Dodatkowo trzeba będzie zadbać o prawidłową wartość parametru size. Oznacza on liczbę elementów 16-bitowych, jest ich więc dwa razy mniej niż bajtów. Nie możemy jednak wykonać prostego dzielenia przez dwa. Przykładowo zapis 3 bajtów wymaga bowiem 2 słów. Jeśli wykonamy zwykłe dzielenie całkowite, wynik zostanie zaokrąglony w dół i z liczby 3/2 zostanie 1, a jest to za mało. Zostanie zapisane jedno słowo, czyli dwa bajty z 3. Musimy więc przed dzieleniem dodać 1.

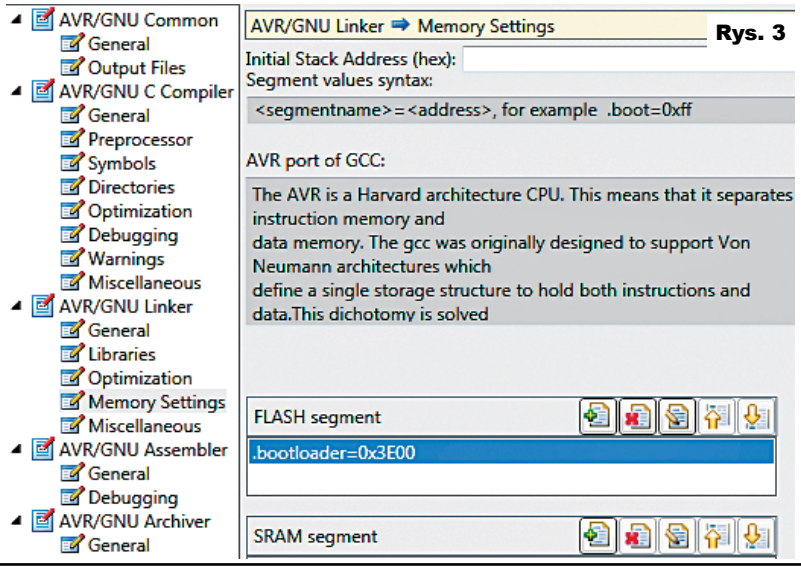
```
uint8_t testData[] = {1, 2, 3, 4, 5};
writeFlash(FLASH_PAGE_SIZE_BYTES * 100, (uint16_t*) testData, (sizeof(testData) + 1) / 2);
```

Analizując funkcję writeFlash(), niektórzy Czytelnicy mogą zapytać, jak właściwie zapewniane jest wywoływanie instrukcji SPM z obszaru bootloadera. Przenieśliśmy tam bowiem tę funkcję, a nie wywoływane przez nią boot_page_erase_safe i pozostałe obsługujące zapis do Flasha. Istotnie, nadawany funkcji atrybut dotyczy tylko jej, nie przenosi się na funkcje wywoływane. Jak więc to może działać? Otóż przypomnijmy sobie, że to, co wołamy we writeFlash(), to nie funkcje, ale makra zawierające wstawki asemblerowe. Funkcja writeFlash() nie zawiera więc tak naprawdę skoków do innych funkcji, ale zostają do niej wklejone przez preprocesor kawałki kodu w języku asemblera. Warto to obejrzeć, klikając wywołanie danego makra i wciskając Alt+G.

Uwaga! Ponieważ w sekcji bootloaderowej umieszczamy tylko funkcję pomocniczą writeFlash(), a nie program bootloadera, upewnijmy się, czy przypadkiem nie zaprogramowaliśmy (nie ustawiliśmy na 0) fuse bitu BOOTRST. Jeśli go zaznaczymy, po włączeniu mikrokontrolera zamiast funkcji main() zostanie uruchomiona funkcja writeFlash(). Pobierane przez nią parametry, w szczególności adres strony, będą losowe i funkcja skasuje przypadkową stronę z pamięci Flash, np. fragment głównego programu. BOOTRST należy zaprogramować jedynie w przypadku, gdybyśmy chcieli napisać bootloader. W tej lekcji jednak zajmujemy się tylko funkcją pomocniczą i BOOTRST powinien pozostać niezaprogramowany (**rysunek 4**).



Rys. 4



Rys. 3

Stałe w pamięci Flash

Dotychczas nie powiedzieliśmy jeszcze nic o odczycie danych z Flasha. Zanim jednak omówimy odczyt zapisanych programowo danych, zastanowimy się nad kwestią stałych zdefiniowanych w kodzie programu.

Ze stałych korzystamy, aby umieścić w programie dane, które będą niezmienne i dostępne przez cały czas działania mikrokontrolera. Najczęściej są to łańcuchy znaków oraz tablice różnych wartości liczbowych. Przykładem może być deklaracja wyglądu ikonki dzwonka z poprzedniej lekcji:

```
const uint8_t customCharBell[8] = {
```

```
0b00000,
0b00100,
0b01110,
0b01110,
0b01110,
0b11111,
0b00100,
0b00000  };
```

Czym właściwie różnią się stałe od zmiennych? Stałe w swoich deklaracjach zawierają słowo kluczowe const, które zabezpiecza przed modyfikacją ich wartości początkowej. Zabezpieczenie to działa tylko w momencie kompilacji. Kompilator zaprotestuje, jeśli zauważy, że próbujemy zmienić wartość stałej. Natomiast w trakcie działania programu można próbować ją zmodyfikować, np. przy użyciu wskaźnika. Na platformach bez ochrony pamięci może się to udać.

Tak więc stałe zajmują miejsce w pamięci operacyjnej tak samo jak zmienne. Dotychczas nie miało to dla nas znaczenia, ale kiedyś może zajść potrzeba umieszczenia w stałych dużej liczby danych, np. grafiki. Nasz mikrokontroler ma jednak tylko 2kB pamięci SRAM. Pojawia się więc pytanie, czy stałe nie mogłyby znajdować się w pamięci Flash. Przecież ich wartości i tak są we Flashu, dzięki czemu stałe mogą być zainicjowane po resecie mikrokontrolera, przez specjalny kod uruchamiany przed funkcją main().

Stałe mogłyby znajdować się w pamięci Flash, ale jest jeden problem: nasz mikrokontroler ATmega32 ma architekturę harwardzką, w której są oddzielne pamięci programu i danych, mające oddzielną przestrzeń adresową. Tymczasem język C stworzono z myślą o architekturze von Neumanna, w której program i dane znajdują się w jednej przestrzeni adresowej. W związku z tym, jeśli mamy wskaźnik pokazujący na jakiś adres w pamięci, to nie ma jak rozróżnić, czy jest to wskaźnik na pamięć programu (Flash), czy pamięć danych (SRAM).

Na szczęście zawarty w Atmel Studio kompilator GCC ma odpowiednie rozszerzenia, dzięki którym możemy odczytać dane z pamięci Flash. Po pierwsze musimy poinformować kompilator, które stałe mają znaleźć się we Flashu i nie mają być kopiowane do SRAM. Robimy to podobnie jak przy zmianie położenia funkcji, przez dodanie atrybutu __progmem__. Dla ułatwienia mamy do tego makro PROGMEM dostępne w bibliotece <avr/pgmspace.h>:

```
const uint8_t customCharBell[8] PROGMEM = {
 0b00000,
 0b00100,
 0b01110,
 0b01110,
 0b01110,
 0b11111,
 0b00100,
 0b00000  };
```

Uwaga! Tablicy ciągów znaków, a dokładniej tablicy wskaźników na ciągi, nie da się w ten sposób przenieść do pamięci Flash. Musimy stworzyć oddzielne zmienne tablicowe dla każdego ciągu, opatrzyć je makrem PROGMEM, a następnie stworzyć tablicę zawierającą wskaźniki na te tablice. Ona również musi być opatrzona makrem PROGMEM. Ponieważ tablica będzie zawierać wskaźniki na dane w pamięci Flash, musimy zadbać o określenie jej typu. Wykonujemy to makrem PGM_P. Jako przykład weźmy tablicę dni tygodnia z przykładów z lekcji 15 i 16:

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

Aby przenieść ją do pamięci Flash, musimy podzielić ją następująco (**listing 6**):

```
const char WeekDay1[] PROGMEM = "  ";
const char WeekDay2[] PROGMEM = "pn";
const char WeekDay3[] PROGMEM = "wt";
const char WeekDay4[] PROGMEM = "sr";
const char WeekDay5[] PROGMEM = "cz";
const char WeekDay6[] PROGMEM = "pt";
const char WeekDay7[] PROGMEM = "so";
const char WeekDay8[] PROGMEM = "nd";

PGM_P const WeekDays[] PROGMEM = {
 WeekDay1,
 WeekDay2,
 WeekDay3,
 WeekDay4,
 WeekDay5,
 WeekDay6,
 WeekDay7,
 WeekDay8 ,  }  ;
```

Ciągi znaków umieszczamy nieraz bezpośrednio jako argumenty wywołania funkcji, np. printf(). Czy one też mogą być umieszczone w pamięci Flash? Tak, służy do tego makro PSTR:

```
lcdString_P(PSTR("EdW"));
```

Odczyt danych z pamięci Flash

Gdy stałą mamy już we Flashu, możemy uzyskać do niej dostęp za pomocą odpowiednich makr:

- pgm_read_byte(address) – odczytuje daną jednobajtową
- pgm_read_word(address) – odczytuje daną dwubajtową
- pgm_read_dword(address) – odczytuje daną czterobajtową
- pgm_read_float(address) – odczytuje daną typu float
- pgm_read_ptr(address) – odczytuje wskaźnik

Jako parametr address podajemy adres stałej lub jej elementu w przypadku tablic:

```
uint8_t byte=pgm_read_byte(&(customCharBell[2]));
```

W ten właśnie sposób możemy też odczytywać z Flasha dane, które zapisaliśmy w określonej stronie pamięci:

```
byte = pgm_read_byte(FLASH_PAGE_SIZE_BYTES * 100);
```

Jak widać, choć możemy odczytywać pamięć Flash, to jednak ograniczenia wynikające z architektury harwardzkiej nie pozwalają nam na łatwy dostęp do tejże pamięci. Nie możemy np. przekazać do funkcji printf() wskaźnika na napis umieszczony we Flashu lub w inny sposób „przejechać" wprost po tablicy znajdującej się we Flashu. Dane możemy wczytywać maksymalnie po cztery bajty i ewentualnie buforować je w RAM-ie. To samo zresztą dotyczy EEPROM-u. W związku z tym jeśli chcemy korzystać z funkcji operujących na tablicach danych, musimy napisać ich wersje dostosowane do danych z pamięci Flash. Załóżmy, że chcemy wyświetlić na LCD ciąg znaków zapisany we Flashu. Dostosujmy w tym celu istniejącą funkcję lcdString() tworząc jej nowy wariant o nazwie lcdString_P():

```
void lcdString_P(const char * str) {
 uint8_t c = 0;
 while ((c = pgm_read_byte(str++)) != 0) {
  lcdWriteData(c);  } }
```

W funkcji tej odczytujemy bajty (znaki) z pamięci Flash, począwszy od adresu przekazanego w parametrze str. Parametr ten za każdym bajtem zwiększamy o 1, aby przejść do kolejnego bajtu. Dane odczytujemy tak długo, aż natrafimy na bajt o wartości 0, oznaczający koniec ciągu. Analogicznie możemy zdefiniować funkcję definiującą znak dla wyświetlacza, pobierającą dane z Flasha:

```
void lcdDefineChar_P(const uint8_t * charDefinition, uint8_t code) {
 lcdWriteCommand(LCD_COMMAND_SET_CGRAM_ADDRESS + (code % 8) * 8);
 for (uint8_t i = 0; i < 8; i++) {
  lcdWriteData(pgm_read_byte(charDefinition + i));
 }
}
```

Funkcja ta wygląda niemal identycznie jak oryginalna. Dodane zostało tylko wywołanie funkcji pgm_read_byte().

Podsumowaniem naszej wiedzy o obsłudze pamięci Flash będzie krótki program testowy. Został on przedstawiony na **listingu 7**.

Po inicjalizacji LCD deklarujemy tablicę z przykładowymi danymi testowymi. Zapisujemy je w wybranej stronie pamięci Flash a następnie odczytujemy i wyświetlamy na LCD. Tablica testowa zajmuje 5 bajtów, wymaga więc zapisania trzech dwubajtowych słów. Stąd do rozmiaru tablicy dodawane jest jeden, a wynik dzielony jest na 2.

Kolejny test dotyczy stałej przechowywanej we Flashu. Stałą tą jest tablica przechowująca definicję wyglądu symbolu dla LCD. Adres tablicy przekazujemy do zmodyfikowanej funkcji odpowiedzialnej za przesyłanie definicji symbolu. Definicja zostaje umieszczona na początku pamięci CG RAM, pod adresem 0. Aby wyświetlić symbol, stosujemy standardowo funkcję lcdWriteData(). Przed nią musimy wywołać lcdGotoXY(), ponieważ po zakończeniu definiowania znaku wyświetlacz pozostaje w trybie przyjmowania danych

```
int main(void) {
 lcdInit();
 lcdInitPrintf();
 //test zapisu danych w stronie nr 100
 uint8_t testData[] = {1, 2, 3, 4, 5};
 writeFlash(FLASH_PAGE_SIZE_BYTES * 100, (uint16_t*) testData, (sizeof(testData) + 1) / 2);
 for (uint8_t i = 0; i < sizeof(testData); i++) {
  uint8_t byte = pgm_read_byte(FLASH_PAGE_SIZE_BYTES * 100 + i);
  printf("%d ", byte);    }
 //test odczytu definicji znaku zapisanej w stałej
 lcdDefineChar_P(customCharBell, 0);
 lcdGotoXY(10, 0);
 lcdWriteData(0);
 //test odczytu ciągu znaków z tablicy ciągów
 lcdGotoXY(0, 1);
 lcdString_P(pgm_read_ptr(&(WeekDays[2])));
 //test odczytu literału
 lcdString(" ");
 lcdString_P(PSTR("EdW"));
 while(1){ }   }
```

Listing 7

do pamięci CG RAM i trzeba go przestawić na DD RAM.

W teście trzecim pobieramy napis z tablicy wskaźników na ciągi znaków. Z tablicy WeekDays trzeba więc najpierw pobrać adres odpowiedniego ciągu i dopiero wtedy przekazać go do funkcji wyświetlającej ciąg. Tutaj też musimy skorzystać ze zmodyfikowanej funkcji, która będzie pobierała dane z Flasha zamiast z RAM-u.

Ostatni test również wykorzystuje funkcję lcdString_P(), jednak tym razem przekazujemy do niej makro PSTR, dzięki któremu możemy przekazać bezpośredni literał zamiast uciekać się do stałej o konkretnej nazwie.

Zadania

Wbudowane w mikrokontroler pamięci nieulotne Flash oraz EEPROM dają dużo możliwości. Pozwalają nie tylko przechowywać konfigurację czy dane pomocnicze, ale też zbierać dane w trakcie działania programu. Warto więc z nimi poeksperymentować. W tym odcinku proponuję następujące ćwiczenia:

1. Termometr z okresowym zapisem temperatury w EEPROM-ie i przeglądaniem historii pomiarów (czas z RTC)
1. Zapis w pamięci Flash momentów włączenia i wyłączenia zasilania, wykorzystać funkcje RTC

W materiałach dodatkowych umieszczone są rozwiązania zadań z poprzedniej lekcji oraz projekty testujące obsługę wewnętrznych pamięci EEPROM i Flash.

Grzegorz Niemirowski
grzegorz@grzegorz.net