Mikroprocesory i technika mikroprocesorowa

skrypt do laboratorium

Dariusz Chaberski

2 października 2012

Spis treści

1 Budowa mikrokontrolera i programowanie w języku Assemblera

1.1 Środowisko Atmel AVR Studio oraz zestaw uruchomieniowy (2 godziny)

Cel ćwiczenia

Celem ćwiczenia jest zapoznanie ze środowiskiem Atmel AVR Studio [4] na poziomie umożliwiającym późniejsze swobodne korzystanie w większości ćwiczeń laboratoryjnych oraz z zestawem uruchomieniowym ZL10 [5]. W trakcie ćwiczenia przedstawione zostaną podstawowe określenia i mechanizmy stosowane podczas pisania, assemblowania, testowania i uruchamiania napisanych programów, programowania mikrokontrolera oraz obsługi zestawu uruchomieniowego.

Zagadnienia do przygotowania

Wymagana jest znajomość obsługi komputera, bardzo podstawowa wiedza dotycząca architektury mikroprocesorów oraz znajomość reprezentacji szesnastkowej oraz dwójkowej liczb. Zagadnienie te nie powinny być niczym nowym dla studentów uczestniczących w zajęciach, więc nie wymagają dodatkowego przygotowania i mogę być prowadzone od pierwszych zajęć.

Wymagana jest również znajomość podstaw elektroniki oraz elektrotechniki. Wykonujący ćwiczenie powinien zdawać sobie sprawę z obecności ładunku elektrostatycznego, który w połączeniu z małymi pojemnościami może powodować napięcia zdolne zniszczyć układy wykonane w technologii MOS. Aby do tego nie dopuścić należy przed dotknięciem zestawu najpierw wyrównać swój ładunek dotykając masy zestawu (złącze sprzęgu szeregowego DB9), a dopiero później elementu (zworki, potencjometru). Należy również być świadomym faktu, że zasilacz i komputer mogą mieć masy o różnych potencjałach, dlatego najpierw należy podłączać złącza do programowania (USB, SPI), a dopiero później wtyk zasilacza. Kolejność odłączania powinna być oczywiście odwrotna.

Przebieg ćwiczenia

1. Tworzenie nowego projektu

   (a) Utworzyć w katalogu d:\users katalog użytkownika. Nazwa katalogu powinna jasno identyfikować użytkownika, nie może jednak zawierać polskich znaków diaktrytycznych oraz znaków specjalnych, w tym spacji, ponieważ wykorzystywane w trakcie zajęć narzędzia nie działają w takich sytuacjach poprawnie. Utworzony katalog może mieć na przykład nazwę “ImieNazwisko” lub “INazwisko”, ale nie może mieć nazwy “Imię Nazwisko” lub “ImięNazwisko”.

   (b) Uruchomić Atmel AVR Studio.

   (c) Wcisnąć przycisk “New Project”.

   (d) Wybrać typ projekt “Atmel AVR Assembler”.

   (e) Wybrać nazwę projektu (nazwa projektu ani żadnego pliku projektu nie może zawierać polskich znaków diaktrytycznych oraz znaków specjalnych, w tym spacji). Najlepiej na nazwę pierwszego projektu wybrać po prostu “pierwszy”.

   (f) Wybrać katalog projektu jako wcześniej utworzony katalog użytkownika.

   (g) Zaznaczyć opcję “Create Initial file”, która umożliwia dodanie do projektu pliku głównego o nazwie innej niż projekt, na przykład “main.asm” oraz opcję “Create folder”, która powoduje, że cały projekt będzie przechowywany w osobnym katalogu. Nazwa tego katalogu będzie taka sama jak nazwa projektu. Następnie nacisnąć przycisk “Next >>>”.

(h) Wybrać platformę do debugowania jako “AVR Simulator”, co umożliwi sprawdzanie działania pisanego kodu na programowym modelu mikrokontrolera, następnie wybrać układ “Device” jako “ATmega16” i nacisnąć przycisk “Finish”.

(i) Po wykonaniu powyższych czynności AVR Studio jest gotowe do wprowadzania kodu źródłowego.

2. Wpisywanie kodu źródłowego, assemblacja i przeglądanie zawartości pamięci

(a) Wprowadzić poniższy kod do głównego pliku źródłowego projektu. Kody z całego bieżącego ćwiczenia można wprowadzić do pliku źródłowego z wykorzystanie opcji (kopiuj, wklej) schowka. W przypadku pozostałych ćwiczeń, w których kody źródłowe nie przekraczają połowy strony należy je przepisywać ręcznie w celu lepszego opanowania materiału.

```
.include "m16def.inc"

ldi r16, low(ramend)
out spl, r16
ldi r16, high(ramend)
out sph, r16

call podprogram

ldi r16, 0xab
mov r0, r16
inc r0

koniec:
rjmp koniec

podprogram:
        nop
        ret

.dseg
dmem: .byte 7

.cseg
cmem: .db "ABCD", 0

.eseg
emem: .db "1234", 0
```

Powyższy kod programu na obecnym etapie nie musi być rozumiany w całości. Celem tego ćwiczenia jest poznanie narzędzia programistycznego AVR Studio i w tym celu wystarczy tylko pobieżna orientacja w działaniu tego kodu.

(b) Dokonać assemblacji wprowadzonego kodu poprzez wybranie w menu “Build” polecenia “Build”, wciśnięcia przycisku “F7” lub naciśnięcia ikony “Assemble (F7)” w pasku zadań “Assemble”.

(c) W wyniku assemblacji zostaną utworzone pliki w formacie HEX opisujące zawartości pamięci programu FLASH (rozszerzenie hex) oraz pamięci danych nieulotnych EEPROM (rozszerzenie eep). Sprawdzić w katalogu projektu ich obecność oraz przejrzeć ich zawartość w notatniku.

(d) Zwrócić uwagę na raport Assemblera w okienku "Build" zakładki "Build". W segmencie programu, czyli w pamięci FLASH (.cseg) zostało użytych 30 bajtów, z czego 24 bajtów przeznaczonych zostało na kod programu, a 6 bajtów na napis "ABCD", 0 (string zero). Pomimo, że napis, łącznie ze znacznikiem końca (0) posiada wielkość 5 bajtów, to z uwagi na charakter adresowania pamięci programu, co zostanie wyjaśnione w jednym z kolejnych ćwiczeń, przydzielone zostaje mu 6 bajtów i generowane jest ostrzeżenie o wstawieniu dodatkowego bajtu o wartości 0. Z raportu również można wywnioskować, że powyższy kod wykorzystuje 7 bajtów w pamięci danych - segment danych (.dseg) oraz 5 bajtów w pamięci danych nieulotnych - segment EEPROM (.eseg).

(e) Uruchomić debugowanie poprzez wybranie w menu "Debug" polecenia "Start Debugging" lub wciśnięcie kombinacji przycisków Ctrl+Shift+Alt+F5.

(f) Wybrać w menu "View" polecenie "Memory" i przesunąć nowo utworzone okienko w dogodne miejsce. Należy zauważyć, że podczas przeciągania na obrzeżach siatki pojawiają się strzałki sugerujące położenie okna po zwolnieniu przycisku mysz. W nowo powstałym okienku "Memory" wybrać zasób "Program" i przejrzeć zawartość pamięci. Wyszukać napisu "ABCD", zwrócić uwagę, że tuż za napisem znajdują się dwa bajty o wartości 0x00, a tuż za nimi pamięć niezaprogramowana - bajty o wartościach 0xFF. Nie wnikając w szczegóły, które na tym etapie są nieistotne, łatwo jest zlokalizować miejsce w pliku źródłowym, które odpowiedzialne jest za napis "ABCD". Zmienić napis "ABCD" na jakiś inny, dokonać ponownej assemblacji, uruchomić debugowanie i sprawdzić, czy zawartość pamięci programu została zmieniona zgodnie z oczekiwaniami. W przypadku, zmiany długości napisu należy zwrócić uwagę na informację w raporcie dotyczącą liczby bajtów przeznaczonych na dane w pamięci programu.

(g) W oknie "Memory" wybrać zasób "EEPROM". Jak widać pamięć nie jest zaprogramowana - zawiera same bajty o wartościach 0xFF. W pamięci tej powinien znajdować się jednak napis "1234", 0. Zawartość pamięci EEPROM nie jest uzupełniana automatycznie w momencie uruchamiania debugowania tak, jak ma to miejsce w przypadku pamięci programu. Aby zawartość pamięci EEPROM w programie debugującym była zgodna z zawartością pliku źródłowego należy uzupełnić ją poprzez odczyt pliku z rozszerzeniem eep znajdującym się w katalogu projektu. W tym celu należy wybrać z menu "Debug" polecenie "Up/Download Memory". W nowo powstałym okienku wybrać typ pamięci jak "EEPROM" oraz wskazać właściwy plik *.eep, a następnie wcisnąć przycisk "Load from File", zignorować ostrzeżenie o różnicy pomiędzy pojemnością pamięci EEPROM a ilością danych w pliku i zamknąć okno. Odszukać napis w pliku źródłowym, dokonać jego modyfikacji, ponownie dokonać assemblacji, uruchomienia symulacji i odczytu zawartości pamięci EEPROM z pliku.

(h) Wybrać w oknie "Memory" zasób "Data". Zawartości pamięci danych nie można określać bezpośrednio z pliku. Jej zawartość może być określana poprzez działający program, co zostanie zademonstrowane w kolejnym punkcie ćwiczenia.

3. Śledzenie działania programu na modelu

(a) Podejrzeć menu "Debug". Zwrócić uwagę na polecenie "Step Into", które dostępne jest pod skrótem klawiszowym F11, "Step over" - F10, "Run to Cursor" - Ctr+F10 oraz "Reset" - Shift+F5. Zamknąć menu, zwrócić uwagę, na linię na którą wskazuje żółta strzałka z lewej strony pola edycji

pliku źródłowego. Strzałka ta wskazuje instrukcję, która ma zostać wykonana. Rozwinąć gałąź "Registers" w okienku "Procerssor", wykonać pojedynczą instrukcję poprzez wciśnięcie przycisku F11. Zwrócić uwagę na przesunięcie się żółtej strzałki na kolejną instrukcję oraz czerwone zabarwienie zmodyfikowanych zasobów - w tym przypadku rejestry "R16" oraz "Program Counter". W ten sposób wykonać cały program do momentu uwięzienia w pętli koniec. Zwrócić uwagę na to, że modyfikowane zasoby są zaznaczane na czerwono oraz że wykonywanie podprogramu (podprogram:) jest wizualizowane. Można również zwrócić uwagę, na modyfikację ostatnich komórek pamięci danych tuż po wywołaniu podprogramu.

(b) Wykonać reset modelowanego mikrokontrolera, poprzez wciśnięcie kombinacji przycisków Shift+F5 i wykonać cały program ponownie, ale z użyciem przycisku skrótu F10, co spowoduje, że wizualizacja wykonywanego programu nie będzie obejmowała podprogramu. Często zdarza się tak, że w testowanym programie wykorzystywany jest dobrze działający podprogram, którego śledzenie nie ma sensu, gdyż z całą pewnością działa on poprawnie.

(c) Zresetować mikrokontroler (Shift+F5), ustawić kursor w linii 11 i wcisnąć Ctr+F10. Spowodowało to przekazanie sterowania do linii, w której znajdował się kursor. Opcja ta jest przydatna w przypadku, kiedy chcemy rozpocząć śledzeni programu od wybranej instrukcji a nie od początku.

4. Programowanie mikrokontrolera

(a) Rozpocząć nowy projekt zgodnie z wcześniejszymi wskazówkami, uzupełnić zawartość głównego pliku źródłowego projektu kodem zawartym poniżej, a następnie dokonać assemblacji.

```
.include "m16def.inc"

ldi r16, 0xff
out ddrb, r16

loop:
        ldi r16, 255
        delay1:
                ldi r17, 255
                delay0:
                        dec r17
                brne delay0
                dec r16
        brne delay1
        inc r0
        out portb, r0
rjmp loop
```

(b) Sprawdzić połączenie pomiędzy programatorem a komputerem (interfejs USB) oraz pomiędzy programatorem a zestawem uruchomieniowym (przewód 6-ścio żyłowy, złącze 10 igłowe IDC, interfejs SPI, złącze JP16 - ISP). Sprawdzić zasilanie, zasilacz powinien być umieszczony w listwie zasilającej z ustawionym napięciem 9 V, wtyczka (wtórna) zasilacza powinna być umieszczona w gnieździe Gn1 - PWR.

(c) Włączyć zasilanie w listwie zasilającej, obie diody w programatorze powinny zaświecić się na zielono. Dioda znajdująca się bliżej złącza USB sygnalizuje stan połączenie programatora z kompute-

rem, natomiast druga dioda sygnalizuje połączenie programatora z zestawem uruchomieniowym. Należy doprowadzić do sytuacji, w której obie diody świecą na zielono. Podczas odłączania i podłączania przewodów (zasilacz, USB, SPI) należy pamiętać o właściwej kolejności podłączania wspomnianej w Zagadnieniach do przygotowania, w przeciwnym przypadku zestaw może ulec uszkodzeniu.

(d) Z podmenu "Program AVR" menu "Tools" wybrać polecenie "Connect...". W oknie wyboru programatora ustawić platformę jako "AVRISP mkII" oraz port jako "USB", a następnie dokonać połączenia wciskając przycisk "Connect".

(e) W zakładce "Program" w sekcji "Flash" wskazać wciskając przycisk "..." plik wynikowy z rozszerzeniem hex, a następnie nacisnąć przycisk "Program" celem zaprogramowani pamięci FLASH mikrokontrolera.

(f) Skonfigurować zestaw laboratoryjny w taki sposób, aby stan portu PB był przekazywany na linijkę diod LED (LEDs - lewy dolny róg zestawu). W tym celu należy zworkę JP17 umieścić w położeniu PB.

(g) Obserwować stan diod LEDs. Zwrócić uwagę, że dioda reprezentująca stan bitu o najniższej wadze znajduje się z lewej strony, a dioda reprezentująca stan bitu o najwyższej wadze znajduje się z prawej strony zestawu. Na diodach powinny pojawiać się teraz reprezentacje kolejnych stanów w kodzie naturalnym binarnym co czas równy około 24 ms. Dioda o najniższej wadze powinna migać z częstotliwością około 20 Hz, a dioda o najwyższej wadze powinna migać z częstotliwością około 160 mHz (okres około 6 s).

(h) Zmienić wartość stałej (255) wpisywanej do rejestru w linijce 7 lub 9 na inną, np. 128 (wartości tych stałych muszą mieścić się w zakresie od 0 do 255). Dokonać ponownej assemblacji i programowania pamięci FLASH. Zaobserwować różnicę w częstotliwości zmiany stanu diod.

(i) Zmienić port, na który wysyłane są kolejne stany. W tym celu należy w linijce 4 zmienić literał "ddrb" na "ddrc", w linijce 16 literał "portb" należy zmienić na "portc" oraz skonfigurować zestaw tak, aby stan portu PC był przekazywany na linijkę diod LED. (JP17 umieścić w położeniu PC). Dokonać assemblecji i programowania. Sprawdzić poprawność działania.

5. Utrwalanie obsługi AVR Studio

(a) W celu utrwalenie posługiwania się narzędziem utworzyć od początku nowy projekt zgodnie z wcześniejszymi wskazówkami. Jako zawartość głównego pliku źródłowego projektu wybrać program umieszczony w punkcie 4 ćwiczenia.

(b) Sprawdzić jego działanie na modelu. Dokonać assemblacji, uruchomić debugowanie, rozwinąć gałąź "PORTA" w okienku "I/O View". Podczas symulacji obserwować zmiany stanów poszczególnych bitów rejestru wejścia wyjścia PORTA modułu PORTA.

(c) Ustawić kursor na linii 17, wybrać polecenie "Run to Cursor" zanotować stan licznika cykli "Cycle Counter" znajdujący się w okienku "Processor", ponownie wybrać polecenie "Run to Cursor" i ponownie zanotować stan licznika cykli. Z różnicy stanów licznika cykli oraz częstotliwości pracy mikrokontrolera (8 MHz) wyznaczyć częstotliwość wykonywania instrukcji z linii 17. Porównać tą częstotliwość z częstotliwością zmiany stanu diod.

(d) Zmienić znacząco wartości stałych wpisywanych do rejestrów w liniach 7 i 8. Ponownie sprawdzić na modelu czas wykonywania instrukcji z linii 17, następnie zaprogramować i sprawdzić zgodność symulacji na modelu z działaniem rzeczywistego układu.

6. Materiały źródłowe

   Z lokalizacji www.fizyka.umk.pl/~daras/mtm/cwiczenia ściągnąć pliki w formacie pdf dokumentacji cytowanych w instrukcji bieżącego ćwiczenia i zapisać je w podkatalogu "dokumentacja", który należy utworzyć w katalogu użytkownika. Krótko zapoznać się z ich zawartością. Wskazana lokalizacja zawiera pliki, które będą stanowiły podstawę źródła wiedzy dla zagadnień do przygotowania w kolejnych ćwiczeniach.

1.2 Kody maszynowe instrukcji (1 godzina)

Cel ćwiczenia

Celem ćwiczenia jest zdobycie doświadczenia i pogłębienie wiedzy na temat kodowania instrukcji mikrokontrolera lub mikroprocesora i przechowywania reprezentacji bitowych tych instrukcji w pamięci programu. Ćwiczenie pozwala na dogłębne zapoznanie z działaniem i budową (kod maszynowy) kilku wybranych instrukcji mikrokontrolera ATmega16.

Zagadnienia do przygotowania

Przed przystąpieniem do ćwiczenia należy zaznajomić się z zagadnieniem kodowania instrukcji Assemblera w pamięci programu. Należy zdawać sobie sprawę z ograniczeń argumentów instrukcji Assemblera oraz zależności rodzaju argumentu od mnemonika instrukcji. Wymagana jest znajomość kodowania liczb w kodzie binarnym naturalnym (KBN) oraz kodzie binarnym z uzupełnieniem do 2 (U2). Korzystając z dokumentacji [1] należy zapoznać się z działaniem, kodowaniem i zakresem argumentów instrukcji o mnemonikach LDI, MOV, JMP, RJMP. Nie należy oczywiście uczyć się kodów maszynowych tych instrukcji na pamięć, ale należy umieć sprawnie określać wartości kodów maszynowych tych instrukcji w zależności od argumentów korzystając z dokumentacji.

Przebieg ćwiczenia

1. Określanie kodów maszynowych instrukcji

   (a) Dokonać assemblacji poniższego kodu i uruchomić śledzenie działania programu (na modelu).

   ```
   LDI R16, 0xAB
   LDI R18, 0xCD

   KONIEC:
   RJMP KONIEC
   ```

   (b) Wyświetlić zawartość pamięci programu, odszukać kod maszynowy pierwszej instrukcji. Odnaleźć bity w kodzie maszynowym pierwszej instrukcji, których zadaniem jest kodowanie stałej (0xAB). Sprawdzić poprawność kodowania stałej korzystając z dokumentacji. W ten sam sposób sprawdzić poprawność kodowanie stałej dla drugiej instrukcji. Zwrócić uwagę na fakt, że kody maszynowe instrukcji są umieszczone w pamięci programu z wykorzystaniem kolejności bajtowej little endian.

   (c) Porównać odczytaną z pamięci programu wartość rdzenia instrukcji (bity od 12 do 15) z wartością odczytaną z dokumentacji. Zauważyć, że wskazane bity są jedynymi wyróżniającymi instrukcję LDI wśród innych instrukcji.

(d) Z wykorzystaniem dokumentacji określić bity, których zadaniem jest kodowanie rejestru docelowego. Porównać kody maszynowe dwóch pierwszych instrukcji i wyjaśnić sposób kodowania rejestru docelowego.

(e) Wyjaśnić reprezentację bitową kodu maszynowego trzeciej instrukcji. Sprawdzić poprawność reprezentacji bitowej rdzenia instrukcji, a następnie dokładnie przyjrzeć się reprezentacji wartości kodującej miejsce do którego ma zostać przekazane sterowanie. Należy mieć na uwadze to, że RJMP jest instrukcją skoku względnego, miejsce do którego ma zostać przekazane sterowanie jest kodowane bezpośrednio w kodzie maszynowym z wykorzystaniem kodowania U2 oraz, że odniesieniem dla instrukcji RJMP jest adres kolejnej instrukcji.

(f) Pomiędzy etykietą KONIEC: a instrukcją RJMP KONIEC umieścić pustą instrukcję NOP. Porównać wartość kodującą w kodzie maszynowym instrukcji RJMP miejsce, do którego ma zostać przekazane sterowanie z analogiczną wartością z punktu (e).

(g) Korzystając z raportu w okienku "Build" zakładki "Build" określić wielkość wykorzystanej pamięci programu. Zmienić instrukcję RJMP na instrukcję JMP, która jest instrukcją skoku bezwarunkowego bezwzględnego. Po dokonaniu assemblacji określić wielkość wykorzystanej pamięci programu, porównać tą wielkość z wielkością uzyskaną na początku tego podpunktu. Wyjaśnić różnicę. Korzystając z dokumentacji sprawdzić poprawność kodu maszynowego wygenerowanego dla tej instrukcji. Zwrócić uwagę na zakres wartości kodowanego adresu oraz liczbę bitów, które zostały przeznaczone w kodzie maszynowym instrukcji JMP na zakodowanie tego adresu.

(h) Usunąć dodaną wcześniej instrukcję NOP i wyjaśnić dlaczego kod maszynowy instrukcji JMP nie uległ zmianie podczas, gdy kod maszynowy instrukcji RJMP był zależny od tego, czy pomiędzy etykietą KONIEC: a instrukcją RJMP KONIEC znajdowała się lub też nie znajdowała się instrukcja NOP.

2. Śledzenie działania instrukcji

(a) Wykonać pierwszą instrukcję i sprawdzić zawartość rejestru docelowego, następnie postąpić tak samo z drugą instrukcją.

(b) Spowodować zapętlenie wykonywanego programu.

(c) Rozbudować kod źródłowy o kolejne instrukcje tak, aby przyjąć on postać poniższego listingu.

```
LDI R16, 0xAB
LDI R18, 0xCD

MOV R0, R16
MOV R1, R18

RJMP PC
```

Argument PC w instrukcji RJMP jest synonimem rejestru licznika programu (ang. Program Counter) i jest on w trakcie assemblacji zamieniany na adres w pamięci programu bieżącej instrukcji. Konstrukcje z wykorzystaniem literału PC nie wymagają wprowadzania dodatkowych etykiet przez co są wygodniejsze w użyciu.

(d) Prześledzić działanie instrukcji o mnemoniku MOV. Zwrócić uwagę na modyfikowane zasoby (rejestry R0 i R1).

(e) Sprawdzić poprawność kodów maszynowych wygenerowanych dla instrukcji o mnemonikach MOV. Zwrócić uwagę na poprawność kodowania argumentów (adresowanie bezpośrednie rejestrowe) tych instrukcji.

3. Prosty projekt

(a) Napisać kod realizujący wpisanie wartości 0x1234 do pary rejestrów R1:R0, przyjąć, że bajt bardziej znaczący (0x12) zostanie wpisany do rejestru o wyższym indeksie (R1), a bajt mniej znaczący zostanie wpisany do rejestru R0. Projekt należy zrealizować z wykorzystaniem przedstawionych wcześniej instrukcji.

(b) Wyszukać w dokumentacji opis instrukcji MOVW i następnie zrealizować powyższy projekt z jej wykorzystaniem.

1.3 Zasoby pamięciowe mikrokontrolera ATmega16 (3 godziny)

Cel ćwiczenia

Celem ćwiczenia jest zdobycie wiedzy dotyczącej zasobów pamięciowych mikrokontrolera Atmel ATmega16, dogłębne zrozumienie podstawowych trybów adresowania oraz praktyczne poznanie działania instrukcji mikrokontrolera umożliwiających dostęp do pamięci SRAM oraz FLASH. W trakcie ćwiczenia student udoskonali umiejętność posługiwania się dokumentacją mikrokontrolera oraz zdobędzie umiejętność wyboru właściwego sposobu adresowania połączoną z umiejętnością wyboru właściwej instrukcji częściowo z wykorzystaniem dokumentacji. Dodatkowym celem ćwiczenia jest zapoznanie z mechanizmem etykiet oraz z podstawowymi dyrektywami Assemblera jak również dogłębne zrozumienie metody pozyskiwania adresu dla danych w pamięci programu.

Zagadnienia do przygotowania

Przed przystąpieniem do ćwiczenia należy zapoznać się z adresowaniem bezpośrednim, adresowaniem pośrednim i instrukcjami ładującymi wartości stałe do rejestrów na poziomie rozumienia argumentów instrukcji Assemblera oraz zależności rodzaju argumentu od mnemonika instrukcji. Oczywistym powinno być, że w instrukcji adresowania pośredniego jeden z argumentów reprezentuje nazwę rejestru, który z kolei przechowuje adres komórki pamięci, która stanowi albo element docelowy, albo przechowuje wartość źródłową, a drugi z argumentów reprezentuje nazwę rejestru, który przechowuje albo wartość źródłową albo stanowi rejestr docelowy. Należy również rozumieć, że liczba bitów w kodzie maszynowym przeznaczona do kodowania rejestru indeksowego jest uzależniona od liczby rejestrów indeksowych oraz liczby wariantów ich wykorzystania, a nie od liczby bitów, które te rejestry posiadają.

W przypadku instrukcji ładujących wartość stałą do rejestru w sposób natychmiastowy należy rozumieć fakt, że stała ta znajduje się (jest kodowana) w kodzie maszynowym instrukcji, a nie na przykład w pamięci danych lub rejestrze. Należy również zdawać sobie sprawę z ograniczenia jakie daje skończona liczba bitów, na których ta stała jest kodowana, oraz sposób jej kodowania na jej zakres. Wymagana jest znajomość naturalnego kodu binarnego (NKB) oraz kodu uzupełnień do dwóch (U2).

Odnośnie adresowania bezpośredniego należy zdawać sobie sprawę, że adres wykorzystywanej komórki pamięci kodowany jest w kodzie maszynowym bezpośrednio - na tej samej zasadzie co nazwa(indeks) rejestru ogólnego przeznaczenia oraz, że liczba bitów przeznaczonych do zakodowania tego adresu jest zależna od wielkości pamięci, do której adresowanie to umożliwia dostęp. Takie wady adresowania bezpośredniego jak rozbudowany(obszerny) kod maszynowy, brak możliwość automatycznej modyfikacji adresu oraz zaleta, którą jest szybkość uzyskiwania adresu powinny być rozumiane u podstaw.

Należy być zorientowanym w typowych zastosowaniach pamięci RAM, FLASH oraz dla pełniejszego obrazu zasobów mikrokontrolera również i pamięci EEPROM, która w tym ćwiczeniu nie będzie wykorzystywana, a poświęcone jest jej w całości inne ćwiczenie.

Prowadzący jest zobowiązany do weryfikacji wiedzy z poprzez system 5-minutowych sprawdzianów, jak również do weryfikacji wiedzy i umiejętności nabytych w trakcje ćwiczenia

Przebieg ćwiczenia

1. Ładowanie 16-to bitowej stałej do par rejestrów

   (a) Prosty odczyt pamięci programu

      i. Dokonać assemblacji poniższego kodu.

```
LDI ZL, low(cmem)
LDI ZH, high(cmem)

LPM R0, Z
LPM R1, Z+
LPM R2, Z

RJMP PC

.cseg
cmem: .db "ABCD", 0
```

      ii. Uruchomić opcję śledzenia programu na modelu. Wyświetlić zawartość pamięci programu, odszukać napis (string 0) “ABCD”, 0 i określić jego adres (początek). Ów adres będzie użyteczny dla instrukcji o mnemoniku LPM, która pozwala odczytywać z pamięci programu dane w porcjach o wielkości pojedynczego bajtu. Dokładnie instrukcja LPM R0, Z odczytuje z pamięci programu bajt o adresie znajdującym się w rejestrze Z i wpisuje go do rejestru R0 [2]. Jest to instrukcja, która ze względu na drugi argument stanowi przykład adresowania pośredniego, a ze względu na pierwszy argument stanowi przykład adresowania bezpośredniego rejestrowego. Często jednak instrukcję tą klasyfikuje się wyłącznie jako instrukcję adresowania pośredniego, nie wspominając o sposobie adresowania pierwszego argumentu. Dodatkowo z uwagi na fakt, że zasobem przechowującym adres jest rejestr indeksowy (wyróżniony rejestr, dla którego dodano możliwość automatycznej modyfikacji) mówi się o tej instrukcji jako o instrukcji z adresowaniem indeksowym.

      iii. Dyrektywa .db służy do definiowania zawartości pamięci. Może być ona używana w pamięci programu (FLASH) oraz pamięci danych nieulotnych (EEPROM). Dyrektywa .cseg jest przełącznikiem segmentu pamięci [1]. Powoduje przełączenie na pamięć programu, co oznacza, że znajdujące się za nią definicje i rezerwacje będą umieszczane przez Assembler w pamięci programu. W tym przypadku użycie dyrektywy .cseg jest zbyteczne, ponieważ domyślnie od początku pliku, jego zawartość przeznaczona jest dla pamięci programu. W sytuacji jednak gdyby wcześniej użyto przełącznika .eseg lub .dseg zastosowanie .cseg byłoby konieczne [1].

      iv. Wykonać dwie pierwsze instrukcje, których zadaniem jest wpisanie adresu reprezentowanego przez etykietę CMEM: do 16-to bitowego rejestru indeksowego Z. Rejestr indeksowy Z jest złożeniem dwóch ostatnich rejestrów ogólnego przeznaczenia R31:R30, podobnie jak rejestr

indeksowy X jest złożeniem rejestrów R27:R26, a rejestr indeksowy Y jest złożeniem rejestrów R29:R28 [3]. Dla zwiększania przejrzystości kodu oraz zwiększenia wygody programowania młodsza część rejestru Z otrzymała synonim ZL, a starsza część rejestru Z otrzymała synonim ZH. Podobnie zostały utworzone synonimy dla rejestrów X (XH:XL) oraz Y (YH:YL). Funkcje high(x) oraz low(x) zwracają odpowiednio część starszą oraz część młodszą argumentu x [1]. Powinno okazać się, że wartość w rejestrze Z jest dwukrotnie mniejsza aniżeli adres, pod którym został umieszczony kod ASCII literki A. Jest to spowodowane tym, że wartości etykiet umieszczonych w pamięci programu są generowane w celu pozyskiwania adresu instrukcji, a kody maszynowe instrukcji są albo 16-to, albo 32-dwu bitowe. Adresy etykiet umieszczonych w pamięci programu są zwiększane nie co jeden bajt ale co dwa bajty, ponieważ jest to bardziej optymalne, gdyż z wykorzystaniem tej samej ilości bitów w rejestrze generującym adres (licznik programu - PC) można uzyskać dostęp do dwukrotnie większej przestrzeni pamięciowej. Z tego samego powodu zostało użytych 6 bajtów na dane umieszczone w pamięci programu, pomimo, że napis "ABCD" wraz z kończącym go bajtem o wartości 0 zajmują tylko 5 bajtów, o czym informuje raport Assemblera. Dodatkowo raport Assemblera generuje ostrzeżenie, że dla wyrównania do pełnego adresu został dodatkowo dodany bajt o wartości 0.

v. Wykonać instrukcję z linii numer 4. W rejestrze R0 znajdzie się wówczas młodsza część kodu maszynowego instrukcji z linii 5, co oczywiście jest zgodne z programem, ale nie jest zgodne z oczekiwaniami programisty. W celu wykorzystania etykiet umieszczonych w pamięci programu jako adresu do danych dla potrzeb instrukcji LPM należy zawsze przed użyciem wartości tych etykiet mnożyć przez 2. Mnożenia przez 2 można dokonać z wykorzystaniem operatora mnożenia * (*2) lub operatora przesunięcia w lewo << (<<1). Dwie pierwsze linie powyższego programu mogą więc wyglądać następująco.

```
LDI ZL, low(cmem*2)
LDI ZH, high(cmem<<1)
```

vi. Po modyfikacji kodu dokonać ponownej jego assemblacji i symulacji. Zwrócić uwagę na wynik działania instrukcji z linii 4 i 5. Podczas wykonywania tych instrukcji zwrócić uwagę na stan rejestru Z (okienko "Processor"). Należy zauważyć, że podanie jako drugiego argumentu rejestru Z z przyrostkiem + powoduje jego późniejszą inkrementację (post inkrementację). Modyfikację należy traktować jako post inkrementację, ponieważ w rezultacie wykonania tej instrukcji do rejestru R1 zostanie wpisany kod ASCII literki A, a w rejestrze Z będzie znajdował się adres komórki pamięci przechowującej kod ASCII literki B.

vii. Wykonać program do końca (zapętlić instrukcją RJMP PC) i prześledzić jego działanie.

(b) Zapis do pamięci danych

i. Dokonać assemblacji poniższego kodu.

```
LDI XL, LOW(DMEM)
LDI XH, HIGH(DMEM)

LDI R16, DMEM_END-DMEM

LOOP:
        ST X+, R0
        INC R0
        DEC R16
```

```
BRNE LOOP

RJMP PC

.DSEG
DMEM: .BYTE 8
DMEM_END:
```

Dokonać jego symulacji. Zwrócić uwagę na fakt, że w linii 4 do rejestru R16 wpisywana jest różnica pomiędzy adresem bajtu znajdującym się tuż za zarezerwowanym, przy użyciu dyrektywy .BYTE, obszarem pamięci o wielkości 8 bajtów a adresem pierwszego zarezerwowanego, z użyciem tej dyrektywy, bajtu. Innymi słowy w linii 4 do rejestru R16 zostanie wpisana wielkość zarezerwowanego obszaru pamięci (8). Dwie pierwsze linie programu powodują wpisanie do rejestru X adresu rezerwowanego obszaru pamięci. Z uwagi na fakt, że jest to pamięć danych, która jest pamięcią o dostępie 8-mio bitowym, wartość etykiety przed użyciem nie należy, przeciwnie jak w przypadku etykiet umieszczonych w pamięci programu, mnożyć przez 2.

ii. Instrukcja ST X+, R0 powoduje wpisanie do komórki pamięci danych o adresie zawartym w rejestrze X zawartości rejestru R0 z post inkrementacją rejestru X. Jest to instrukcja z adresowaniem pośrednim indeksowym, podobnie jak instrukcja o mnemoniku LPM. Instrukcja INC R0 dokonuje inkrementacji rejestru R0, a instrukcja DEC R16 dokonuje dekrementacji rejestru R16.

iii. Instrukcja BRNE jest instrukcją skoku warunkowego (adres skoku kodowany jest względnie) [2]. Działa ona ze względu na znacznik operacji arytmetycznych Z (znaczniki zera, nie mylić ze wskaźnikiem Z). Znacznik Z ustawiany jest w sytuacji, kiedy wynik operacji arytmetycznej wynosi 0, natomiast w przypadku, kiedy wynik operacji arytmetycznej jest różny od zera znacznik Z jest zerowany. W powyższym przykładzie instrukcja BRNE LOOP przekaże sterowanie do etykiety LOOP: w przypadku, kiedy znacznik Z jest wyzerowany, natomiast przekaże sterowanie do kolejnej instrukcji (RJMP PC), w przypadku, kiedy znacznik Z jest ustawiony. W powyższym przykładzie instrukcją arytmetyczną, która ma bezpośredni wpływ na stan znacznika Z badanego przez instrukcję BRNE LOOP jest oczywiście instrukcja DEC R16.
Początkowo rejestr R16 zawiera wielkość wyrażoną w bajtach obszaru pamięci i w każdej iteracji (DEC R16) jest zmniejszany o jeden. Tak długo jak wartość przechowywana w rejestrze R16 jest różna od 0, tak długo sterowanie przekazywane jest do miejsca wskazywanego etykietą LOOP:. W przypadku, kiedy jednak wartość w rejestrze R16 osiągnie wartość 0, warunek na dokonanie skoku nie będzie spełniony i pętla zostanie zakończona. Pętla wykona się taką ilość razy jaka została wpisana pierwotnie (przed etykietą LOOP) do rejestru R16.

iv. Prześledzić dokładnie na modelu działanie całego kodu programu. Przed wykonaniem każdej instrukcji przewidywać stan modyfikowanych rejestrów lub komórek pamięci i sprawdzać z wynikiem symulacji. Wykonywaniu kodu pętli LOOP należy dokładnie przyjrzeć się przynajmniej podczas jednej iteracji.

(c) Kopiowanie z pamięci programu do pamięci danych

Napisać uruchomić i przetestować program, który dokonuje kopiowania z wykorzystaniem mechanizmu pętli zdefiniowanego wcześniej napisu “12345”, 0 w obszarze pamięci programu do zarezerwowanego wcześniej obszaru pamięci danych.

(d) Kopiowanie w pamięci danych ze zmianą kolejności

i. Instrukcja LD R0, X powoduje wpisanie do rejestru ogólnego przeznaczenia R0 zawartości komórki pamięci, której adres znajduje się w rejestrze X. Jest ona instrukcją komplementarną dla instrukcji o mnemoniku ST. Posiada ona również warianty z post inkrementacją X+ oraz z pre dekrementacją -X. Korzystając z dokumentacji [2] zapoznać się ze wszelkimi wariantami instrukcji o mnemonikach ST oraz LD.

ii. Napisać uruchomić i przetestować program, który kopiuje zawartość wybranego obszaru pamięci danych, w miejsce innego obszaru pamięci danych ze zmianą kolejności tych danych. Z uwagi na fakt, że zawartości pamięci danych nie można definiować, należy zawartość obszaru źródłowego danych ustalić poprzez wykonanie kodu kopiującego z pamięci programu, lub ręcznie. Ręczne określenie zawartości pamięci jest możliwe po uruchomieniu debugowania i wciśnięciu przycisku myszy w momencie, kiedy kursor myszy wskazuje na wybraną komórkę pamięci w okienku "Memory". Dane należy wpisywać z klawiatury w notacji szesnastkowej.

(e) Adresowanie bezpośrednie

i. Dokonać assemblacji poniższego kodu programu.

```
LDI R16, 0xAB
LDI R17, 0xCD

STS DMEM, R16
STS 0, R17

LDS R18, DMEM
LDS R18, 17

.DSEG
.ORG 0x100
DMEM: .BYTE 16
```

Dyrektywa .ORG umożliwia określenie adresu w pamięci znajdujących się za nią w pliku źródłowym elementów. W powyższym przykładzie rezerwacja 16 bajtów rozpocznie się od adresu 0x100 a nie od domyślnego 0x60=96 (32 rejestru ogólnego przeznaczenia + 64 rejestry wejścia wyjścia).

ii. Instrukcja STS DMEM, R16 spowoduje wpisanie zawartości rejestru R16 do komórki pamięci danych o adresie 0x100=256. Należy zwrócić uwagę na fakt, że wszystkie instrukcje o mnemonikach ST, LD, STS, LDS (jak również LDD oraz STD) jako adres zerowy traktują rejestr R0, a nie zerową komórkę pamięci SRAM. Tak więc pod adresem zerowym dla tych instrukcji znajduje się rejestr R0, pod adresem 31 rejestr R31, pod adresem 32 zerowy port wejścia wyjścia (I/O0), a pod adresem 95 ostatni (I/O63) port wejścia wyjścia. Dopiero pod adresem 96 znajduje się zerowa komórka pamięci SRAM.
Często pod nazwą RAM rozumie się cały obszar pamięci danych, czyli rejestry ogólnego przeznaczenia, rejestry wejścia wyjścia oraz właściwa pamięć danych SRAM, natomiast akronim SRAM odnosi się do tylko i wyłącznie do pamięci danych, która dla mikrokontrolera ATmega16 posiada wielkość 1 kB.

iii. Instrukcja STS 0, R17 spowoduje wpisanie zawartości rejestru R17 do rejestru R0. Funkcjonalnie w tym przypadku można ją zastąpić instrukcją MOV R0, R17, która dodatkowo będzie

posiadać mniejszy kod maszynowy, czyli będzie bardziej optymalna. W kodzie maszynowym instrukcji MOV na zakodowanie informacji o rejestrze przeznaczenia przeznaczonych jest 5 bitów, podczas, gdy w kodzie maszynowym instrukcji STS na zakodowanie adresu przeznaczenia przewidziane są aż 2 bajty. Stąd kod maszynowy instrukcja STS zajmuje dwa słowa, a kod maszynowy instrukcji MOV tylko jedno słowo.

Wykonanie instrukcji MOV zajmuje jeden cykl zegarowy, a wykonanie instrukcji STS dwa cykle zegarowe. Instrukcja STS jest bardziej uniwersalna aniżeli instrukcja MOV, ale należy ją stosować tylko w przypadku, kiedy przeznaczeniem jest pamięć SRAM.

Podobne wnioski można wysnuć dla instrukcji LDS odnośnie argumentu źródłowego.

iv. Prześledzić dokładnie na modelu działanie całego kodu programu. Przed wykonaniem każdej instrukcji przewidywać stan modyfikowanych rejestrów lub komórek pamięci i sprawdzać z wynikiem symulacji.

(f) Zapis i odczyt rejestrów wejścia wyjścia

i. Instrukcje o mnemonikach IN oraz OUT służą do czytania oraz pisania rejestrów wejścia wyjścia. Zarówno adres rejestru wejścia wyjścia jak i indeks rejestru ogólnego przeznaczenia przekazywane są bezpośrednio.

ii. Wyszukać w dokumentacji [2] dokładnego opisu tych instrukcji. Napisać program wpisujący do rejestru (we/wy) kierunkowego DDRA portu PA wartość 0x00, do rejestru (we/wy) właściwego (w tym przypadku polaryzującego) PORTA portu PA wartość 0xFF i odczytującego do rejestru (ogólnego przeznaczenia) R0 wartość z rejestru (we/wy) wejściowego PINA portu PA. Innymi słowy napisać program w Assemblerze AVR realizujący poniższy pseudokod

```
DDRA=0x00
PORTA=0xFF
R0=PINA
```

Rejestry wejścia wyjścia służą do komunikacji z dodatkowymi modułami mikrokontrolera (sprzęg szeregowy, pamięć EEPROM, liczniki/czasomierze itp.). Budowa poszczególnych rejestrów wejścia wyjścia (w tym również tych z obecnego ćwiczenia) będzie sukcesywnie wyjaśniana w miarę potrzeb w kolejnych ćwiczeniach.

iii. Dokonać symulacji napisanego programu. Przed wykonaniem instrukcji realizującej odczyt rejestru wejścia wyjścia PINA należy w okienku “I/O View” ustawić wartość różną od wartości 0x00 i zaobserwować jej wpis do rejestru R0 po wykonaniu tej instrukcji.

(g) Napisać i przetestować program w języku Assemblera realizujący funkcje pseudokodu z poprzedniego ćwiczenia bez wykorzystywania instrukcji IN oraz OUT. Dla potrzeb instrukcji z rodziny ST(D/S)/LD(D/S) wartość adresu rejestru wejścia wyjścia PORTA, DDRA oraz PINA, można uzyskać następująco PORTA+32, DDRA+32 oraz PINA+32, gdyż rejestry wejścia wyjścia są umieszczone w przestrzeni adresowej tych instrukcji z przesunięciem równym 32 (32 rejestry ogólnego przeznaczenia znajdują się tuż przed rejestrami wejścia wyjścia). Operator + nie będzie zamieniony w żaden sposób na instrukcję mikroprocesora. Operacja ta zostanie wykonana w trakcie assemblowania i w żaden sposób nie wpłynie na wielkość kodu wynikowego.

1.4 Znaczniki operacji arytmetycznych (1 godzina)

Cel ćwiczenia

Celem ćwiczenia jest pogłębienie wiedzy na temat znaczników operacji arytmetycznych oraz zdobycie doświadczenia w wykorzystywaniu tych znaczników jako źródła dodatkowej informacji dotyczącej wyniku operacji.

Zagadnienia do przygotowania

Przed przystąpieniem do ćwiczenia należy zapoznać się z zasadami ustawiania następujących znaczników operacji arytmetycznych: znacznik przeniesienia - Carry, znacznik przepełnienia - Overflow, znacznik znaku - Sign, znacznik wartości ujemnej - Negative, znacznik zera Zero oraz znacznik przeniesienia połówkowego - Half Carry. Należy również znać binarne kody liczbowe (KNB, U2) oraz kod dziesiętny kodowany binarnie (BCD), dla których operacje arytmetyczne są wspierane tymi znacznikami.

Przebieg ćwiczenia

1. Znaczniki przeniesienia i przepełnienia

   (a) Rozpocząć nowy projekt. Zawartość głównego pliku źródłowego projektu uzupełnić poniższym kodem.

   ```
   .EQU A=100
   .EQU B=200

   LDI R16, A
   LDI R17, B

   ADD R16, R17

   RJMP PC
   ```

   Dyrektywa .EQU przypisuje wyrażenie nazwie [1]. W tym przypadku nazwa A będzie przechowywała wartość liczbową 100. Instrukcja ADD dodaje zawartość rejestru R17 do rejestru R16 [2]. Należy zwrócić uwagę, że niezależnie od kodowania liczb (U2 czy też KNB) do dodawania liczb stosowana jest ta sama instrukcja.

   (b) Sprawdzić stan znaczników po wykonaniu operacji dodawania. O ile wyjaśnienie stanu znacznika C jest oczywiste o tyle wyjaśnienie stanu znacznika V może być kłopotliwe. Należy jednak zwrócić uwagę na fakt, że rejestry są 8-mio bitowe oraz, że wartość 200 w kodzie NKB posiada reprezentację binarną, która w kodzie U2 stanowi wartość -128-128+200=-56. Stany wszystkich znaczników są widoczne w rejestrze SREG dostępnym podczas symulacji w okienku “Processor”.

   (c) Wykonać operacje dodawania dla podanych poniżej par liczb, przed wykonaniem symulacji dookreślić wartości tych liczb w zależności od potrzeb albo w kodzie U2 albo kodzie NKB, a następnie określić zgodnie z zasadami dodawania sumę oraz stany znaczników C oraz V.

NKB		U2		A+B		C	V
A	B	A	B	NKB	U2		
0	0						
100			100				
250	200						
64	64						
128			-128				
255	255						
128			-1				

Wszelkie niezgodności z wynikami symulacji lub niejasności należy bezwzględnie szczegółowo wyjaśnić.

2. Znaczniki znaku oraz wartości ujemnej

   (a) Wykonać operacje dodawania dla podanych poniżej par liczb, przed wykonaniem symulacji dookreślić wartości tych liczb w zależności od potrzeb albo w kodzie U2 albo kodzie NKB, a następnie określić zgodnie z zasadami dodawania sumę oraz stany znaczników S, N oraz V.

NKB		U2		A+B		S	N	V
A	B	A	B	NKB	U2			
0	0							
130			-100					
		-10	-20					
100	100							

   Wszelkie niezgodności z wynikami symulacji lub niejasności należy bezwzględnie szczegółowo wyjaśnić.

   (b) Na podstawie uzyskanych wyników określić relacje S(N,V), N(S,V) oraz V(N,S).

3. Znacznik przeniesienia połówkowego

   (a) Określić reprezentacje liczb 29 oraz 38 w kodzie BCD. Wykonać operację dodawania reprezentacji BCD tych liczb. Określić stan znacznika H, podjąć decyzję o konieczności dokonania ewentualnej poprawki BCD. Wykonać poprawkę jeżeli była konieczna. Sprawdzić poprawność uzyskanego wyniku (po poprawce BCD). Korzystając z symulacji sprawdzić poprawność wyznaczonego wcześniej znacznika H.

   (b) Wykonać te same czynności dla par liczb {11, 33}, {25, 35} oraz {90, 20}.

4. Znacznik zera

   (a) Podać dwie pary liczb, których dodanie powoduje wyzerowanie znacznika Z oraz dwie pary liczb, których dodanie powoduje ustawienie znacznika Z.

5. Ustawianie znaczników dla operacji odejmowania

   (a) Podać wartości argumentów, dla których instrukcja odejmowania (SUB [2]) powoduje jednoczesne ustawienie znacznika C oraz wyzerowanie znacznika V. Sprawdzić zgodność przewidywań z wynikiem symulacji. Jaką dokładnie rolę pełni znacznik C w instrukcji odejmowania.

(b) Podać wartości argumentów, dla których instrukcja odejmowania spowoduje jednoczesne ustawienie znacznika V, wyzerowanie znacznika S oraz ustawienie znacznika N. Czy możliwe jest aby w wyniku operacji odejmowania lub też operacji dodawania otrzymać wszystkie trzy wyżej wymienione znaczniki (V, S, N) ustawione? Czy możliwe jest aby w wyniku operacji odejmowania lub też operacji dodawania otrzymać wszystkie trzy wyżej wymienione znaczniki (V, S, N) wyzerowane?

6. Dodawanie i odejmowanie liczb dwu i więcej bajtowych

   (a) Który ze znaczników operacji arytmetycznych powinien zostać wykorzystany przy implementacji dodawania dwóch 16-to bitowych liczb w kodzie NKB, a który podczas wykonywania operacji odejmowania liczb zapisanych w tym kodzie?

   (b) Który ze znaczników operacji arytmetycznych powinien zostać wykorzystany do implementacji dodawania, a który do implementacji odejmowania liczb 16-to bitowych zapisanych w kodzie U2?

1.5 Podprogramy (2 godziny)

Cel ćwiczenia

Celem ćwiczenia jest nabycie doświadczenia w pisaniu, uruchamianiu oraz symulacji podprogramów z wykorzystaniem AVR Studio w języku Assemblera. W trakcie ćwiczenia utrwalone zostaną mechanizmy stosu oraz działanie instrukcji wywołania i powrotu z podprogramu.

Zagadnienia do przygotowania

Przed przystąpieniem do ćwiczenia należy zapoznać się z zagadnieniem podprogramów w języku Assemblera, zastosowaniem pamięci stosu w podprogramach oraz instrukcjami wywoływania i powrotu z podprogramów.

Przebieg ćwiczenia

1. Proste podprogramy

```
.INCLUDE "m16def.inc"

LDI R16, LOW(RAMEND)
OUT SPL, R16

LDI R16, HIGH(RAMEND)
OUT SPH, R16

NOP
CALL PODPROGRAM2
NOP
CALL PODPROGRAM1
NOP

RJMP PC

PODPROGRAM1:
        NOP
```

```
        NOP
        RET

PODPROGRAM2:
        NOP
        CALL PODPROGRAM1:
        NOP
        CALL PODPROGRAM1:
        NOP
        RET
```

RAMEND jest nazwą zdefiniowaną w pliku m16def.inc, pod którą kryje się adres ostatniej komórki pamięci RAM. Rejestr wskaźnika stosu SP (ang. Stack Pointer) w mikrokontrolerze AVR znajduje się w przestrzeni wejścia wyjścia. Jest on 16-to bitowy i składa się z dwóch rejestrów 8-mio bitowych SPH oraz SPL. Zapis do tych rejestrów wykonywany jest oczywiście z wykorzystaniem instrukcji OUT.

Instrukcja CALL wywołuje podprogram. Adres podprogramu w instrukcji CALL przekazywany jest bezpośrednio. Instrukcja RET powraca z podprogramu. Powrót z podprogramu polega na przekazaniu sterowania do instrukcji znajdującej się tuż za instrukcją wywołania podprogramu.

(a) Prześledzić działanie powyższego programu. Zwrócić uwagę na stan wskaźnika stosu oraz korelację pomiędzy stanem licznika programu a wartościami odkładanymi na stos, w tym celu należy obserwować ostatnie komórki pamięci danych (stos) oraz oczywiście licznik programu.

(b) Określić kierunek modyfikacji wskaźnika stosu oraz stwierdzić czy jest to pre czy też może post modyfikacja. Stwierdzić jak jest kolejność bajtowa podczas odkładania adresu kolejnej instrukcji na stos.

2. Dodawanie i odejmowanie liczb 32 bitowych

(a) Napisać dwa podprogramy, jeden realizujące dodawanie, a drugi realizujący odejmowanie liczb 32 bitowych. Skorzystać z wniosków z poprzedniego ćwiczenia. Wartości tych liczb do podprogramów przekazywać z wykorzystaniem rejestrów ogólnego przeznaczenia. Na przykład czwórka rejestrów R19:R18:R17:R16 może przechowywać jeden z argumentów źródłowych.

Wartości liczb do programu należy wprowadzać z wykorzystaniem dyrektyw .EQU, dzięki czemu modyfikacja argumentu będzie wymagała edycji tylko w jednym miejscu. Do wyłuskania ośmiu najmłodszych bitów można użyć funkcji low(), do wyłuskania bitów o indeksach od 8 do 15 można użyć oczywiście funkcji high(). Aby wyłuskać bity o indeksach od 16 do 23 należy użyć funkcji byte3(), a do ośmiu najstarszych bitów funkcji byte4() [1].

(b) Wykonać dodawanie dwóch liczb, do zakodowania których potrzebne są przynajmniej 3 bajty np (1e6 oraz 2e6), sprawdzić poprawność wyniku zapisanego w czwórce rejestrów wynikowych.

(c) Wykonać odejmowanie dwóch liczb. W celu sprawdzenia poprawności działania funkcji odejmującej dla liczb w kodzie U2 przyjąć odjemnik większy od odjemnej, np. wykonać operację 100 - 115.

1.6 Sortowanie tablicy liczb (2 godziny)

Cel ćwiczenia

Celem ćwiczenia jest jest praktyczne zapoznanie z instrukcjami warunkowymi mikrokontrolera oraz zdobycie doświadczenia w posługiwaniu się kodowaniem NKB oraz U2. Ćwiczenie umożliwia poznanie od strony praktycznej algorytmu sortowania bąbelkowego oraz sposobu jego implementacji w języku Assemblera.

Zagadnienia do przygotowania

Przed przystąpieniem do ćwiczenia należy zapoznać się z regułami ustawiania znaczników przeniesienia oraz znaku jak również ze sposobem kodowania liczb w kodzie NKB oraz U2 na poziomie rozumienia ustalania relacji pomiędzy dwoma liczbami z wykorzystaniem wyżej wymienionych znaczników. Zapoznać się z algorytmem sortowania bąbelkowego oraz wyszukać instrukcje [2], których działanie zależy od znaczników C oraz S.

Przebieg ćwiczenia

1. W głównym pliku źródłowym nowo utworzonego projektu zdefiniować 10-cio elementową tablicę liczb 8-mio bitowych o takich wartościach, aby posiadały one między sobą różne relacje w kodach NKB oraz U2. Przykładowo tablica może zawierać elementy 0, 200, 100, ponieważ ich kolejność w kodzie NKB będzie 0, 100, 200, natomiast kolejność w kodzie U2 będzie 200(-56), 0 ,100.

2. W pamięci danych zadeklarować miejsce na dwie 10 elementowe tablice. Zadaniem jednej z tych tablic jest przechowywanie (docelowe oraz tymczasowe) liczb porządkowanych w kodzie KNB, natomiast drugiej jest przechowywanie liczb w kodzie U2. Z uwagi na fakt, że definicja tablicy źródłowej dotyczy pamięci FLASH porządkowanie przeprowadzać wyłącznie na kopii w pamięci danych. Do programu dołączyć ciąg instrukcji kopiujących z pamięci FLASH do obu obszarów pamięci danych.

3. W celu zaprogramowania porządkowania w kodzie KNB wyszukać instrukcję, której wykonanie jest zależne od stanu znacznika C oraz w celu zaprogramowania porządkowania w kodzie U2 wyszukać instrukcję, której działanie jest zależne od znacznika S. Zapoznać się również z działaniem instrukcji CP [2], która będzie służyła do zwracania raportu (znaczniki C i S) na temat relacji dwóch liczb.

4. Napisać kod warunkowy, który dokonuje zamiany zawartości dwóch komórek pamięci danych w przypadku, kiedy komórka o niższym adresie zawiera wartość mniejszą aniżeli komórka o wyższym adresie. Kod warunkowo przestawiający napisać i przetestować w dwóch wariantach (KNB oraz U2). Na tym etapie tworzenia programu indeksy obu komórek będą stałymi, wartościami znanymi w trakcie assemblacji.

5. Każdy z napisanych w poprzednim punkcie ćwiczenia programów warunkowych należy umieścić w pętli, w której stan licznika zmienia się od 0 do n-2 (n=10). Jako indeksów porównywanych komórek pamięci należy użyć stanu licznika pętli oraz stanu licznika zwiększonego o 1, co będzie gwarantowało, że w każdej iteracji pętli porównywane będą dwie sąsiednie komórki pamięci.

6. Napisać kod, który tak długo będzie wykonywał jedną z powstałych w poprzednim punkcie pętli jak długo w każdej z nich będzie następowała choćby jedna zamiana wartości komórek pamięci. Aby po zakończeniu wykonywania się pętli posiadać informację o tym, czy w trakcie jej wykonywania nastąpiła przynajmniej jedna zamiana, można użyć znacznika T, który będzie ustawiany w trakcie zamiany i później badany na ewentualność kolejnego wykonania pętli lub zakończenia programu.

7. Napisane programy sortujące dobrze przetestować. Wyjaśnić różnice zawartości obu tablic docelowych.

8. Zaproponować szczegóły (komparator) implementacji programu sortującego liczby n bajtowe.

1.7 Porty wejścia wyjścia ogólnego przeznaczenia (2 godziny)

Cel ćwiczenia

Celem ćwiczenia jest zdobycie wiedzy na temat budowy, działania oraz obsługi na poziomie języka Assemblera portów ogólnego przeznaczenia. Ćwiczenia umożliwia również zdobycie doświadczenia w korzystaniu a dokumentacji mikrokontrolera oraz w posługiwaniu się peryferiami zestawu uruchomieniowego.

Zagadnienia do przygotowania

Przed przystąpieniem do ćwiczenia należy zapoznać się z działaniem podstawowych elementów techniki cyfrowej, należą do nich przerzutniki D (Flip-Flop), zatrzaski D (Latch), bufory trójstanowe, funktory logiczne, klucze analogowe i bramka z układem Schmitta. Należy również znać działanie tranzystorów MOS, posiadać podstawową wiedzę z teorii obwodów, nie obcym powinna również być idea magistrali komputerowej.

Przebieg ćwiczenia

1. Tryb pracy wyjściowej

   (a) Mając na uwadze rysunek 23 zamieszczony w dokumentacji [3] należy zauważyć, że Pxn jest wyprowadzeniem mikrokontrolera reprezentującym pojedynczy bit portu cyfrowego, a "DATA BUS" jest magistralą łączącą moduł portu z rdzeniem mikrokontrolera.

   (b) Poprzez tryb pracy wyjściowej rozumie się, taką konfigurację portu, w której dane są wyprowadzane z rdzenia mikrokontrolera poprzez magistralę i moduł portu na wyprowadzenie Pxn. Port konfigurowany jest jako wyjściowy w przypadku sterowania układami zewnętrznymi, np. przy podłączaniu diod LED, przetworników elektroakustycznych lub wejść innych układów cyfrowych.

   (c) Aby skonfigurować port jako wyjściowy należy do przerzutnika DDxn wpisać stan wysoki. Wówczas wyjściowy bufor trójstanowy (środek rysunku) jest aktywny (przekazuje stan wejścia na wyjście). Stan wyjścia Pxn w trybie wyjściowym określany jest więc stanem przerzutnika PORTxn.

   (d) Dokonać assemblacji poniższego kodu źródłowego.

```
.include "m16def.inc"

LDI R16, 0xFF
OUT DDRA, R16

LDI R16, 0x55
OUT PORTA, R16

RJMP PC
```

   Wpisanie do rejestru wejścia wyjścia DDRA wartości 0xFF spowoduje ustawienie wszystkich bitów (linii) portu PA w trybie wyjściowym. Rejestr DDRA zawiera 8 bitów reprezentujących stany wszystkich przerzutników DDxn (x=A) o indeksach n od 0 do 7. Podobnie rejestr wejścia wyjścia PORTA reprezentuje stany wszystkich przerzutników PORTxn (x=A) o indeksach n od 0 do 7.

W powyższym przykładzie na port PA mikrokontrolera zostanie wystawiona sekwencja naprzemiennie ustawionych i wyzerowanych bitów (01010101).

(e) Wpisać do pamięci programu mikrokontrolera zawartość pliku uzyskanego w trakcie assemblacji. Skonfigurować zestaw uruchomieniowy tak, aby stan portu PA był przekazywany na linijkę diod LEDs [5].

(f) Zauważyć, że w trybie pracy wyjściowej pojedynczy bit portu może znajdować się albo w stanie wysokim, albo w stanie niskim. Oba te stany realizowane są wówczas, kiedy bufor trójstanowy (środek rysunku) znajduje się w stanie aktywnym.

2. Tryb pracy wejściowej

(a) Aby skonfigurować port w trybie pracy wejściowej należy ustawić bufor trójstanowy (środek rysunku) w trybie wysokiej impedancji. W tym celu do przerzutnika DDxn należy wpisać stan niski. Stan przerzutnika PORTxn steruje w tym trybie pracy portu wstępną polaryzacją wyprowadzenia Pxn. W przypadku, kiedy przerzutnik PORTxn znajduje się w stanie niskim na bramce tranzystora (lewy górny róg rysunku) będzie znajdował się stan wysoki. Z uwagi na brak napięcia pomiędzy źródłem (górne wyprowadzenie) tranzystora a jego bramką, tranzystor będzie znajdował się w stanie odcięcia, a górne wyprowadzenie rezystora będzie odłączone. Mówimy wówczas o pracy w trybie wejściowym z brakiem polaryzacji, bez polaryzacji (wstępnej) lub czasami prawdopodobnie niefortunnie "bez podwieszania" (ang. Pull-Up). Tryb pracy wejściowej bez polaryzacji może zostać wykorzystany do podłączania wyjść innych układów cyfrowych, nie bardzo jednak nadaje się do podłączania przycisków.

(b) Możliwość ustawienia pracy portu jako wejściowej ze wstępną polaryzacją umożliwia podłączanie przycisków bezpośrednio do wyprowadzeń mikrokontrolera. Jedno wyprowadzenie przycisku podłącza się wówczas najczęściej do masy, a drugie do wyprowadzenia portu mikrokontrolera. W przypadku, kiedy przycisk jest wciśnięty (zwarty) na wybranym wejściu mikrokontrolera znajduje się poziom niski, natomiast w przypadku, kiedy przycisk nie jest wciśnięty (rozwarty) na wybranym wyprowadzeniu mikrokontrolera, będzie się znajdował, dzięki wewnętrznemu rezystorowi polaryzującemu stan wysoki. W przypadku braku polaryzacji wstępnej w momencie, kiedy przycisk byłby wyciśnięty stan wejścia mikrokontrolera byłby nieokreślony.

(c) W trybie wejściowym wstępną polaryzację wybranego bitu (n) portu (x) uzyskuje się w momencie, kiedy stan przerzutnika PORTxn jest wysoki, stan przerzutnika DDxn jest niski oraz stan bitu globalnego blokowania polaryzacji wstępnej PUD (ang. Pull-Up Disable) jest niski. Wówczas na bramce tranzystora MOS znajduje się stan niski i tranzystor przewodzi - podłącza rezystor do napięcia zasilania.

(d) Odczytu stanu wyprowadzenia Pxn należy wykonywać poprzez odczyt przerzutnika PINxn. W przestrzeni rejestrów wejścia wyjścia są to zasoby PINx, gdzie x koduje nazwę portu.

(e) Prześledzić drogę sygnału od wyprowadzenia Pxn, poprzez klucz analogowy, bramkę z wejściem Schmitta i synchronizator do rejestru PINxn. Zauważyć, że synchronizator powoduje wprowadzenie opóźnienia o wartości od 0.5 do 1.5 cyklu zegarowego.

3. Prosty program kopiujący stan jednego portu na inny port

(a) Dokonać assemblacji poniższego kodu i zaprogramować mikrokontroler.

```
.include "m16def.inc"

LDI R16, 0xFF
OUT DDRA, R16

OUT PORTB, R16

LDI R16, 0x00
OUT DDRB, R16

LOOP:

IN R0, PINB
OUT PORTA, R0

RJMP LOOP
```

Program konfiguruje port PA jako wyjściowy oraz port PB jako wyjściowy z polaryzacją, a następnie cyklicznie, w pętli LOOP kopiuje stan wejść portu PB na wyjścia portu PA.

(b) Podłączyć 4 przyciski używając wyprowadzeń R0, R1, R2, R3 złącza JP15 [5] do najstarszych bitów mikrokontrolera portu PB (złącze Zl2 wyprowadzenia 7, 6, 5, 4) z wykorzystaniem dodatkowych przewodów. Skonfigurować zestaw uruchomieniowy tak, aby stan portu PA był przekazywany na diody LEDs. Złączem JP26 należy skonfigurować klawiaturę do pracy liniowej (niematrycowej) ustawiając zworkę w pozycji 4x1.

(c) Obserwować stany diod w zależności od stanu przycisków.

(d) Tymczasowo wyłączyć polaryzację wstępną komentując instrukcję (";" lub "//") z linii 6. Przy obserwacji stanu diod w zależności od stanu przycisków należy zauważyć, że po wciśnięciu przycisku dioda zapala się z małym opóźnieniem lub nie zapala się wcale. Można również odpiąć przewód od złącza JP15 i podłączać odpiętą końcówkę przewodu naprzemiennie do zasilania (+5V - dolne piny złącza Zl3) lub masy (GND - dół złącza Zl1). Na podstawie obserwacji wysnuć wnioski dotyczące parametrów elektrycznych wejścia, pomocnym może być rysunek 22 zamieszczony w dokumentacji [3].

(e) Dokonać takiej modyfikacji programu, aby stan diod o indeksach 7 i 6 był negacją stanu wejściowego bitów 7 i 6 portu PB, natomiast stan diod o indeksach 5 i 4 przedstawiał stan bitów 5 i 4 portu PB w sposób prosty. Należy w tym celu wykorzystać instrukcję realizującą funkcję EXOR [2].

4. Opóźnienie synchronizatora

(a) Dokonać assemblaji poniższego kodu.

```
.include "m16def.inc"

LDI R16, 0xFF
OUT DDRA, R16
```

```
LOOP:

INC R20
OUT PORTA, R20
IN R0, PINA

RJMP LOOP
```

(b) Uruchomić śledzenie na modelu. Zauważyć, że stan rejestru R0 w stosunku do stanu rejestru R20 opóźniony jest o jeden cykl wykonania pętli LOOP. Jest to spowodowane oczywiście pracą synchronizatora.

(c) Dokonać takiej modyfikacji programu, aby stan rejestru R0 był ustawiany na podstawie stanu rejestru R20 w każdej iteracji pętli.

1.8 Realizacja opóźnień (1 godzina)

Cel ćwiczenia

Celem ćwiczenia jest nabycie doświadczenia w pisaniu programów w języku Assembler realizujących opóźnienie o ściśle określonych wartościach. Ćwiczenie ma również na celu uświadomienie obecności pasożytniczego parametru instrukcji jakim jest czas jej wykonania.

Zagadnienia do przygotowania

Korzystając z dokumentacji [2] ustalić czasy wykonywania instrukcji o mnemonikach LDI, DEC oraz BRNE. Powtórzyć dokładnie działanie tych instrukcji, ustalić w jaki sposób wynik działania instrukcji DEC może wpływać na zachowanie instrukcji BRNE.

Przebieg ćwiczenia

1. Pojedyncza pętla opóźniająca

   (a) Wykonać symulację działania na modelu poniższego kodu programu obserwując dokładnie zmiany stanów rejestru R16 oraz znacznika zera.

   ```
   .include "m16def.inc"
   .equ n=5

   LDI R16, n
   LOOP:
           DEC R16
   BRNE LOOP

   RJMP PC
   ```

   (b) Określić wartość opóźnienia wyrażonego w cyklach zegara generowanego przez kod zawarty w liniach od 4 do 7 w zależności od parametru n. Pamiętając o tym, że rejestry są 8-mio bitowe określić maksymalne opóźnienie wyrażone w cyklach zegarowych, które możliwe jest do uzyskania w wyżej wymienionej konstrukcji opóźniającej, podać wartość n, przy której występuje maksymalne opóźnienie.

(c) Obserwując licznik cykli dokonać symulacji sprawdzającej czas wykonania kodu zamieszczonego w liniach od 4 do 7 i porównać uzyskany wynik z wyliczoną w poprzednim punkcie wartością opóźnienia wyrażoną w cyklach zegarowych.

(d) Uwzględniając częstotliwość pracy mikrokontrolera określić maksymalny czas opóźnienia w sekundach możliwy do uzyskania przy użyciu wyżej podanego kodu.

2. Podwójna pętla opóźniająca

(a) Przeanalizować i dokonać symulacji działania poniższego kodu.

```
.include "m16def.inc"
.equ n=5
.equ m=2

LDI R17, m
LOOP2:
        LDI R16, n
        LOOP1:
                DEC R16
        BRNE LOOP1
        DEC R17
BRNE LOOP2

RJMP PC
```

Zastanowić się czy powyższy program będzie działał zgodnie z oczekiwaniami jeżeli instrukcja DEC R17 zostanie przeniesiona z linii 11 do linii 8 tak, że będzie znajdowała się przed etykietą LOOP1:.

(b) Określić opóźnienie generowane przez instrukcje zamieszczone w liniach od 5 do 12 w zależności od wartości parametrów n oraz m.

(c) Określić maksymalne opóźnienie wyrażone w sekundach możliwe do uzyskania z wykorzystaniem powyższego kodu.

3. Program sterujący diodami

(a) Na podstawie programu zawartego w poprzednim punkcie ćwiczenia napisać kod realizujący potrójną pętlę opóźniającą. Wyznaczyć wartość opóźnienia generowanego przez kod potrójnej pętli opóźniającej w zależności od liczby iteracji na każdym poziomie pętli (m, n oraz np. k). Dobrać tak liczbę iteracji każdego poziomu, aby opóźnienie generowane przez całą potrójną pętle było zbliżone do 1 s.

(b) Na podstawie utworzonego kodu opóźniającego napisać podprogram o nazwie sekunda realizujący opóźnienie jednej sekundy.

(c) Wykorzystując podprogram sekunda napisać program, który będzie zwiększał stan diod LEDs co okres 1 s.

1.9 Eliminacja wpływu drgań styków (2 godziny)

Cel ćwiczenia

Celem ćwiczenia jest ugruntowanie wiedzy dotyczącej drgań styków oraz eliminacji ich wpływu na stan systemu/układu cyfrowego. Ćwiczenie umożliwia również głębsze zapoznanie z podstawowymi instrukcjami warunkowymi oraz zdobycie umiejętności w ich wykorzystywaniu w celu pisania prostych programów realizujących interfejsy wejściowe użytkownika. Samo przygotowanie się do ćwiczenia pozwoli nabrać doświadczenia w pozyskiwaniu wiedzy na podstawie materiałów źródłowych dotyczących mikrokontrolera ATmega16.

Zagadnienia do przygotowania

Przed przystąpieniem do ćwiczenia należy zapoznać się z działaniem instrukcji warunkowych, których wykonanie zależne jest od stanu wybranych bitów rejestrów wejścia wyjścia (instrukcje o mnemonikach SBIC i SBIS) oraz znacznika kopii T (instrukcje o mnemonikach BRTS i BRTC) [2]. Należy również zapoznać się z instrukcjami manipulującymi stanem znacznika kopii T (instrukcje o mnemonikach SET i CLT). Uzupełnić wiedzę dotyczącą zagadnienia drgań styków oraz sposobu eliminacji ich wpływu na stan systemu cyfrowego a także przygotować przykładowy algorytm realizujący taką eliminację.

Przebieg ćwiczenia

1. Prosty wstępny

   Napisać i uruchomić program realizujący odczyt wybranego bitu portu wejściowego i przekazujący jego stan na inny port lub wybrany bit tego portu. Do wybranego portu wejściowego podłączyć przycisk [5] a port, wyjściowy podłączyć na diod LEDs. Zwrócić uwagę na właściwe ustawienie kierunków portów oraz na włączenie polaryzacji portu wejściowego.

2. Wykrywanie wciśnięcia przycisku

   (a) Opracować wspólnie z prowadzącym na tablicy algorytm wykrywający zbocze opadające (wciśnięcie przycisku powoduje zmianę stanu z wysokiego niski) na wybranym bicie portu wejściowego. Wykrywanie zbocza opadającego można realizować poprzez ciągłe porównywanie dwóch kolejnych stanów wybranego bitu portu wejściowego. Raportowanie zbocza opadającego należy przeprowadzić w sytuacji, kiedy bieżący odczyt daje stan niski podczas, gdy odczyt poprzedni dawał stan wysoki. Jako zmienną pamiętającą stan poprzedni można wykorzystać znacznik T.

   (b) Po uzgodnieniu algorytmu wspólnie na tablicy dokonać jego zakodowania z wykorzystaniem instrukcji Assemblera. Raportowanie zbocza narastającego należy przeprowadzić poprzez inkrementację licznika (dowolny, niewykorzystywany rejestr ogólnego przeznaczenia), którego stan będzie cyklicznie wystawiany na diody LEDs. Zauważyć, że pojedyncze wciśnięcie przycisku powoduje czasami zwiększanie stanu licznika o wartość większą od 1, co jest spowodowane pojawiającymi się drganiami sygnału elektrycznego, wywołanymi mechanicznymi drganiami styków - szczególnie efekt ten objawia się podczas puszczania przycisku.

3. Eliminacja wpływu drgań styków

   (a) Dodać do programu podprogram realizujący opóźnienie o wartości około 100 ms.

   (b) W miejscu programu, które jest osiągane w momencie wykrycia zbocza opadającego wstawić kod wywołujący podprogram opóźniający a za nim kod sprawdzający czy przycisk jest nadal wciśnięty. W przypadku, kiedy przycisk nie jest wciśnięty należy przekazać sterowanie do pętli głównej. W

sytuacji, kiedy przycisk jest wciśnięty należy wykonać kod, który do tej pory był wykonywany w przypadku wykrycia zbocza opadającego.

(c) Zmniejszyć opóźnienie metodą połowienia przedziału do najmniejszej wartości, przy której drgania styków nie są obserwowane.

4. Obsługa większej ilości przycisków

(a) W celu dodania obsługi większej ilości przycisków należy program zmodyfikować w taki sposób, aby wykrywać zbocze opadające na którymkolwiek obsługiwanym przycisku. W tym celu należy wykrywać sytuacje, w których stan poprzedni portu posiada taką reprezentację, w której wszystkie bity są ustawione, a stan bieżący taką w której przynajmniej jeden bit jest wyzerowany. Z uwagi na fakt, że porównanie będzie dotyczyło większej liczby bitów należy tuż na początku każdej iteracji przechwycić (wpisać do rejestru) stan wszystkich bitów portu, do którego są podłączone przyciski i później (po opóźnieniu) porównywać stan bieżący przycisków ze stanem przechwyconym.

(b) Rozbudować tą część kodu, która odpowiedzialna była za wykonywanie inkrementacji licznika wystawianego na diody LEDs o kod realizujący inkrementację licznika w sytuacji, kiedy wciśnięty jest przycisk 0, dekrementację licznika, kiedy wciśnięty jest przycisk 1, zerowanie licznika w przypadku, kiedy wciśnięty jest przycisk 2 oraz na przykład wstawianie wartości 128 w przypadku, kiedy wciśnięty zostanie przycisk 3.

1.10 Obsługa pamięci EEPROM (2 godziny)

Cel ćwiczenia

Celem ćwiczenia jest zdobycie umiejętności obsługi pamięci EEPROM mikrokontrolera ATmega16 na poziomie języka Assemblera. Ćwiczenie umożliwia pogłębienie doświadczenia w wykorzystywaniu instrukcji warunkowych mikrokontrolera oraz zwiększenie umiejętności posługiwania się dokumentacją.

Zagadnienia do przygotowania

Korzystając z dokumentacji [3] zapoznać się z opisem pamięci EEPROM. Ustalić sekwencję stanów bitów portu kontrolnego pamięci EEPROM (EECR) powodującą zapis oraz sekwencję powodującą odczyt pamięci EEPROM. Przeanalizować kod programu wraz z opisem z bieżącego ćwiczenia tak, aby zajęcia laboratoryjne wykorzystać maksymalnie na pracę z zestawem i nie marnować czasu na pracę, którą można wykonać poza laboratorium.

Przebieg ćwiczenia

1. Zapis pamięci EEPROM

(a) Z dokumentacji [3] skopiować kod podprogramu zapisującego do pamięci EEPROM w miejsce głównego pliku źródłowego projektu.

(b) Rozbudować program tak, aby następowało wywołanie podprogramu zapisującego do pamięci EEPROM po uprzednim określeniu adresu (rejestry R18 i R17) oraz danej (R16). Należy pamiętać o zainicjalizowaniu stosu.

(c) Dokładnie prześledzić na modelu działanie programu na przykładzie zapisu do pamięci EEPROM wartości 0xAA pod adres 0x180.

2. Odczyt pamięci EEPROM

   Postąpić dokładnie tak, jak w poprzednim punkcie ćwiczenia z przykładem z dokumentacji dotyczącym odczytu pamięci EEPROM. Załadować do rejestrów R18 i R17 adres komórki, która ma zostać odczytana a następnie wywołać podprogram odczytujący pamięć EEPROM. Prześledzić działanie programu na przykładzie odczytu wcześniej zapisanej komórki pamięci, jak również innych komórek, których zawartość można modyfikować ręcznie podczas wykonywania symulacji na modelu w AVR Studio.

3. Kopiowanie zawartości obszaru pamięci

   Wykorzystując podprogramy zapisu i odczytu pamięci EEPROM napisać program dokonujący kopiowania zawartości wybranego obszaru pamięci FLASH do pamięci EEPROM, a następnie z pamięci EEPROM do pamięci SRAM. Fragment źródłowy pamięci FLASH może stanowić np. napis “ABCD”, 0, natomiast wybrany fragment w pamięci EEPROM powinien zawierać komórki o adresach 0xFF oraz 0x100, co ma na celu przetestowanie działania programu na ewentualność poprawnej inkrementacji adresu 9-cio bitowego. Należy pamiętać, że dyrektywą, która pozwala przełączyć na pamięć EEPROM jest “.cseg” natomiast dyrektywą pozwalającą dokonać określenia adresu, od którego zostanie wykonana rezerwacja lub definicja pamięci jest “.org”.

4. Edytor pamięci EEPROM

   (a) Utworzyć nowy projekt. Poniższy kod wprowadzić (należy kopiować z wykorzystaniem schowka) jako zawartość głównego pliku źródłowego projektu. Zaprogramować układ.

```
.include "m16def.inc"

start:

ldi r16, low(ramend)
out spl, r16
ldi r16, high(ramend)
out sph, r16

ldi r16, 0x00
out ddrb, r16

ldi r16, 0x0f
out portb, r16

ldi r16, 0xff
out ddra, r16

ldi r16, 0xff
out porta, r16

ldi r19, 0x01
ldi r18, 0x00
ldi r22, 0x00
ldi r23, 0x00
```

```
main:

rcall key
rcall menu

rjmp main


menu:

cpi r18, 0
breq menu_end

menuKey0:
sbrs r18, 0
rjmp menuKey1

rol r19
brcc menuKey1
ldi r19, 0x01

menuKey1:

sbrs r18, 1
rjmp menuKey2

in r0, porta
eor r0, r19
out porta, r0

menuKey2:

sbrs r18, 2
rjmp menuKey3

in r0, porta
out eedr, r0
clr r0
out eearh, r0
out eearl, r22

rcall eeWrite

inc r22
```

```
andi r22, 0x03

menuKey3:

sbrs r18, 3
rjmp menuNoKey

clr r0
out eearh, r0
out eearl, r23

rcall eeRead

in r0, eedr
out porta, r0

inc r23
andi r23, 0x03

menuNoKey:

ldi r18, 0
menu_end:
ret


key:

in r16, pinb
ori r16, 0xf0

cpi r16, 0xff
brne keyPressed
bset sreg_t
ret
keyPressed:

brtc key_end

rcall delay

in r17, pinb
ori r17, 0xf0

cp r16, r17
```

```
brne key_end

bclr sreg_t
mov r18, r16
com r18

key_end:
ret


delay:

ldi r20, 0x80
d0:
ldi r21, 0x80
d1:
dec r21
brne d1
dec r20
brne d0

ret


eeRead:

sbic eecr, eewe
rjmp eeRead

sbi eecr, eere

ret


eeWrite:

sbic eecr, eewe
rjmp eeWrite

sbi eecr, eemwe
sbi eecr, eewe

ret

.eseg
```

```
.org 0x00
ememˆ: .db 0x8c, 0xaa, 0x55, 0xf0
```

Program umożliwia edycję bajtu oraz zapis tego bajtu do pamięci EEPROM jak również odczyt pamięci EEPROM. Przycisk podłączony do bitu 0 portu PB umożliwia wybór edytowanego bitu, natomiast przycisk podłączony do bitu 1 portu PB umożliwia jego edycję, która polega na negowaniu stanu bitu. Przycisk podłączony do bitu 2 portu PB daje możliwość zapisu do pamięci EEPROM, a przycisk podłączony do bitu 3 tegoż portu umożliwia odczyt pamięci EEPROM. Stan edytowanego lub odczytanego z pamięci EEPROM bajtu wypisywany jest na porcie PA, który należy oczywiście podłączyć do diod LEDs.

(b) Testowanie działania programu

i. Wciskając przycisk 3 dokonać odczytu pamięci EEPROM, porównać odczytywane wartości z zawartością pamięci EEPROM określoną w ostatniej linii programu. Zauważyć, że cyklicznie odczytywane są 4 komórki pamięci EEPROM.

ii. Używając przycisków 0 oraz 1 dokonać edycji bajtu, a następnie wciskając przycisk 2 dokonać zapisu do pamięci EEPROM. W ten sposób dokonać 4 kolejnych zapisów.

(c) Analiza programu źródłowego

i. Podprogram key odczytuje stan klawiatury i wpisuje informację w postaci prostej do rejestru R18. Podprogram ten działa na tej samej zasadzie co program obsługujący klawiaturę z poprzedniego ćwiczenia. Nie będzie on analizowany szczegółowo, podobnie podprogram delay.

ii. Podprogramy eeWrite oraz eeRead również nie będą analizowane, ponieważ zawierają kod, który był już omawiany w tym ćwiczeniu.

iii. Pętla główna main ma za zadanie cykliczne uruchamianie odczytu stanu przycisków oraz wykonywanie podprogramu menu, którego zadaniem jest określenie funkcjonalności interfejsu użytkownika.

iv. Podprogram menu sprawdza ustawienie kolejnych 4 bitów (począwszy od bitu 0) rejestru R18 i jeżeli dany bit jest ustawiony wykonuje kod odpowiedzialny za realizację danej funkcji interfejsu. Przykładowo instrukcja “sbrs r18, 0” pominie kolejną instrukcję, jeżeli bit 0 rejestru R18 jest ustawiony. Wówczas nie zostanie wykonany skok do instrukcji znajdującej się za etykietą menuKey1:, ale zostanie wykonany kod przesuwający w lewo (instrukcja rol) zawartość rejestru R19, co ma na celu wybór kolejnego bitu do edycji. Należy zwrócić uwagę, że instrukcja rol przesuwa w lewo z wykorzystaniem znacznika przeniesienia. Osiem kolejnych stanów rejestru R19 (od reprezentacji bitowej 00000001 do 10000000) wyraźnie wskazuje na bit, który ma być edytowany, natomiast po 8 przesunięciach stan rejestru R19 będzie wynosił 0, ale ustawiony będzie znacznik przeniesienia. Wówczas instrukcja brcc menuKey1 [2] nie będzie miała spełnionego warunku i wykonane zostanie załadowanie wartości 0x01 do rejestru R19. Dzięki czemu w rejestrze R19 otrzymuje się tylko i wyłącznie stany w kodzie 1 z 8.
Podobnie jeżeli przycisk 1 będzie wciśnięty, wówczas zostanie wykonany kod, który odczytuje stan portu podłączonego do diod LEDs, w którym przechowywany jest edytowany bajt, negowany jest bit edytowanego bajtu wskazany zawartością rejestru R19 i ponownie już po edycji, która miała miejsce w rejestrze R0 zmodyfikowany bajt zapisywany jest z powrotem do portu.
W przypadku wciśnięcia przycisku 2 odczytany zostanie stan rejestru wejścia wyjścia PORTA do rejestru danych pamięci EEPROM, wykonany zostanie podprogram zapisujący do pamięci EEPROM, a następnie w arytmetyce modulo 4 zostanie zwiększony adres komórki pamięci,

do której ma być dokonywany zapis. Kolejne zapisy wykonywane są do komórek pamięci EEPROM o adresach 0, 1, 2, 3, 0, 1 itd. Części kodu odpowiedzialne za odczyt pamięci (R23) i zapis pamięci (R22) posiadają własne liczniki (R23 i R22). Część programu odpowiedzialna za odczyt nie wymaga już raczej komentarza.

1.11 Podprogramy obsługi przerwań (2 godziny)

Cel ćwiczenia

Celem ćwiczenia jest zdobycie umiejętności w rejestrowaniu podprogramów obsługi przerwań oraz nabycie wiedzy dotyczącej działania i konfiguracji kontrolera przerwań zewnętrznych mikrokontrolera ATmega16. Ćwiczenie umożliwia obserwację szczegółów obsługi przerwań w szczególności zmian stanów licznika programu, wskaźnika stosu oraz zawartości pamięci stosu.

Zagadnienia do przygotowania

Zaznajomić się ze sposobem obsługi przerwań w systemach mikroprocesorowych. Rozpatrzyć dwa sposoby rejestracji podprogramów obsługi przerwań, mianowicie z wykorzystaniem tablicy skoków do podprogramów oraz z wykorzystaniem tablicy adresów podprogramów. Zastanowić się nad sposobem rejestracji podprogramów obsługi przerwań w przypadku, kiedy wykorzystywana jest tablica skoków. Wykorzystując dokumentację [3] zapoznać się ogólnie systemem przerwań oraz bardziej szczegółowo z kontrolerem przerwań zewnętrznych mikrokontrolera. Spróbować ustalić jakie wartości powinny zostać zapisane do rejestrów konfiguracyjnych kontrolera przerwań, aby ustawiona została czułość na zbocze narastające dla wejścia 0 oraz czułość na zbocze opadające dla wejście 1 kontrolera przerwań zewnętrznych.

Przebieg ćwiczenia

1. Wartości słów konfiguracyjnych

   Ustalić wspólnie na tablicy z prowadzącym wartości jakie powinna zostać wpisane do rejestrów konfiguracyjnych kontrolera przerwań zewnętrznych, aby uzyskać czułość wejścia zerowego na zbocze narastające oraz czułość wejścia 1 na zbocze opadające.

2. Program wykonujący konfigurację

   Wykorzystując instrukcje operujące na pojedynczych bitach, jeżeli to możliwe, rejestrów wejścia wyjścia (SBI, CBI [2]) lub w przeciwnym przypadku rejestrów ogólnego przeznaczenia (SBR, CBR) napisać program konfigurujący kontroler przerwań zewnętrznych zgodnie z ustaleniami z poprzedniego punktu ćwiczenia. Użycie instrukcji operujących na pojedynczych bitach ma na celu zmianę tylko tych ustawień, które dotyczą kontrolera przerwań z pozostawieniem innych bitów nietkniętych. Alternatywą jest wpisywanie stałych wartości do rejestrów konfiguracyjnych, co jednak może, w niektórych przypadkach psuć ustawienia innych modułów współdzielących bity w rejestrach wejścia wyjścia z modułem aktualnie konfigurowanym.

3. Rejestracja podprogramu obsługi przerwania

   (a) Poniższy kod wprowadzić jako zawartość głównego pliku źródłowego projektu.

```
.include "m16def.inc"

jmp start
```

```
.org INT0addr
rjmp sub_ext_0

;1. dodać skok do właściwego podprogramu dla przerwania zewnętrznego 1

start:

;2. inicjalizacja stosu

;3. konfiguracja kontrolera przerwań zewnętrznych

sei

loop:
        nop
        nop
        nop
rjmp loop

sub_ext_0:
        nop
        inc r0
        nop
ret

;4. dodać podprogram obsługi przerwania 1
```

Program rejestruje podprogram (sub_ext_0) obsługi zerowego przerwania zewnętrznego poprzez umieszczenie skoku do do tego podprogramu w ściśle określonym miejscu pamięci programu. Miejsce to jest wskazane etykietą INT0addr, wyszukaną w pliku m16def.inc.

Instrukcja SEI ustawia bit I w rejestrze SREG. Bit ten odpowiedzialny jest za globalne blokowanie przerwań, kiedy jest wyzerowany. W przypadku, kiedy bit jest ustawiony przerwania dla poszczególnych modułów mogą być odblokowywane indywidualnie.

Podprogram sub_ext_0 stanowi treść przerwania. Zakończony został on niepoprawnie instrukcją ret, co przyczynia się do niewłaściwego działania programu. Problem ten zostanie poruszony w dalszej części ćwiczenia.

(b) Uzupełnić punkty 2 i 3 programu.

(c) Wykonywać program instrukcja po instrukcja aż do zapętlenia w pętli loop, a następnie korzystając z okienka "I/O View" ustawić wejście przerwania zerowego. Dokładne ustalenie bitu portu oraz nazwy portu należy wykonać z wykorzystaniem topologii układu zamieszczonej w dokumentacji [3]. Śledzić dalej program instrukcja po instrukcji zwracając uwagę na stany modyfikowanych zasobów (licznik programu, wskaźnik stosu oraz pamięć stosu).

(d) Zauważyć, że ponowne wygenerowanie zbocza narastającego na wejściu zerowego przerwania zewnętrznego nie powoduje już wywołania przerwania tak, jak poprzednio.

(e) Korzystając z dokumentacji [2] odszukać opis instrukcji ret oraz reti i na podstawie zauważonych różnic uzasadnić, dlaczego przerwanie było wykonane tylko jeden raz. Po poprawieniu programu dokonać ponownej symulacji tym razem zwracając szczególną uwagę na stan znacznika I.

(f) Wykorzystując zawartość pliku m16def.inc, który należy wyszukać w internecie odnaleźć wartość etykiety reprezentującej adres do którego zostanie przekazane sterowanie w przypadku wywołania przerwania 1.

(g) Zarejestrować podprogram obsługi pierwszego przerwania zewnętrznego, który w odróżnieniu od już istniejącego ma wykonywać dekrementację rejestru R0.

(h) Podobnie jak dla zerowego przerwania zewnętrznego dokonać symulacji działania pierwszego przerwania zewnętrznego. Numer bitu portu oraz jego nazwę ustalić podobnie jak dla zerowego przerwania zewnętrznego.

4. Priorytety przerwań

(a) Doprowadzić do sytuacji, w której program znajduje się w pętli głównej i jednocześnie zostaną wywołane oba przerwania zewnętrzne (przerwanie 0 - zbocze narastające, przerwanie 1 - zbocze opadające)

(b) Ustalić, które z przerwań zostało obsłużone jako pierwsze oraz czy równolegle zgłoszone przerwanie o niższym priorytecie zostanie obsłużone w późniejszym terminie czy też wcale.

5. Atomowe wykonanie części podprogramu obsługi przerwania

(a) Rozbudować kod podprogramu obsługi przerwania o wyższym priorytecie o kilka instrukcji NOP, w między które wpleść instrukcję SEI.

(b) Przyjąć, że instrukcje NOP znajdujące się w podprogramie przed instrukcją SEI stanowią kod, który musi się wykonać bez przerywania (atomowo), natomiast kod znajdujący się za instrukcją SEI może być już przerywany, zależności czasowe pomiędzy instrukcjami w tej części nie są istotne.

(c) Dokonać symulacji. Wygenerować przerwanie o wyższym priorytecie a następnie przerwanie o niższym priorytecie. Prześledzić program zwracając uwagę na opuszczenie podprogramu obsługi przerwania o wyższym priorytecie w celu wykonania podprogramu obsługi przerwania o niższym priorytecie. Zauważyć, że opuszczenie programu o wyższym priorytecie wykonuje się dopiero po wykonaniu instrukcji SEI.

1.12 Licznik czasomierz 0 (2 godziny)

Cel ćwiczenia

Celem ćwiczenia jest poznanie możliwości licznika czasomierza 0 mikrokontrolera ATmega16. Realizacja ćwiczenia umożliwi zapoznanie się z trybami pracy licznika, współpracą licznika z kontrolerem przerwań oraz zwiększy umiejętność pisania programów konfigurujących moduły składowe mikrokontrolera. Celem ćwiczenia jest również nabycie doświadczenia w posługiwaniu się dokumentacją mikrokontrolera.

Zagadnienia do przygotowania

Korzystając z dokumentacji [3] zapoznać się budową, działaniem i trybami pracy licznika czasomierza 0. Wyszukać różnic w działaniu licznika w poszczególnych trybach pracy oraz informacji dotyczących generowania przerwań przez ten licznik.

Przebieg ćwiczenia

1. Rejestracja podprogramów obsługi przerwań

   (a) Napisać program rejestrujący podprogramy obsługi przerwań pochodzących od licznika czasomierza 0 w przypadku przepełnienia się licznika oraz zrównania stanu licznika ze stanem rejestru z nim skojarzonym OCR0. Pisząc program należy bazować na kodzie z poprzedniego ćwiczenia. W pliku m16def.inc należy wyszukać etykiet reprezentujących adresy, pod które jest przekazywane sterowanie w momencie wystąpienia przepełniania lub zrównania.

   (b) Wspólnie na tablicy z prowadzącym napisać kod konfigurujący licznik w trybie normalnym. Do rejestru skojarzonego z licznikiem należy wpisać wartość różną od zera, np równą 0x0F, dzięki czemu możliwe będzie rozróżnianie przepełnienia i zrównania. Preskaler skonfigurować w taki sposób aby częstotliwość taktowania licznika była maksymalna, co ułatwi późniejsze śledzenie programu na modelu.

   (c) Dokonać symulacji programu zwracając szczególną uwagę na wywołania podprogramów podczas przepełniania się licznika oraz osiągania przez licznik stanu rejestru.

   (d) Aktualnie program działa w ten sposób, że w momencie kiedy licznik osiągnie wartość zapisaną w rejestrze (0x0F) zlicza dalej w tym samym tempie i moment przepełnienia osiąga po stosunkowo długim czasie (0xFF-0x0F+1 cykli zegara) od pierwszego zdarzenia. Zmodyfikować program w taki sposób, aby moment przepełniania się licznika następował po znacznie mniejszej liczbie cykli zegara aniżeli obecnie.

2. Generowanie przebiegu o określonej częstotliwości

   (a) Skonfigurować licznik do pracy w trybie CTC [3].

   (b) Wartość podziału częstotliwości dokonywanego przez preskaler oraz wartość wpisaną do rejestru skojarzonego z licznikiem dobrać na takim poziomie aby częstotliwość występowania zdarzenia polegającego na zrównaniu stanu licznika ze stanem rejestru była częstotliwością akustyczną (np. 1 kHz).

   (c) Sposób zachowania się wyjścia OC0 ustawić na przełączanie podczas osiągania przez licznik wartości zapisanej w rejestrze. Bit portu wyjścia OC0 należy ustawić jako wyjściowy poprzez ustawienie odpowiedniego bitu kierunkowego (DDxn).

   (d) Zaprogramować układ. Z wykorzystaniem przewodu zestaw skonfigurować w taki sposób, aby sygnał z wyjścia OC0 był przekazywany na wejście przetwornika elektroakustycznego (jedną z końcówek przewodu podłączyć do środkowego wyprowadzenie JP23).

   (e) Wykorzystując jeden z poprzednio napisanych programów pozwalających wykorzystywać 4 przyciski napisać program, który będzie generował sygnał o częstotliwości zależnej od wciśniętego przycisku. Częstotliwość generowanego sygnału należy regulować zawartością OCR0.

3. Regulacja średniego natężenia świecenia diody

   (a) Skonfigurować licznik do pracy w trybie FAST PWM.

   (b) Sygnał z wyjścia OC0 doprowadzić do jednej z diod LEDs.

   (c) Zmieniać wartość wpisywaną do rejestru OCR0 i obserwować natężenie świecenia diody. Zauważyć, że zmiany natężenia świecenia diody nie zależą w sposób liniowy od wypełnienia generowanego przebiegu.

(d) Wykorzystując program obsługujący klawiaturę z poprzedniego punktu napisać aplikację, która pozwala za pomocą przycisków regulować natężenie świecenia diody. Dwa przyciski mogą zostać wykorzystane do zwiększania i zmniejszania wypełnienia, trzeci przycisk może powodować gaszenie diody, a czwarty przycisk może powodować ustawianie średniej intensywności świecenia diody.

1.13 Generator dźwięków (2 godziny)

Cel ćwiczenia

Celem ćwiczenia jest zdobycie umiejętności wykorzystania modulacji PWM do generowania sygnałów dźwiękowych oraz nabycie doświadczenia w programowaniu licznika czasomierza 0 mikrokontrolera ATmega16 w trybie FAST PWM.

Zagadnienia do przygotowania

Przed przystąpieniem do ćwiczenia należy zapoznać się szczegółowo z działaniem oraz konfiguracją licznika czasomierz 0 w trybie FAST PWM. Przemyśleć problem generowania sygnału analogowego z wykorzystaniem modulatora PWM oraz jaki jest wpływ częstotliwości pracy modulatora na jakość tego sygnału (obecność nośnej w widmie akustycznym).

Przebieg ćwiczenia

1. Poniższy kod wprowadzić do głównego pliku źródłowego projektu.

```
.include "m16def.inc"

jmp start

.org OVF0addr
jmp ovf0sub

start:

ldi r16, low(ramend)
out spl, r16

ldi r16, high(ramend)
out sph, r16

sbi ddrb, 3

ldi zl, low(sound<<1)
ldi zh, high(sound<<1)

lpm r0, z+
out ocr0, r0

ldi r16,0x69
```

```
out tccr0, r16

in r16, timsk
ori r16, 1
out timsk, r16

sei

ldi r16, 8

main:

rjmp main

ovf0sub:

dec r16
breq setr16newval
reti

setr16newval:

ldi r16, 8

lpm r0, z+
out ocr0, r0

cpi zl, low(sound_end<<1)
brne ovf0syb_end
cpi zh, high(sound_end<<1)
brne ovf0syb_end

ldi zl, low(sound<<1)
ldi zh, high(sound<<1)

ovf0syb_end:
reti

sound: .db .include "sound.txt"
sound_end:
```

2. Z lokalizacji http://www.fizyka.umk.pl/~daras/avr/src/ ściągnąć pliki sound.txt oraz sound1.txt i umieścić je w katalogu projektu. Dokonać assemblacji celem stwierdzenie poprawności kodu źródłowego oraz właściwego ściągnięcia i umieszczenia plików *.txt.

3. Analiza działania programu

(a) Na samym początku zwrócić uwagę na dwie ostatnie linie kodu oraz zawartości plików *.txt. Dyrektywa .include włącza wskazany plik w miejscu wywołania. W efekcie uzyskuje się kod definiujący zawartość pamięci FLASH na podstawie zawartości pliku sound.txt.

(b) W liniach 18 i 19 pobierany jest do rejestru indeksowego Z adres pamięci programu, od którego została umieszczona próbka dźwięku.

(c) W linii 33 określany jest stan rejestru R16, pełniącego rolę licznika wywołań podprogramu obsługi przerwania. Stan tego licznika jest dekrementowany (linia 41) co każde wywołanie podprogramu obsługi przerwania generowanego w momencie przepełniania się licznika. W przypadku, kiedy stan tego rejestru osiągnie 0, ponownie załadowywana jest do niego wartość początkowa (8), do rejestru OCR0 ładowana jest wartość kolejnego bajtu z próbki dźwięku i cykl dekrementacji licznika zaczyna się od nowa.

Powyższy zabieg jest konieczny ze względu na zwiększenie częstotliwości repetycji sygnału PWM (zwiększenie częstotliwości sygnału nośnego) przy zachowaniu stosunkowo małej częstotliwości wystawiania kolejnych bajtów próbki dźwięku. Stosunkowo mała częstotliwość wystawiania kolejnych bajtów jest wymagana ze względu na małą ilość pamięci programu, w której zapisane są kolejne bajty próbki dźwięku. Przyjęto, że próbka dźwięku (plik sound.txt) posiada częstotliwość próbkowania 3906.25 Hz = 8 MHz/(256*8), gdzie czynnik 256 bierze się z ilości stanów, które posiada licznik w cyklu od przepełnienia do przepełnienia, a czynnik 8 pochodzi od wspomnianego w tym podpunkcie zabiegu zwiększania liczby okresów modulatora dla pojedynczego bajtu próbki dźwięku. Z tego samego powodu pojedyncze elementy próbki dźwięku są 8-mio a nie 16-to bitowe oraz próbka jest monofoniczna.

W przypadku, kiedy częstotliwość pracy modulatora byłaby równa częstotliwości próbkowania, wówczas oprócz pożądanego dźwięku próbki pojawiałby się bardzo wyraźnie dźwięk częstotliwości nośnej (około 3.9 kHz), który byłby bardzo uciążliwy. Po ośmiokrotnym zwiększeniu częstotliwości pracy modulatora częstotliwość fali nośnej leży poza zakresem słyszalnym i albo nie jest odbierana przez ucho, albo jest tłumiona we wzmacniaczu sygnału akustycznego.

(d) W liniach od 52 do 55 znajdują się instrukcje odpowiedzialne za sprawdzanie, czy ostatni bajt z próbki dźwięku został już wykorzytany. W przypadku, kiedy obie instrukcje warunkowe z tych linii będą miały warunek niespełniony, czyli zawartość rejestru Z będzie równa wartości etykiety sound_end, wówczas adres początkowy zostanie załadowany do rejestru Z ponownie i odtwarzanie próbki dźwięku rozpocznie się od początku.

(e) Przeanalizować dokładnie powyższy program. Wyszukać linii kodu odpowiedzialnych za konfigurację licznika czasomierza 0. Określić korzystając z dokumentacji [3] tryb pracy oraz parametry czasowe. W przypadku braku zrozumienia działania instrukcji lub fragmentów kodu szukać odpowiedzi w dokumencie [2] lub pytać prowadzącego.

4. Zaprogramować mikrokontroler. Odsłuchać próbkę dźwięku z wykorzystaniem albo zamieszczonego w zestawie uruchomieniowym przetwornika elektroakustycznego, albo systemu głośników monitora komputerowego. W celu wykorzystania głośników monitora należy zwrócić uwagę, że masa monitora i masa zestawu są podłączone przez komputer i następnie przez programator, dzięki czemu wymagane jest podłączenie wyłącznie sygnału lewego lub prawego kanału do wyjścia OC0.

5. Zmienić przedostatnią linię kodu tak, aby włączać plik o nazwie sound1.txt. Dokonać assemblacji i programowania. Odtwarzana próbka dźwięku okaże się nieczytelna, ponieważ poszczególne jej elementy zapisane są w odwrotnej kolejności. Dokonać takiej modyfikacji kodu, aby odtwarzanie próbki odbywało się w naturalnej dla niej kolejności (czyli od końca). Program wymaga modyfikacji w miejscach

ładowania adresu początkowego, modyfikowania wskaźnika Z oraz sprawdzania warunku końca. Zwrócić uwagę na fakt, że mikrokontroler nie posiada instrukcji LPM z opcją dekrementacji i należy użyć osobnej instrukcji zmniejszającej stan rejestru Z [2].

6. Zmodyfikować program w taki sposób, aby okres modulatora był równy częstotliwości wystawiania próbek. W tym celu należy usunąć realizowane z wykorzystaniem rejestru R16 zapętlenie oraz zmienić stopień podziału częstotliwości dokonywany przez preskaler na 8. Po zaprogramowaniu układu zwrócić uwagę na bardzo dobrze słyszalną częstotliwość nośną.

1.14 Zaawansowany Assembler (2 godziny)

Cel ćwiczenia

Celem ćwiczenia jest zapoznanie z językiem Assemblera na poziomie średnio zaawansowanym. Ćwiczenie umożliwia poznanie od strony praktycznej działania większości najczęściej wykorzystywanych elementów języka Assemblera oraz zauważenie różnic pomiędzy instrukcjami mikrokontrolera a językiem Assemblera.

Zagadnienia do przygotowania

Przed przystąpieniem do ćwiczenia należy powtórzyć działanie następujących makr Assemblera .include, .equ, .dseg, .cseg, .eseg, .db, .byte i .org, funkcji low() i high() i operatorów <<, -, =. Należy również przemyśleć różnicę pomiędzy instrukcjami mikrokontrolera a samym Assemblerem.

Przebieg ćwiczenia

1. Dokonać analizy, assemlacji oraz sprawdzenia działania poniższego programu, który składa się z głównego pliku o zawartości zamieszczonej poniżej

```
.include "m16def.inc"
.include "macros.asm"

.equ num=pmem_end-pmem<<1
;.equ num=4

.set dest=dmem1

gPMA z, pmem
gDMA x, dest


ldi r20, num
loop:
cpPD z, x
dec r20
brne loop


stop:
rjmp stop
```

```
pmem:
.db 0xaa, 0xbb, 0xcc, 0xdd, 0x44, 0x55
pmem_end:

.dseg
.org 0x60
dmem:
.byte num

.set dest=dmem1
.dseg
.org 0x80
dmem1:
.byte num
```

oraz pliku o nazwie “macros.asm”, który powinien znaleźć się w katalogu projektu, którego zawartość jest następująca.

```
.set dest=sram_start

.macro gPMA
ldi @0l, low(@1<<1)
ldi @0h, high(@1<<1)
.endmacro


.macro gDMA
ldi @0l, low(@1)
ldi @0h, high(@1)
.endmacro


.macro cpPD
.def tmp=r16
lpm tmp, @0+
st @1+, tmp
.endmacro

;ldi tmp, 0 ; brak zasiegu
```

Druga linia głównego programu włącza do projektu zawartość pliku “macros.asm”, dzięki czemu zdefiniowane w tym pliku makra będą widoczne i możliwe do użycia na poziomie głównego programu.

Dyrektywa .equ służy do przypisywania nazwie wyrażenia. Raz przypisane nazwie wyrażenie nie może być później zmieniane. Podobnie działa dyrektywa .set z tą różnicą, że w dalszej części kodu możliwa jest zmiana przypisanego wcześniej danej nazwie wyrażenie. Dyrektywa .def jest używana do przypisywania nazw rejestrom, co ma ułatwić ich używanie, ponieważ funkcja jaką rejestr będzie pełnił w programie

może być ukryta w nazwie. Dany rejestr może mieć przypisanych jednocześnie kilka nazw, które mogą być modyfikowane (kasowane) poprzez przypisanie ich innym rejestrom.

Dyrektywa .makro definiuje makro o nazwie podanej jako pierwszy argument. Kod zawarty w nawiasach .macro .endmacro stanowi treść, która będzie wstawiana w miejscu wywołania makra. Makro wywoływane jest poprzez podanie w kodzie jego nazwy. Makra mogą posiadać argumenty, które dostępne są za pomocą operatora @n, gdzie n jest numerem argumentu.

W linii 9 wywoływane jest makro o nazwie gPMA z dwoma argumentami. Argument zerowy informuje o rejestrze indeksowym, do którego należy wpisać adres przekazany jako etykieta w argumencie pierwszym. Akronim gPMA (ang. get Program Memory Address) powstał od angielskiego tłumaczenie pobierz adres w pamięci programu. Należy zauważyć, że przed wpisaniem do rejestru docelowego wartość etykiety jest mnożona przez 2 w tym makrze.

Podobnie działa makro gDMA (ang. get Data Memory Address) służące do pobierania do wskazanego zerowym argumentem rejestru wartości etykiety podanej argumentem pierwszym.

Makro cpPD (ang. copy from Program to Data Memory) kopiuje zawartość komórki pamięci programu o adresie znajdującym się w rejestrze indeksowym, którego nazwa przekazywana jest jako argument zerowy makra do komórki pamięci danych o adresie zawartym w rejestrze indeksowym wskazanym pierwszym argumentem makra. W makrze tym dla wygody rejestrowi R16 pełniącemu rolę rejestru tymczasowego nadano adekwatną nazwę tmp. Nazwa ta nie jest widziana poza makrem - w przypadku odkomentowania ostatniej linii pliku zawierającej makra zgłoszony zostałby brak zasięgu dla nazwy tmp.

Program jako całość kopiuje zawartość wybranego obszaru pamięci programu do wybranego obszaru pamięci danych. Liczba kopiowanych bajtów określana jest poprzez nazwę num, która może być albo określana ręcznie (5 linia programu głównego), albo automatycznie (4 linia). Jednoczesne odblokowanie 4 i 5 linii powodowałoby błąd assemblacji. Adres docelowy dest jest pierwotnie określany na początek pamięci SRAM (sram_start) w 1 linii pliku włączanego, ale później w 7 linii głównego pliku projektu jest zmieniany na wartość określaną przez etykietę dmem1. Wartość etykiety dmem1 została celowo zmieniona z domyślnej jaka by tu została przypisana (0x60) na wartość 0x80.

Należy pamiętać o tym, że wywoływanie makrodefinicji powoduje wstawianie kodu przez nie reprezentowanego i w celu właściwego śledzenia kodu źródłowego zawierającego makra należy po uruchomieniu debuggera wybrać z menu View polecenie Disassembler, które otworzy plik zawierający kompletny program.

2. Zastanowić się nad różnicą pomiędzy wykorzytaniem makr a wykorzystaniem podprogramów.

3. Napisać makro rejestrujące podprogram obsługi przerwania. Argumentami wejściowymi makra powinny być adres podprogramu oraz numer przerwania, dla którego rejestrowany jest podprogram.

   Sprawdzanie poprawności działania makra powinno polegać na analizie zawartości pamięci programu zwracanej przez polecenie Disassembler dostępne w menu View. Nie jest konieczna analiza dziania programu na modelu.

2 Podstawy programowania w języku C

2.1 Wstęp (1 godzina)

Cel ćwiczenia

Celem ćwiczenia jest wprowadzenie do programowania mikrokontrolera Atmel AVR ATmega16 w języku C z wykorzystaniem środowiska Atmel AVR Studio oraz pakietu WinAVR zawierającego bibliotekę avr-libc. W trakcie ćwiczenia zostaną przedstawione podstawowe elementy biblioteki avr-libc oraz możliwe będzie nabycie doświadczenia w pisaniu programów w języku C działającym na mikrokontrolerach AVR.

Zagadnienia do przygotowania

Powtórzyć podstawowe elementy języka C, w szczególności instrukcje warunkowe: if, while, typy zmiennych: char, short int, unsigned short int, unsigned char itp. oraz operatory arytmetyczne i bitowe wraz z ich priorytetami.

Przebieg ćwiczenia

1. Uruchomić AVR Studio. Jako typ projektu wybrać "AVR GCC". Pozostałe ustawienia projektu wybrać takie, jakie były wybierane w projektach assemblerowych. Do pliku głównego projektu wprowadzić następujący kod.

```
#include <avr/io.h>
main(void)
{
        DDRA=0xff;
        DDRB=0x00;
        PORTB=0xff;

        while(1)
        {
                PORTA=PINB;
        }
        return 0;
}
```

   Jak łatwo ustalić program odczytuje stan wyprowadzeń portu PB i przekazuje je na port PA. Port PB został skonfigurowany jako wejściowy z polaryzacją i można bezpośrednio do niego podłączać przyciski, port PA został skonfigurowany jako wyjściowy i należy podłączyć do niego diody.

2. Zmienić program pętli głównej na następujący.

```
PORTA++;
_delay_ms(1000);
```

   Aby prototyp funkcji _delay_ms() był widoczny w trakcie kompilacji należy do programu należy włączyć plik nagłówkowy "util/delay.h".

3. Dokonać kompilacji i zaprogramować mikrokontroler. Zauważyć, że kolejne stany na diodach pojawiają się co czas bardzo różniący się od 1 s. Spowodowane jest to dwoma czynnikami. Pierwszym z nich

jest fakt, że do kompilatora nie została przekazana informacja o częstotliwości pracy układu i nie może on określić ilości potrzebnych iteracji pętli opóźniających w celu uzyskania żądanego opóźnienia. Drugim czynnikiem jest (domyślnie) włączona optymalizacja pod względem prędkości -Os, która zastępuje napotkany kod krótszym czasowo, jeżeli takowy istnieje, dającym ten sam efekt liczbowy. Uruchomić polecenie "Configuration" w menu "Project" i w zakładce "General" w polu "Frequency" wpisać "8000000", a w polu "Optimization" ustawić "-O2". dokonać kolejnej kompilacji i programowania. Skomentować różnice w działaniu.

4. Napisać program, który inkrementuje stan wybranego portu, w przypadku, kiedy przycisk jest wciśnięty, a dekrementuje stan tego portu w przypadku, kiedy przycisk jest wyciśnięty. Zestaw uruchomieniowy skonfigurować tak, aby stan wybranego portu był przekazywany na diody LEDs. Modyfikacja stanu wybranego portu powinna następować co czas równy 1 s.

5. Przetestować działanie instrukcji logicznych do sterowania pojedynczymi bitami. Napisać sekwencję, która ustawia, zeruje lub neguje wybrane bity w zmiennej. Wykorzystać operatory bitowe suma logiczna "|", iloczyn logiczny "&" oraz operator sumy modulo "^".

6. Napisać program, który kopiuje trzy najmłodsze bity portu PA na miejsce trzech najstarszych bitów portu PB nie modyfikując przy tej operacji pozostałych bitów portu PB.

2.2 Prosty interfejs użytkownika (2 godziny)

Cel ćwiczenia

Celem ćwiczenia jest utrwalenie wiadomości z poprzedniego ćwiczenia oraz nabycie umiejętności pisania prostych programów w języku C działających na mikrokontrolerach.

Zagadnienia do przygotowania

Powtórzyć i zastanowić się nad nowymi elementami materiału z poprzedniego ćwiczeniu - w przypadku konieczności posiłkować się dokumentacją [6]. Powtórzyć działanie algorytmu eliminującego wpływ drgań styków, który był omawiany i implementowany w jednym z ćwiczeń dotyczących programowania w Assemblerze.

Przebieg ćwiczenia

1. Napisać uruchomić i przetestować program w języku C realizujący algorytm eliminacji wpływu drgań styków.

2. Program rozbudować tak, aby obsługiwał 4 przyciski.

3. Do powstałego programu dodać funkcję edytora bajtu.

   Przycisk 0 powinien powodować wybór poprzedniego, a przycisk 1 wybór kolejnego bitu. Przesuwanie bitów w języku C może być realizowane z wykorzystaniem operatora << (w lewo) oraz operatora >> (w prawo). Negowanie, ustawianie lub zerowanie wybranego bitu edytowanego bajtu może być realizowane z wykorzystaniem odpowiednio operatorów ^ (sumy modulo), | sumy logicznej (bitowej - język C) lub & iloczynu logicznego.

   Wykonywanie operacji na bicie należy dokonywać z wykorzystaniem przycisku 2. Przycisk 3 powinien zostać zaprogramowany do wyboru rodzaju operacji. Każde jego wciśnięcie powinno wybierać

kolejną operację, wśród nich powinny się znaleźć wyżej już wymienione, czyli negowanie, ustawianie lub zerowanie.

2.3 Sterowanie multipleksowe (2 godziny)

Cel ćwiczenia

Celem ćwiczenia jest nabycie umiejętności oraz doświadczenia w pisaniu programów sterujących pracą elementów wykonawczych w sposób multipleksowy. Realizacja ćwiczenia umożliwia ugruntowanie wiedzy na temat sterowania multipleksowego oraz zaznajomienie się z obsługą wyświetlaczy 7-mio segmentowych w trybie multipleksowym.

Zagadnienia do przygotowania

Zaznajomić się z ideą sterowania multipleksowego na poziomie umożliwiającym podanie przebiegów sygnałów dostarczanych do zespołu wyświetlaczy w celu wygenerowania na nich określonej sekwencji. Zastanowić się jak wpływa częstotliwość przełączania aktywności poszczególnych wyświetlaczy na komfort wzrokowy oraz czy ilość wyświetlaczy może ograniczać tą częstotliwość.

Przebieg ćwiczenia

1. Napisać kod generujący w sposób modulo kolejne wartości w kodzie 1 z 4. Należy użyć operatora <<, wartością przesuwaną powinna być 1, natomiast zmienna określająca ilość bitów przesunięcia powinna się zmieniać w zakresie od 0 do 3.

2. Utworzyć tablicę 4-ro elementową, która będzie służyła do przechowywania wartości poszczególnych cyfr wyświetlanej liczby. Wartości poszczególnych cyfr można pozyskiwać z wykorzystaniem operatora zwracającego resztę z dzielenia. Przykładowo wartość jedności liczby x można poznać po wykonaniu operacji x%10, a wartość dziesiątek tej liczby można poznać dzięki operacji x/10%10 itd.

3. Utworzyć tablicę 10-cio elementową, której zadaniem będzie przechowywanie kolejnych kodów wyświetlacza 7-mio segmentowego. Indeksowanie tej tablicy pozwala na uzyskiwanie szybkiej konwersji kodu NKB na kod wyświetlacza 7-mio segmentowego.

4. Korzystając z dokumentacji [5] sprawdzić możliwości podłączania wyświetlaczy do mikrokontrolera. Wybrać opcję, w której segmenty podłączone są do portu PB, a wybór wyświetlacza dokonywany jest młodszymi bitami portu PC. Zwrócić uwagę na to, że bit 0 portu PB podłączony jest do segmentu "a" wyświetlacza, bit 1 do segmentu "b" itd. aż do bitu 7, który steruje znakiem dziesiętnym "dp". Zwrócić uwagę również na fakt, że bit zerowy portu PC aktywuje wyświetlacz tysięcy (całkowicie lewy), bit pierwszy aktywuje wyświetlacz setek itd.

5. Napisać kod uzupełniający tablicę cyfr. Jako liczbę źródłową wybrać np. wartość 1234.

6. Zapełnić tablicę kodów wyświetlacza 7-mio segmentowego. Korzystając z informacji pozyskanych w punkcie 4 ćwiczenia określić wartości poszczególnych bitów reprezentujących cyfry na wyświetlaczu 7-mio segmentowym. Przykładowo reprezentacja wartości 0 na wyświetlaczu 7-mio segmentowym wynosi 0x3F, a np. reprezentacja wartości 7 wynosi 0x07.

7. Napisać program, który cyklicznie w pętli wysyła na zespół wyświetlaczy kolejne 7-mio segmentowe reprezentacje cyfr znajdujących się w tablicy cyfr. Wystawianie kolejnych cyfr powinno odbywać się

w takt zmian stanu sygnału sterującego aktywnością wyświetlaczy. Należy pamiętać o tym, aby przez czas przeprowadzania zmiany cyfry żaden wyświetlacz nie był aktywny.

8. Zmodyfikować program tak, aby kolejne wyświetlacze były aktywowane co czas równy 1s, co umożliwi obserwację idei sterowania multipleksowego. Następnie doprowadzić do sytuacji, kiedy kolejne wyświetlacze są aktywowane co czas równy około 1 ms.

9. Rozbudować program o funkcję inkrementacji zmiennej, której wartość jest wyświetlana. W przypadku, kiedy zmienna jest inkrementowana w każdej iteracji, wówczas wyświetlana wartość będzie w przybliżeniu zliczała czas w jednostkach 1 ms. Oznacza to, że wykonany licznik będzie posiadał zakres 10 s.

10. Dodać opcję startu i stopu do wykonanego programu. Naciśnięcie przycisku start powinno powodować uruchomienie zliczania natomiast naciśnięcie przycisku stop powinno zatrzymywać zliczanie.

11. Dodać opcję resetu oraz zmiany kierunku zliczania.

2.4 Prosty kalkulator (2 godziny)

Cel ćwiczenia

Celem ćwiczenia jest nabycie doświadczenia w realizacji średnio zaawansowanych projektów, w których jednocześnie wykorzystuje się podprogramy obsługi przerwań jak i kod pętli głównej. Realizacja ćwiczenia pozwala również na ugruntowanie wiedzy dotyczącej sterowania multipleksowego oraz zdobycie umiejętności obsługi klawiatury macierzowej.

Zagadnienia do przygotowania

Przed przystąpieniem do ćwiczenia należy zapoznać się z ideą sterowania multipleksowego oraz zasadą działania i obsługi klawiatury macierzowej. Zaznajomić się z programem poprzez analizę kodu oraz przeczytanie zamieszczonego pod programem komentarza. Wykorzystując dokumentację [6] wyszukać informacji dotyczących rejestrowania podprogramów obsługi przerwań, wytwarzania opóźnień oraz wykorzystywania pamięci FLASH na potrzeby przechowywania danych.

Przebieg ćwiczenia

1. Poniższy program wprowadzić jako zawartość pliku głównego projektu.

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <avr/pgmspace.h>

unsigned char lcd_digits[10] __attribute__((progmem)) =
{0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x6f};

unsigned char key;
unsigned char data[4]={0,0,0,0};

void actual_data(unsigned int ad)
```

```
{
        data[3]=ad%10;
        data[2]=(ad/10)%10;
        data[1]=(ad/100)%10;
        data[0]=(ad/1000)%10;
}

ISR(TIMER0_COMP_vect)
{
        static unsigned char num=0;

        PORTC=PORTC&0xf0|1<<num;

        PORTB=pgm_read_byte(lcd_digits+data[num]);

        num++;
        num&=3;
}

void keyboard(void)
{
        static unsigned char row=0;
        static unsigned char p=0;
        char k;

        row++;
        row&=3;

        cli();
        PORTC=(PORTC|0xf0)&~(1<<row+4);
        sei();

        k=PINA|0x0f;

        if(k==0xff)
        {
                p&=~(1<<row);
                return;
        }

        if(p!=0)
                return;

        _delay_ms(10);
```

```
        if(k!=(PINA|0x0f))
                return;

        p=0x0f;
        key=row|(k==0xef?0:k==0xdf?1:k==0xbf?2:3)<<2|1<<4;
}

void menu(void)
{
        if(key==0)
                return;

        static unsigned int suma=0;
        suma+=key&0x0f;
        actual_data(suma);

        key=0;
}

int main(void)
{
        DDRB=0xff;
        DDRC=0xff;
        PORTA|=0xf0;

        TCCR0=0x0c;
        OCR0=0x40;
        TIMSK|=1<<OCIE0;

        SREG|=1<<SREG_I;

        while(1)
        {
                keyboard();
                menu();
        }

        return 0;
}
```

W liniach 6 i 7 znajduje się definicja zawartości tablicy zawierającej reprezentacje cyfr dziesiętnych na wyświetlaczu 7-mio segmentowym. Użycie atrybutu “progmem” powoduje, że tablica ta zostanie umieszczona w pamięci programu, dzięki czemu nie będzie zajmować bardziej cennych zasobów jakimi jest pamięć SRAM z uwagi na jej 16 krotnie mniejszą wielkość oraz większą uniwersalność.

Zmienna key przechowuje kod wciśniętego przycisku, natomiast zmienna tablicowa data przechowuje wartości poszczególnych cyfr.

Funkcja actual_data() dokonuje rozdziału argumentu na wartości poszczególnych cyfr. Element 0 tablicy przechowuje informację o liczbie tysięcy w liczbie, a element 3 informację o liczbie jedności w liczbie.

Makro ISR wywołane z argumentem TIMER0_COMP_vect rejestruje podprogram obsługi przerwania pochodzącego z licznika czasomierza 0 generowanego w momencie osiągnięcia przez licznik wartości zapisanej w rejestrze z nim skojarzonym. Zadaniem tego podprogramu jest obsługa wyświetlaczy 7-mio segmentowych na najniższym poziomie. Jak łatwo ustalić z kodu konfigurującego licznik pracuje w trybie CTC z częstotliwością równą około 500 Hz. W linii 24 wystawiane są na młodsze bity portu PC kolejne wartości w kodzie 1 z 4 (wybór wyświetlacza). Polecenie to zostało napisane w taki sposób, aby stany starszych bitów nie były modyfikowane, z uwagi na wykorzystanie ich do obsługi klawiatury. Zmienna num jest zmienną klasy static, dzięki czemu pamięć na jej przechowywanie przydzielana jest tylko raz i zmienna zachowuje wartość przy ponownym wywołaniu bloku programu. Przypisanie num=0 również jest wykonywane tylko raz. Zaletą zmiennych tej klasy w porównaniu do zmiennych globalnych jest to, że te pierwsze nie zaśmiecają przestrzeni nazw, dzięki czemu nazwy mogą być krótkie (bez przedrostków identyfikujących funkcję lub inny blok). W linii 26 w zależności od aktualnie obsługiwanego wyświetlacza (num) oraz jego stanu (data[num]) na port B wystawiana jest definicja wyglądu cyfry na wyświetlaczu 7-mio segmentowym. Zmienna num przyjmuje cyklicznie wartości w zakresie od 0 do 3 dzięki dwóm ostatnim liniom podprogramu obsługi przerwania.

Funkcja keyboard() odpowiedzialna jest za obsługę 16-tu przycisków klawiatury macierzowej zawierającej 4 wiersze i 4 kolumny. Linia 9 tego podprogramu wystawia kolejne wartości w kodzie 0 z 4 na 4 starsze bity portu PC w taki sposób, że 4 młodsze bity tego portu (wybór wyświetlacza) nie są modyfikowane. Przerwanie wywołujące obsługę wyświetlacza może być wywoływane asynchronicznie w stosunku do kodu pętli głównej. Z uwagi na ten fakt oraz na to, że oba programy współdzielą port PC modyfikacja wybranych bitów portu PC w funkcji keyboard nie może być przerywana podprogramem obsługi przerwania. Aby osiągnąć wykonywanie kodu z linii 9 w sposób nieprzerywany podprogramem obsługi przerwania należy na czas wykonywania tego kodu przerwania zablokować (cli()), a po wykonaniu tego kodu przerwania odblokować (sei()).

Do 4 starszych bitów zmiennej k odczytywane są wartości sygnałów wyjściowych klawiatury macierzowej, natomiast 4 młodsze bity tej 8-mio bitowej zmiennej otrzymują stan wysoki. W przypadku, jeżeli żaden przycisk nie został wciśnięty, wówczas warunek z linii 14 będzie niespełniony. Aby stwierdzić, że żaden z przycisków nie został wciśnięty potrzebne jest niespełnianie tego warunku przez 4 kolejne wywołania tej funkcji. Jeżeli żaden przycisk nie był wciśnięty, wówczas zmienna p przyjmie wartość 0 - pełni ona rolę analogiczną do znacznika T z programu assemblerowego obsługi klawiatury 4-ro przyciskowej. Jeżeli natomiast przycisk został wciśnięty, ale przedtem były wszystkie wyciśnięte (p=0), wówczas określony zostanie stan zmiennej key na podstawie wartości odczytanej ze zmiennej k oraz aktualnego wiersza (fazy obsługi klawiatury) row.

Zadaniem funkcji menu jest dodawanie wartości zależnej od kodu wciśniętego przycisku do zmiennej suma i zlecanie jej wyświetlania. Należy zauważyć, że kod wciśniętego przycisku stanowi numer przycisku zwiększony o wartość 0x10 (1<<4). Po wykonaniu akcji dla danego przycisku do zmiennej key wpisywana jest wartość 0. Aby rozróżnić brak wciśnięcia jakiegokolwiek przycisku od wciśnięcia przycisku numer 0 do każdego kodu przycisku dodaje się właśnie wartość 0x10, która dzięki operacji &0x0f jest odfiltrowywana w funkcji menu. Dzięki wyżej wspomnianemu działaniu funkcji menu możliwe jest stwierdzenie poprawności działania obsługi klawiatury macierzowej, ponieważ każde wciśnięcie przycisku (oprócz przycisku 0) powoduje inkrementację wyświetlanej wartości o indeks przycisku. W przypadku, kiedy eliminacja wpływu drgań styków nie działałaby poprawnie, wówczas na podstawie

błędnego przyrostu wyświetlanych wartości dałoby się to szybko zauważyć.

2. Na podstawie wniosków z analizy powyższego kodu dokonać ustawień zestawu laboratoryjnego tak, aby wykorzystywane peryferia (wyświetlacze oraz klawiatura) zostały właściwie podłączone [5]. Należy pamiętać o tym aby zworka konfigurująca klawiaturę znajdowała się w położeniu 4x4 [5] złącza JP26. Dokonać implementacji oraz sprawdzić czy działanie programu jest zgodne z oczekiwaniami.

3. Zakomentować funkcję blokującą przerwania (cli()). Dokonać obserwacji działania programu. Wysunąć wnioski.

4. Bazując na powyższym programie napisać aplikację prostego kalkulatora. Przyciski o indeksach od 0 do 9 mogą reprezentować kolejne cyfry, natomiast pozostałe przyciski działania arytmetyczne +, -, *, / oraz operatory wykonania operacji = i zerowania.

2.5 Przetwornik analogowo cyfrowy (3 godziny)

Cel ćwiczenia

Celem ćwiczenia jest zdobycie wiedzy dotyczącej działania oraz sposobu wykorzystania przetwornika analogowo cyfrowego mikrokontrolera ATmega16 jak również doświadczenia w rozbudowywaniu wielowątkowych aplikacji. Ćwiczenie umożliwia udoskonalenie umiejętności programowania mikrokontrolera AVR w języku C z wykorzystaniem biblioteki avr-libc.

Zagadnienia do przygotowania

Zapoznać się z działaniem oraz trybami pracy przetwornika analogowo cyfrowego mikrokontrolera AVR [3]. O ile to konieczne zapoznać się z ideą sterowania multipleksowego wyświetlaczy 7-mio segmentowych. Wykorzystując dokumentację [6] wyszukać informacji dotyczących rejestrowania podprogramów obsługi przerwań, wytwarzania opóźnień oraz wykorzystywania pamięci FLASH na potrzeby przechowywania danych.

Przebieg ćwiczenia

1. Dokonać analizy poniższego kodu.

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <avr/pgmspace.h>

unsigned char lcd_digits[10] __attribute__((progmem)) =
{0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x6f};

unsigned char key;
unsigned char data[4]={2,0,0,8};

unsigned mode=0;

void actual_data(unsigned int ad)
{
        data[3]=ad%10;
```

```
        data[2]=(ad/10)%10;
        data[1]=(ad/100)%10;
        data[0]=(ad/1000)%10;
}

ISR(ADC_vect)
{
        if(mode==0)
                actual_data(ADC);
}

ISR(TIMER0_COMP_vect)
{
        static unsigned char num=0;

        PORTC=(PORTC&0xf0)|1<<num;

        PORTB=pgm_read_byte(lcd_digits+data[num]);

        num++;
        num&=3;
}

void keyboard(void)
{
        char k;
        static unsigned char p;

        k=(PINC>>4)|0xf0;

        if(k==0xff)
        {
                p=0;
                return;
        }

        if(p==1)
                return;

        _delay_ms(10);

        if(k!=((PINC>>4)|0xf0))
                return;

        p=1;
```

```
        key=~k;
}

void menu(void)
{
        static unsigned int wynik=0;

        switch(key)
        {
                case 1:
                        if(mode==0)
                                mode=1;
                        else
                                mode=0;
                break;
                case 2:
                        wynik+=ADC;
                break;
                case 4:
                        wynik-=ADC;
                break;
                case 8:
                        wynik=0;
                break;
        }

        key=0;

        if(mode==1)
                actual_data(wynik);
}

int main(void)
{
        DDRB=0xff;
        DDRC=0x0f;
        PORTC|=0xf0;

        TCCR0=0x0c;
        OCR0=0x40;
        TIMSK|=1<<OCIE0;
        SREG|=1<<SREG_I;

        ADMUX=0x40;
        ADCSRA=0xef;
```

```
        while(1)
        {
                keyboard();
                menu();
        }

        return 0;
}
```

Program wykorzystuje w dużej mierze te same mechanizmy, co program z poprzedniego ćwiczenia. Bieżący opis zawiera tylko nowe, niewyjaśniane do tej pory elementy. W przypadku konieczności zapoznać się z opisem z poprzedniego ćwiczenia.

Makro ISR wywołane z parametrem ADC_vect rejestruje podprogram obsługi przerwania dla przetwornika analogowo cyfrowego generowanego w momencie zakończenia przetwarzania [6]. W przypadku, kiedy zmienna mode określająca tryb pracy jest równa 0, wówczas zespół wyświetlaczy będzie pokazywał wartość odczytaną z przetwornika, co będzie miewało miejsce podczas wywołania tego przerwania.

W przypadki jednak, kiedy wartość zmiennej mode wynosi 1, wówczas stan wyświetlacza określany jest przez wartość zmiennej wynik. Wartość zmiennej wynik modyfikowana jest w podprogramie menu (na podstawie stanu przycisków), w którym również modyfikowany jest stan zmiennej mode.

Naciśnięcie przycisku zerowego powoduje zmianę stanu zmiennej mode. Naciśnięcie przycisku pierwszego zwiększa wartość zmiennej wynik o wartość odczytaną z przetwornika, natomiast naciśnięcie przycisku drugiego zmniejsza tą wartość o wartość odczytaną z przetwornika, przycisk trzeci służy do zerowania zmiennej wynik.

2. Na podstawie dokonanej analizy określić sposób konfiguracji zestawu uruchomieniowego. Dokonać sprawdzenia działania powyższego kodu. Zwrócić uwagę na różne zachowanie programu w zależności od trybu pracy (zmienna mode).

3. Rozbudować program w taki sposób aby system pełnił rolę prostego syntezatora z pamięcią. Przycisk 0 służyłby wówczas do generowania dźwięku o częstotliwości nastawionej potencjometrem, przycisk 1 do zapamiętania w pamięci danej nuty, przycisk 2 do odtworzenie zapisanej wcześniej w pamięci sekwencji, a przycisk 3 służyłby do jej kasowania. Z uwagi na fakt, że licznik czasomierz 0 jest już wykorzystany do obsługi wyświetlaczy sygnał dźwiękowy można generować albo za pomocą pętli programowej z wykorzystaniem funkcji _delay_ms() albo też z wykorzystaniem kolejnego licznika czasomierza [3].

   Przeskalować tak wartości odczytywane z potencjometru, aby mieściły się one w zakresie od około 100 do około 4000. Nastawy dokonywane potencjometrem będą określać częstotliwość sygnału generowanego (przycisk 0) lub częstotliwość zapisywanej nuty (przycisk 1). Po naciśnięciu przycisku 2 sekwencja ma być odtwarzana, przyjąć czas pomiędzy odtworzeniami kolejnych nut na poziomie 1 s. Pamięć dla poszczególnych nut może mieć wielkość na poziomie 50 elementów, które powinny być 16 bitowe. Na wyświetlaczu powinna widnieć ilość nut zapisanych do pamięci, a w procesie odtwarzania numer granej nuty.

2.6 Prosty system alarmowy (4 godziny)

Cel ćwiczenia

Celem ćwiczenia jest nabycie doświadczenia w pisaniu średnio zaawansowanych programów w języku C na mikrokontrolerze AVR z wykorzystaniem biblioteki avr-libc. Ćwiczenia umożliwia zwiększenie umiejętności w tworzeniu aplikacji kontrolno pomiarowych oraz poszerzenie wiedzy na temat budowy prostych interfejsów użytkownika.

Zagadnienia do przygotowania

Powtórzyć informacje na temat sterowania multipleksowego wyświetlaczy 7-mio segmentowych. Przygotować w formie pisemnej wartości reprezentujące na wyświetlaczu 7-mio segmentowym litery A, b, C, d, E, F oraz zapoznać się z działaniem kodu programu zawartym w bieżącym ćwiczeniu.

Przebieg ćwiczenia

1. Dokonać analizy poniższego programu.

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <avr/pgmspace.h>

unsigned char lcd_digits[10] __attribute__((progmem)) =
{0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x6f};

unsigned char key;
unsigned char data[4]={2,0,0,8};

ISR(TIMER0_COMP_vect)
{
        static unsigned char num=0;

        PORTB=(PORTB&0xf0)|1<<num;

        PORTA=pgm_read_byte(lcd_digits+data[num]);

        if(num==3)
                num=0;
        else
                num++;
}

void keyboard(void)
{
        char k;
        static unsigned char p;
```

```
        k=(PINB>>4)|0xf0;

        if(k==0xff)
        {
                p=0;
                return;
        }

        if(p==1)
                return;

        _delay_ms(10);

        if(k!=((PINB>>4)|0xf0))
                return;

        p=1;
        key=~k;
}

void menu(void)
{
        static unsigned char pos=0;
        switch(key)
        {
                case 1:
                        if(pos>0)
                                pos--;
                break;
                case 2:
                        if(pos<3)
                                pos++;
                break;
                case 4:
                        if(data[pos]<9)
                                data[pos]++;
                break;
                case 8:
                        if(data[pos]>0)
                                data[pos]--;
                break;
        }
        key=0;
}
```

```
int main(void)
{
        DDRA=0xff;
        DDRB=0x0f;
        PORTB|=0xf0;

        TCCR0=0x0c;
        OCR0=0x40;
        TIMSK|=1<<OCIE0;
        SREG|=1<<SREG_I;

        while(1)
        {
                keyboard();
                menu();
        }

        return 0;
}
```

Program od programów zawartych w dwóch ostatnich ćwiczeniach różni się głównie funkcją menu() i tylko ona zostanie w tym ćwiczeniu omówiona.

Funkcja menu() implementuje najwyższą warstwę interfejsu użytkownika. Przypisuje ona przyciskom 0 i 1 możliwość edycji indeksu edytowanego wyświetlacza (zmienna pos). Edycję cyfry przypisanej danemu wyświetlaczowi przeprowadza się za pomocą dwóch pozostałych przycisków. Jak łatwo ustalić na podstawie programu wartości przypisane wyświetlaczom mogą mieścić się w zakresie od 0 do 9, a edytowany indeks wyświetlacza od 0 do 3.

2. Dokonać kompilacja istniejącego programu. Zaprogramować mikrokontroler i prześledzić działanie systemu.

3. Rozbudować program tak, aby możliwe było dodatkowo ustawianie na wyświetlaczach cyfr typowych dla reprezentacji szesnastkowej.

4. Dodać do programu opcję migania edytowanej cyfry. Należy zauważyć, że aby zaimplementować miganie należy zmodyfikować niskopoziomowy kod obsługi wyświetlacza (ISR). Należy również zauważyć, że nośnikiem informacji o tym, która cyfra jest edytowana jest zmienna pos oraz, że zmienna ta nie jest widoczna w podprogramie obsługi przerwania, ponieważ jest zmienną lokalną zdefiniowaną w innym bloku programu.

5. Naszkicować algorytm w odniesieniu do istniejącego programu prostego systemu alarmowego, który będzie posiadał 4 czujniki o ustawianej na poziomie interfejsu użytkownika czułości (zwarcie lub rozwarcie), zwłoce czasowej (czas na wpisanie kodu) oraz kodzie dostępu. Program powinien umożliwiać edycję czułości oraz zwłoki czasowej niezależnie dla każdego z czujników oraz kodu odblokowującego dla całego systemu alarmowego w trybie edycji (po wpisaniu hasła, kiedy nie wystąpiła aktywność żadnego z czujników).

W przypadku, kiedy nastąpi wykrycie stanu aktywnego któregoś z czujników użytkownik powinien mieć możliwość wpisania kodu odblokowującego. W takiej sytuacji jednak wpisanie kodu odblokowującego nie powinno automatycznie przenosić do trybu edycji, ale tylko rozbrajać daną aktywację czujnika.

Szczegóły interfejsu użytkownika mogą być następujące. Przycisk 0 może służyć w trybie edycji do wyboru czujnika, przycisk 1 może służyć do wyboru parametru dla danego czujnika, natomiast przyciski 2 i 3 mogą służyć do edycji wybranego dla danego czujnika parametru. Zaraz po wejściu w tryb edycji możliwe powinno być ustawianie parametrów czujnika 0, po pierwszym wciśnięciu przycisku 0 system powinien umożliwiać edycję ustawień czujnika 1,.. , po czwartym wciśnięciu przycisku 0 system powinien umożliwiać edycję alarmu, a po piątym system alarmowy powinien zostać uzbrojony. Edycję alarmu w tym trybie należy przeprowadzać analogicznie do edycji ustawień czujnika - przycisk 1 może służyć do wyboru cyfry kodu, a przyciski 2 i 3 do zmiany wartości wybranej cyfry.

Kod odblokowujący może być 4-ro cyfrowy (0, 1,.. , e, f), wówczas wszystkie cyfry będą widoczne jednocześnie. Wpisywanie kodu odblokowującego, kiedy alarm znajduje się w trybie uzbrojonym może zostać zaimplementowane w taki sposób, że każdy z wyświetlaczy zostanie skojarzony z jednym z przycisków, którego naciskanie będzie powodowało wybór kolejnej wartości cyfry na skojarzonym z tym przyciskiem wyświetlaczu. W przypadku, kiedy na wyświetlaczach pojawi się odblokowująca sekwencja, system automatycznie, bez potwierdzania żadnym z przycisków przejdzie w tryb edycji, lub jeśli wystąpiła aktywacja któregoś z czujników zaniecha odliczania do uruchomienia alarmu.

6. Czujniki zwarcia lub rozwarcia można zrealizować z wykorzystaniem przewodów połączeniowych.

7. Do uzyskania akustycznego sygnału alarmowego należy wykorzystać przetwornik elektroakustyczny znajdujący się na zestawie ZL10.

2.7 Port szeregowy (2 godziny)

Cel ćwiczenia

Celem ćwiczenia jest zdobycie wiedzy na temat modułu uniwersalnej transmisji szeregowej mikrokontrolera oraz utrwalanie informacji na temat samego standardu transmisji. Zadaniem ćwiczenia jest również rozszerzenie umiejętności programowania mikrokontrolera w języku C z wykorzystaniem biblioteki avr-libc oraz zdobycie doświadczenia w rozbudowie wielowątkowych aplikacji uruchamianych na mikrokontrolerach.

Zagadnienia do przygotowania

Powtórzyć zagadnienie transmisji szeregowej RS232. Zapoznać się pobieżnie z modułem transmisji szeregowej mikrokontrolera na poziomie pozwalającym odnieść parametry transmisji szeregowej do ustawień i mozliwości modułu mikrokontrolera. Zapoznać się z ideą kolejkowania danych z wykorzystaniem bufora cyklicznego oraz zastanowić nad jego implementacją w języku C.

Przebieg ćwiczenia

1. Dokonać analizy poniższego programu.

```
#include <avr/interrupt.h>
#include <avr/io.h>
#include <stdlib.h>
#include <util/delay.h>
```

```
#define queueDataInSize 8

struct queue
{
        char * data;
        volatile unsigned char begin;
        volatile unsigned char end;
        unsigned char size;
};

struct queue dataIn;

unsigned char nextQueueIndex(unsigned char i, unsigned char n)
{
        if(i==n-1)
                i=0;
        else
                i++;

        return i;
}

ISR(USART_RXC_vect)
{
        volatile unsigned char tmp=nextQueueIndex(dataIn.end,dataIn.size);

        unsigned char data=UDR;
        if(tmp!=dataIn.begin)
        {
                dataIn.data[dataIn.end]=data;
                dataIn.end=tmp;
                PORTA=PORTA&0xf8|dataIn.end;
        }
}

int main(void)
{

        dataIn.data=malloc(queueDataInSize);
        dataIn.size=queueDataInSize;
        dataIn.begin=0;
        dataIn.end=0;

        DDRA=0xff;
        UBRRL=51;
```

```
        //UCSRA=0;
        UCSRB=0x98;
        UCSRC=0x86;

        SREG|=1<<SREG_I;

        PORTA=PORTA&0xf8|dataIn.begin;
        PORTA=PORTA&0x1f|dataIn.end<<5;

        while(1)
        {
                if(dataIn.begin!=dataIn.end)
                {
                        UDR=dataIn.data[dataIn.begin];
                        dataIn.begin=nextQueueIndex(dataIn.begin,dataIn.size);
                        PORTA=PORTA&0x1f|dataIn.begin<<5;

                        unsigned char i;
                        for(i=0; i<100; i++)
                                _delay_ms(10);
                }
        };
        return 0;
}
```

Struktura queue została zadeklarowana w celu uproszczenia i ujednolicenia obsługi danych odbieranych łączem szeregowym. Zawiera ona wskaźnik na dane data, wielkość bufora cyklicznego w bajtach size, indeks ostatnio zapisanego elementu end oraz indeks elementu, który ma zostać odczytany jako pierwszy begin.

Funkcja nextQueueIndex() oblicza kolejny indeks (może to być indeks dla odczytu lub zapisu) w buforze cyklicznym na podstawie jego wielkości oraz aktualnego indeksu.

Makro ISR wywołane z parametrem USART_RXC_vect dokonuje rejestracji podprogramu obsługi przerwania generowanego w momencie odebrania przez układ transmisji szeregowej ramki. Podprogram obsługi przerwania na samym początku oblicza indeks zapisu kolejnego elementu w buforze cyklicznym. W przypadku, kiedy nowo obliczony indeks jest różny (czyli cyklicznie mniejszy) od indeksu pierwszej danej do odczytu, wykonywany jest wpis odebranej danej (UDR) do bufora, w przeciwnym przypadku podprogram nie wykonuje żadnego działania. Wielkość kolejki ustalono na 8 bajtów, stąd też 3 najmłodsze bity zmiennej end wysyłane na 3 najmłodsze bity portu PA (PORTA) pozwalają określić stan indeksu zapisu kolejki - znajomość stanu wskaźników bufora cyklicznego jest pożądana podczas śledzenia działania.

Zadaniem pętli głównej jest cykliczne wysyłanie przez port szeregowy co pewien czas (około 1 s) danych zgromadzonych w buforze. W przypadku, kiedy indeks zapisu end jest różny od indeksu odczytu begin (czyli wówczas, kiedy bufor nie jest pusty) następuje odczyt elementu z bufora cyklicznego do rejestru danych interfejsu szeregowego, co skutkuje rozpoczęciem transmisji. Stan zmiennej begin jest raportowany na 3 najstarszych bitach portu PA.

2. Dokonać kompilacji programu i zaprogramować mikrokontroler. Sprawdzić połączenie (USB) pomiędzy komputerem osobistym a modułem konwertera USR-RS232 umieszczonym w złączu ZL1USB. Uruchomić na komputerze osobistym program Hyper Terminal i skonfigurować ten program do pracy z prędkością ustaloną na podstawie kodu źródłowego.

3. Sprawdzić działanie systemu. Wciskać kolejne przyciski na komputerze i obserwować odbierane dane (okno Hyper Terminala). Należy zauważyć, że kolejne dane odbierane są co czas równy około 1 s oraz, że wysłanie w krótkiej chwili (poniżej 1 s) większej ilości danych (8) skutkuje utratą danych.

4. Kwalifikator typu volatile powoduje, że zmienna nie jest poddawana optymalizacji, czyli tuż po użyciu jest zawsze odkładana do pamięci, a nie przechowywana w rejestrach ogólnego przeznaczenia celem późniejszego szybszego dostępu. Należy zauważyć, że podprogram obsługi przerwania oraz pętla główna działają asynchronicznie i jeżeli zmienna end nie zostałaby wyłączona z optymalizacji moment zrównania jej wartości z wartością zmiennej begin mógłby zostać przegapiony i program mikrokontrolera uległby zawieszeniu. Wykasować kwalifikator typu volatile dla zmiennej end i doprowadzić do zawieszania mikrokontrolera.

5. Rozbudować program tak, aby odbierane z komputera osobistego przez mikrokontroler literki były wysyłane z powrotem do komputera po zmianie ich wielkości.

6. Rozbudować program działający na mikrokontrolerze do poziomu wyszukiwania wybranych napisów i zastępowania ich innymi. Na przykład napis “jeden” wysłany z komputera osobistego może być po odebraniu przez mikrokontroler zamieniony i wysłany jako “1”, a napis “name” zamieniony na “ATmega16” itp.

2.8 Prosty system pomiarowy (3 godziny)

Cel ćwiczenia

Celem ćwiczenia jest pogłębienie umiejętności pisania programów implementujących interfejs użytkownika, które wykorzystują kilka nawzajem współpracujących modułów.

Zagadnienia do przygotowania

Powtórzyć budowę oraz działanie przetwornika analogowo cyfrowego, modułu transmisji szeregowej, licznika czasomierza 0 oraz pamięci EEPROM [3]. Korzystając z dokumentacji [6] oraz śledząc plik avr/interrupt.h wyszukać nazw, które należałoby przekazać jako argumenty makra ISR aby zarejestrować podprogram obsługi przerwania, w przypadku przetwornika analogowo cyfrowego - zakończenia przetwarzania, w przypadku modułu transmisji szeregowej - odebrania, wysłania, opróżnienia bufora danych, licznika czasomierz 0 - przepełnienia, zrównania oraz pamięci EEPROM - zakończenia zapisu. Zwrócić uwagę aby nazwy te były przeznaczone dla mikrokontrolera ATmega16.

Przebieg ćwiczenia

1. Opracować algorytm aplikacji, która z częstotliwością 100 Hz dokonuje odczytu wejścia przetwornika przetwornika analogowo cyfrowego, do którego podpięty jest suwak potencjometru oraz dokonuje uśredniania 100 kolejnych pomiarów. Wartość średnia, która jest generowana co 1 s powinna być zapisywana do bufora cyklicznego umieszczonego w pamięci EEPROM. Wielkość bufora cyklicznego ustalić na 32 elementy. W pamięci EEPROM zapisywać również znacznik końca oraz znacznik początku danych w buforze cyklicznym. Zastanowić się nad obsługą portu szeregowego na poziomie odbierania,

interpretowania i wykonywania pojedynczych komend oraz krótkich raportów. Wśród komend interpretowanych przez mikrokontroler powinny się znajdować takie, które zlecają odczyt wybranej średniej, odczyt wszystkich zapisanych aktualnie średnich w kolejności chronologicznej oraz zlecają wyliczenie i odczyt średniej ze zgromadzonych w pamięci EEPROM średnich. Raportowanie powinno polegać na wysyłaniu na terminal kropki podczas zapisu każdej nowej średniej oraz znaku nowej linii w przypadku zapisu do zerowego elementu bufora.

2. Ustalić, które fragmenty kodu można zrealizować w podprogramach obsługi przerwań. Ustalić kanały komunikacyjne pomiędzy wykorzystywanymi podprogramami obsługi przerwań i pętlą główną.

3. Określić wartości słów konfiguracyjnych wykorzystywanych komponentów, a następnie napisać program dokonujący konfiguracji tych komponentów.

4. Uruchomić komunikację z komputerem. Zaimplementować rozpoznawanie wspomnianych w punkcie komend. Na tym etapie projektu rozpoznanie komendy może być sygnalizowane poprzez negację stanu diody świecącej, a zwracane wartości mogą być stałymi.

5. Uruchomić mechanizm odczytu i zapisu pamięci EEPROM. Zastanowić się nad tym, czy podprogram obsługi przerwania generowany w momencie zakończenia zapisu w pamięci EEPROM można wykorzystać w celu zapisu kolejnego pół-słowa czy może i również w celu zapisu kolejnej wartości średniej.

6. Uczynić mechanizm zapisu do pamięci EEPROM cyklicznym. Zastanowić się czy lepiej jest wykorzystać do tego celu podprogram obsługi przerwania generowanego przez przetwornik analogowo cyfrowy, czy może wykorzystać do tego celu któreś z przerwań generowane przez licznik czasomierz 0. Przewidzieć możliwość blokowania zapisu do pamięci EEPROM.

7. Ostatnim etapem projektu będzie połączenie programu komunikującego z komputerem z programem dokonującym zapisu do pamięci EEPROM.

Literatura

[1] Atmel. *Development Tools User Guide*. Atmel Corporation, 1998. AVR-Assembler_User_Guide.pdf.

[2] Atmel. *8-bit AVR Instruction Set*. Atmel Corporation, 2005. 8-bit_AVR_Instruction_Set.pdf.

[3] Atmel. *8-bit AVR Microcontroller with 16K Bytes In-System Programmable Flash ATmega16 ATmega16L*. Atmel Corporation, 2007. ATmega16.pdf.

[4] Dick Barnett. *AVR Simulation with the ATMEL AVR Studio 4*. Purdue University, 2005. AVR_Studio.pdf.

[5] BTC. *ZL10AVR Uniwersalny zestaw uruchomieniowy dla mikrokontrolerów AVR*. Wydawnictwo BTC, 2005. ZL10AVR-pl.pdf.

[6] GNU Project. *avr-libc Reference Manual 1.5.1.20071029*. Doxygen, 2007. avr-libc-Reference_Manual.pdf.