Podstawy Internetu Rzeczy
Instrukcja do zajęć laboratoryjnych

Laboratorium 5

Przerwania
Enkoder

Opracował opiekun przedmiotu dr inż. Krzysztof Chudzik

Wydział Informatyki i Telekomunikacji
Politechnika Wrocławska

Wrocław, 2024.09.28 14:57:14

Spis treści

Lista zadań

1 Przerwania

1.1 Przerwania i ich obsługa

Informacje na temat zdarzeń w systemie z mikrokontrolerem możemy uzyskać na dwa sposoby. Pierwszy to metoda odpytywania (ang. *pooling*), drugi to przerwania. Odpytywanie stosowaliśmy do tej pory, gdy sprawdzaliśmy, na przykład, stan przełączników przyciskanych (przycisków czerwonego lub zielonego).

Przerwanie (żądanie przerwania) sygnalizuje, że nastąpiło jakieś zdarzenie (lub osiągnięto stan), na który należy zareagować odkładając wykonywanie bieżących zadań i zająć się obsługą zdarzenia, które zostało zasygnalizowane (zgłoszone). Oznacza to, że w reakcji na zdarzenie zmieniamy wykonywanie programu. Przerwania są elementem sterowania przebiegiem wykonywania programu.

Przerwania mogą pochodzić z różnych urządzeń mikrokontrolera. Część może być wyzwolona na skutek konfiguracji podsystemu (urządzenia) mikrokontrolera. Przykładem są przerwania zegarowe od liczników (timers), które są konfigurowalne programowo. Inną grupę stanowią przerwania będące wynikiem interakcji z otoczeniem. W tym przypadku źródeł jest bardzo dużo. Można skonfigurować przerwania generowane na skutek zmiany stanu pinu wejściowego. Można skonfigurować przerwania informujące o określonych zdarzeniach lub stanach na porcie komunikacyjnym, przetworniku analogowo-cyfrowym,

komparatorze, itp. Przerwania mogą być zgłaszane w zawiązku z przebiegiem programu, gdzie mechanizm Watchdog mikrokontrolera może wykryć „zawieszenie się" programu. Przerwanie może też być zgłoszone w związku ze stanem zasilania mikrokontrolera, na przykład, obniżonym napięciem zasilania mikrokontrolera, co przy zasilaniu bateryjnym lub akumulatorowym pozwala mikrokontrolerowi ograniczyć pobór mocy lub wyłączyć się w sposób bezpieczny dla użytkownika i sprzętu, w którym jest zainstalowany. Powyższy krótki przegląd nie wyczerpuje źródeł i powodów zgłaszania przerwań, ale pokazuje jak szerokie jest wykorzystanie mechanizmu przerwań.

W skład mikrokontrolera wchodzą podsystemy i urządzenia mikrokontrolera, które potrafią generować wiele różnych przerwań związanych z różnymi zdarzeniami w nich zachodzącymi. Przerwanie, aby wystąpiło, musi być skonfigurowane i włączone.

Przerwania obsługuje kontroler przerwań. Obsługa przerwań może odbywać się z wykorzystaniem priorytetów przerwań na mikrokontrolerach, które implementują taki mechanizm. Obsługa przerwania może być wykonana natychmiast lub odłożona w czasie, na przykład, w związku z aktualnie trwającą obsługą przerwań o wyższym priorytecie.

Gdy kontroler przerwań zadecyduje o wykonaniu obsługi przerwania, to licznik programu jest ustawiony tak, aby wskazywał wektor przerwania. Wektor przerwania zawiera zazwyczaj skok do procedury obsługi danego przerwania. I ten skok jest wykonywany w przebiegu wykonywania programu.

Procedura obsługi przerwania nie posiada argumentów i nie zwraca rezultatu. Posiada jednak dostęp do rejestrów, w których mogą być zawarte kolejne informacje o źródle i przyczynach przerwania. Ma też dostęp do danych w programie, na przykład, za pomocą zmiennych globalnych lub wywołań innych funkcji czy procedur. W tym ostatnim przypadku należy zachować bardzo daleko idącą ostrożność. Obsługa przerwania musi być maksymalnie krótka, gdyż w przerywa ona wykonywanie standardowych operacji wykonywanych przez mikrokontroler oraz zawiesza wykonywanie obsługi innych przerwań o niższym priorytecie i nie pozwala obsługiwać nowych przerwań o priorytecie takim samym lub niższym. Dlatego też w ramach obsługi przerwań programuje się tylko konieczne operacje. Nie ma tu miejsca na długie procedury, obliczenia, czy oczekiwanie na cokolwiek. Obsługa polega zazwyczaj na tym, że wykonujemy krótki zestaw instrukcji, które muszą być wykonane jak najszybciej w ramach obsługi przerwania, a za pomocą zmiennych globalnych przekazujemy informację o zdarzeniu i zebrane dane do programu głównego. Program główny odczyta je w ramach zwykłego przebiegu programu zdefiniowanego w iteracji niekończącej się pętli głównej programu (`while(true){...}`, lub funkcji `loop()` w Arduino).

Gdy obsługa przerwania zakończy się, wtedy następuje powrót z obsługi przerwania. Wznawiane jest wykonywanie programu, od miejsca, w którym był wykonywany przed wystąpieniem przerwania.

Z przerwaniami i przerywaniem wykonywania programu wiąże się poważny problem. Załóżmy, że następował odczyt danych i w czasie trwania tej operacji zgłoszone zostało przerwanie, którego obsługa zmieniła dane, w trakcie których odczytywania był program. Po powrocie z przerwania program kontynuuje odczyt tych danych. Oczywiście dane te będą przypadkowe w tym momencie. Każdy odczyt, który zajmuje czas większy niż cykl zegarowy może zostać przerwany, i na przykład, odczyt czy zapis liczb 16 czy 32 bitowych na kontrolerach 8 bitowych może być przerwany i wynik należy uznać za przypadkowy. Chcielibyśmy uzyskać operację atomową, której nie można przerwać, bez sygnalizacji błędu. Aby przeciwdziałać takim sytuacjom, wyłącza się obsługę przerwań. Następnie, w ramach odczytu, przepisuje się wartości zmiennych modyfikowanych przez obsługę przerwań do zmiennych, które nie są modyfikowane przez procedury obsługi przerwań. W przypadku zapisu operacja przebiega w odwrotnym kierunku. Po zakończeniu tej operacji przywraca się obsługę przerwań. Podobnie jak przerwania taka operacja też musi być jak najkrótsza. W niektórych środowiskach dostępne są pomoce do realizacji takich operacji. W zestawie narzędzi do programowania mikrokontrolerów AVR, jakim jest ATmego328P z Arduino UNO, jest to makro `ATOMIC_BLOCK()`. Zaprezentujemy je nieco dalej.

Drugi problem, który związany jest z wymianą danych pomiędzy programem głównym i obsługą przerwań, to problem ulotności danych. W zasadzie jest to problem jeszcze bardziej ogólny. Dotyczy on operowania danymi, które są zmieniane poza przebiegiem programu, tak jak go widzi kompilator. Będą to zatem dane zmieniane przez przerwania oraz dane, które zmieniane są zewnętrznie w stosunku do programu, na przykład, zmienne związane ze stanami portów mikrokontrolera. Główny problem polega na tym, że kompilator, podczas optymalizacji kodu, może uznać, że wartość zmiennej na pewnym odcinku przebiegu programu nie zmienia się, więc może zadecydować, że wartość ta będzie przechowywana w rejestrze mikrokontrolera, aby przyspieszyć do niej dostęp. Jeśli jednak zmienna zostanie zmodyfikowana przez przerwanie lub inne zdarzenie poza przebiegiem programu analizowanym przez kompilator, to program użyje nieaktualnej wartości z rejestru, zamiast aktualnej zmienionej wartości. Aby temu przeciwdziałać wykorzystujemy słowo kluczowe języka programowania C/C++: `volatile`. Użyte przed deklaracją typu zmiennej informuje kompilator, że nie wolno optymalizować operacji odczytu i zapisu tej zmiennej. Proszę jednak mieć na uwadze, że słowo to nie rozwiązuje wszystkich problemów związanych z wielowątkowością. W przypadku mikrokontrolerów AVR, mamy jeden wątek, przerywany przerwaniami, więc to będzie sprawdzać się. Więcej informacji dla zainteresowanych w artukule Wikipedii „Zmienna ulotna" (link).

Przejdźmy teraz do rozwiązania konkretnego problemu.

1.2 Wykorzystanie przerwań do sterowania programem

Sformułujmy problem programistyczny. Należy przygotować program do uruchomienia na zestawie laboratoryjnym. Program ma za zadanie zliczać naciśnięcia przełącznika przyciskanego (przycisku) i na ekranie LCD podawać ilość naciśnięć oraz odstęp czasu pomiędzy kolejnymi naciśnięciami. Program ma być odporny na zjawisko drgania styków. Należy wykorzystać przerwanie, aby po naciśnięciu przycisku przerwać wykonywanie programu i odnotować dokładny czas jego naciśnięcia do obliczeń odstępu czasu pomiędzy naciśnięciami.

Mikrokontroler ATmega328P z Arduino UNO dysponuje dwoma typami przerwań, które można skonfigurować z pinami mikrokontrolera i płytki Arduino.

Jeśli chodzi o pierwszy typ przerwań, to w dokumentacji Arduino na stronie producenta, jesteśmy informowani, że mamy do dyspozycji dwa przerwania, tak zwane zewnętrzne, które związane są z dwoma pinami Arduino. Jest to przerwanie `INT.0` związane z pinem 2 (`D2`), oraz `INT.1` związane z pinem 3 (`D3`). Na schemacie pinów (wyprowadzeń) Arduino, są one oznaczone jako `INT[0]` i `INT[1]`. W karcie katalogowej mikrokontrolera ATmega328P mają oznaczenie `INT0` i `INT1`.

Od tego pierwszego typu przerwań zaczniemy.

Nieprzypadkowo, przycisk czerwony podłączony jest do pinu 2 (`D2`). Można wykorzystać przerwanie zewnętrzne INT.0, które jest związane z tym pinem, do obsługi zdarzenia naciśnięcia przycisku.

Popatrzmy na propozycję rozwiązania naszego problemu programistycznego z wykorzystaniem przerwania INT.0 w postaci kodu 1. Opis kodu znajduje się pod kodem.

Kod 1: Obsługa przerwań z wykorzystaniem przerwania zewnetrznego INT0

```
#include <LiquidCrystal_I2C.h>

#define RED_BUTTON 2
#define DEBOUNCING_PERIOD 100

LiquidCrystal_I2C lcd(0x27, 16, 2);

int pressCounter = 0;
volatile unsigned long buttonTimestamp = 0UL;
unsigned long previousButtonTimestamp = 0UL;
unsigned long elapsedStartTimestamp = 0UL;

void interruptAction()
{
    buttonTimestamp = millis();
}

void printResults(int count, unsigned long time)
{
    char buffer[40];
    sprintf(buffer, "Press count%5d", count);
    lcd.setCursor(0, 0);
    lcd.print(buffer);
    sprintf(buffer, "Time [ms]%7lu", time);
    //sprintf(buffer, "Time [s]%4lu.%03d", time / 1000, time % 1000);
    lcd.setCursor(0, 1);
    lcd.print(buffer);
}

void setup()
{
    pinMode(RED_BUTTON, INPUT_PULLUP);
    lcd.init();
    lcd.backlight();
    printResults(0, 0UL);

    attachInterrupt(digitalPinToInterrupt(RED_BUTTON), interruptAction, FALLING);
}

void loop()
{
    noInterrupts();
    unsigned long localButtonTimestamp = buttonTimestamp;
    interrupts();

    if (localButtonTimestamp != previousButtonTimestamp && millis() > localButtonTimestamp + DEBOUNCING_PERIOD)
    {
        if (digitalRead(RED_BUTTON) == LOW)
        {
            pressCounter++;

            unsigned long elapsedTime = localButtonTimestamp - elapsedStartTimestamp;
            elapsedStartTimestamp = localButtonTimestamp;

            printResults(pressCounter, elapsedTime);
        }
        previousButtonTimestamp = localButtonTimestamp;
    }
}
```

Zacznijmy od funkcji `setup()`. Po znanej już inicjalizacji czerwonego przycisku i wyświetlacza LCD, pojawia się funkcja `attachInterrupt()`. Jej zadaniem, jest powiązanie procedury obsługi przerwania z przerwaniem. Proszę zapoznać się

z dokumentacją funkcji `attachInterrupt()` (link). Pierwszym argumentem jest wskazanie przerwania. Można tam wpisać odpowiednią wartość, ale zamiast tego można użyć makra `digitalPinToInterrupt()`, które wyznaczy tą wartość na podstawie numeru pinu. Drugi argument to wskaźnik procedury obsługi przerwania. Nazwa procedury, baz nawiasów zawiera adres funkcji, lub inaczej, jest wskaźnikiem na funkcję. Trzeci argument mówi nam na jaki typ zmiany mamy zareagować. Użyty identyfikator „FALLING" reprezentuje zmianę stanu pinu z `HIGH` na LOW, dokładnie tak, jak dzieje się to przy naciskaniu przycisku w naszym zestawie.

Pamiętajmy, że naciśnięciu przycisku towarzyszą drgania styków. Zwolnieniu przycisku też towarzyszą drgania styków i też zostanie zgłoszone przerwanie.

Procedura przerwania, nazwana tu `interruptAction()`, jest bardzo prosta i polega wyłącznie na odnotowaniu terminu zgłoszenia przerwania, czy naciśnięcia przycisku, aby mieć możliwie dokładny ten termin. Kolejne akcje nie wymagają już takiej szybkości i zostaną przekazane do pętli głównej programu. Proszę zauważyć, że zmienna `buttonTimestamp` wykorzystywana przez procedurę obsługi przerwania do przekazania terminu, zadeklarowana jest z modyfikatorem typu `volatile`.

W pętli głównej (funkcja `loop()`) odczytujemy wartość zapisaną przez przerwanie. Ponieważ mamy drgania styków, można spodziewać się, że kolejna zmiana może nastąpić w czasie odczytu. Aby chronić przed tym, wyłączamy przerwania, odczytujemy wartość zmiennej i zapisujemy w zmiennej lokalnej niedostępnej dla obsługi przerwania, i przywracamy obsługę przerwań. Czynione jest to za pomocą makr, z których dokumentacją proszę się zapoznać, `noInterrupts()` (link) i `interrupts()` (link).

Następnie, w ramach instrukcji warunkowej `if()` następuje eliminacja drgań styków. Po ustaniu drgań styków odbywa się sprawdzenie stanu przycisku. Chodzi o weryfikację, czy przerwanie nie było zgłoszone na skutek drgań styków przy zwalnianiu przycisku. Tu dla uproszczenia przykładu dydaktycznego, aby nie komplikować, założono, że użytkownik nie jest tak szybki, aby zwolnić przycisk, zanim nie dokona się to sprawdzenie jego stanu po ustaniu drgań styków. Proponuję zainteresowanym, rozważenie reimplementacji tak, aby to założenie nie było konieczne. W następnych krokach następuje obliczenie odstępu czasowego pomiędzy naciśnięciami przycisku i wydrukowanie wyników na ekranie LCD z wykorzystaniem funkcji `printResults`.

W fukcji `printResults`, do przygotowania i formatowania tekstu wyświetlanego na ekranie wykorzystano funkcję „sprintf()". Czas wyświetlany jest w milisekundach, tak jak go podaje funkcja `millis()`. Dobrze by było, aby pojawił się w sekundach z częścią ułamkową. Są jednak pewne ograniczenia związane z drukowaniem wartości zmiennoprzecinkowych przez sprintf. Zainteresowanych proszę o zapoznanie się, na przykład, z artykułami „sprintf() with Arduino" (link) oraz „Do you know Arduino? – sprintf and floating point" (link). Pozostaje więc, albo użyć funkcji `print()` i drukować łańcuch tekstowy fragmentami, albo zastosować przeliczanie na liczbach całkowitych tak, jak w zakomentowanej linii w kodzie 1.

Tak na marginesie, o ile jest to tylko możliwe, należy unikać operacji i liczb zmiennoprzecinkowych na mikrokontrolerach, które nie są wyposażone w jednostki zmiennoprzecinkowe (FPU), ponieważ operacje na nich wykonywane są programowo, co zajmuje stosunkowo dużo czasu procesora.

Zauważmy, że typ przerwań właśnie przez nas wykorzystany, jest mało liczny. Tylko dwa przerwania związane z dwoma konkretnymi pinami na Arduino UNO. Co zrobić z przyciskiem zielonym, który nie jest podłączony do tego typu przerwań? W dalszej części tego laboratorium, będziemy zajmować się enkoderem, który też można obsługiwać poprzez przerwania. Gdybyśmy wykorzystali INT.1 na przycisk zielony, to na enkoder już tego typu przerwań w naszym zestawie laboratoryjnym brakuje.

Trzeba zatem problem rozwiązać nieco inaczej i z pomocą przychodzi nam drugi typ przerwań. Nie jest on już tak reklamowany i wspierany mocno przez producenta Arduino, ale informacje o nim znajdziemy w karcie katalogowej mikrokontrolera ATmega328P w rozdziale zatytułowanym „External Interrupts". Jest to temat także poruszany na forach poświęconych Arduino.

Za chwilę będziemy posługiwać się pojęciami: porty I/O i rejestry mikrokontrolera. Przypominam, że te pojęcia zostały wprowadzone na wykładzie drugim i trzecim.

Wykorzystamy tu też przerwania, które też są formalnie przerwaniami zewnętrznymi, ale ich konfiguracja i obsługa odbywa się nieco inaczej. Są to przerwania skojarzone z każdym portem I/O w ATmega328P osobno i każde zgłaszane jest dla wszystkich pinów w porcie łącznie. Przypomnijmy, że ATmega328P udostępnia trzy porty B, C i D, jedno przerwanie obsługuje cały port, więc przerwań jest trzy. Ten typ przerwania jest konfigurowalny na każdym pinie płytki Arduino UNO, który nie jest pinem zasilania, albo napięcia referencyjnego. Na schemacie pinów (wyprowadzeń) Arduino UNO są one oznaczone jako `PCINT[x]`, gdzie `x` jest liczbą z zakresu od 0 do 23. W karcie katalogowej katalogowej ATmega328P, w oznaczeniach, nie ma nawiasów, i oznaczenie ma postać `PCINTx`.

Zaletą tych przerwań jest ich dostępność, ale mają też wady. Mają mniejszą funkcjonalność niż poprzednio omówione `INT.0` i `INT.1`, i trudniej je skonfigurować. Co więcej, jak wspomnieliśmy, jedno przerwanie obsługuje cały port, więc programowo trzeba ustalić, na którym pinie nastąpiła zmiana, co komplikuje program. Spróbujmy zatem skonfigurować ten typ przerwania.

W dalszej części będziemy posługiwać się przyciskiem zielonym, który nie ma dostępu do przerwań wykorzystanych w kodzie 1.

Popatrzmy na propozycję rozwiązania naszego problemu programistycznego z wykorzystaniem przerwania `PCINT[x]` w postaci kodu 2. Opis kodu znajduje się pod kodem.

Kod 2: Obsługa przerwań z wykorzystaniem przerwania zewnetrznego typu `PCINT[x]`.

```
1 #include <util/atomic.h>
2 #include <LiquidCrystal_I2C.h>
```

```
3
4  #define GREEN_BUTTON 4
5
6  #define DEBOUNCING_PERIOD 100
7
8  LiquidCrystal_I2C lcd(0x27, 16, 2);
9
10 int pressCounter = 0;
11 volatile unsigned long buttonTimestamp = 0UL;
12 unsigned long previousButtonTimestamp = 0UL;
13 unsigned long elapsedStartTimestamp = 0UL;
14
15 ISR(PCINT2_vect)
16 {
17     buttonTimestamp = millis();
18 }
19
20 void printResults(int count, unsigned long time)
21 {
22     char buffer[40];
23     sprintf(buffer, "Press count%5d", count);
24     lcd.setCursor(0, 0);
25     lcd.print(buffer);
26     sprintf(buffer, "Time [ms]%7lu", time);
27     //sprintf(buffer, "Time [s]%4lu.%03d", time / 1000, time % 1000);
28     lcd.setCursor(0, 1);
29     lcd.print(buffer);
30 }
31
32 void setup()
33 {
34     pinMode(GREEN_BUTTON, INPUT_PULLUP);
35     lcd.init();
36     lcd.backlight();
37     printResults(0, 0UL);
38
39     PCICR |= (1 << PCIE2);
40     PCMSK2 |= (1 << PCINT20);
41 }
42
43 void loop()
44 {
45     unsigned long localbuttonTimestamp;
46
47     ATOMIC_BLOCK(ATOMIC_RESTORESTATE)
48     {
49         localbuttonTimestamp = buttonTimestamp;
50     }
51
52     if (localbuttonTimestamp != previousButtonTimestamp && millis() > localbuttonTimestamp + DEBOUNCING_PERIOD)
53     {
54         if (digitalRead(GREEN_BUTTON) == LOW)
55         {
56             pressCounter++;
57
58             unsigned long elapsedTime = localbuttonTimestamp - elapsedStartTimestamp;
59             elapsedStartTimestamp = localbuttonTimestamp;
60
61             printResults(pressCounter, elapsedTime);
62         }
63         previousButtonTimestamp = localbuttonTimestamp;
64     }
65 }
```

Zielony przycisk podłączony jest do pinu 4 (D4), który może wywołać sygnał przerwania PCINT[20].

Ten typ przerwania nie jest zbyt mocno wspierany przez standardowy język Arduino. Zatem musimy wykorzystać notę katalogową kontrolera ATmega328P i rozdział „External Interrupts” i programować zgodnie z zestawem narzędzi (toolchain) dla mikrokontrolerów z rodziny AVR. Tam są zdefiniowane makra i identyfikatory przezentowane dalej. Do programowania wygodniej jest wykorzystać VS Code, choć w naszym przykładzie nie jest to konieczne.

Jak wspomniano, mamy trzy przerwania, po jednym dla każdego portu. Zmiana związana z PCINT[20] będzie zgłaszana poprzez przerwanie oznaczone w nocie katalogowej jako PCI2. Do włączania przerwań służy rejestr PCICR – Pin Change Interrupt Control Register. Nasze przerwanie ustawiane jest na pozycji o indeksie 2 tego rejestru przez zapisanie tam wartości „1”. W kodzie programu, w funkcji `setup()` jest to uczynione zapisem `PCICR |= (1 « PCIE2);`. Identyfikator PCIE2 jest zdefiniowany jako wartość 2. To nie powoduje jeszcze, że przerwanie zostanie zgłoszone. Trzeba w rejestrze odpowiadającym tej grupie pinów (portowi) zgłosić, na zmiany których pinów należy reagować przerwaniem PCI2. Czyni się to w rejestrze PCMSK2 – Pin Change Mask Register 2. Dla naszego przerwania PCINT20 to pozycja o indeksie 4. Informacja ta jest wprowadzana linią kodu: `PCMSK2 |= (1 « PCINT20);`.

Rejestracja procedury obsługi przerwań wykonywana jest przez makro `ISR()`, w którego ciele piszemy całą procedurę. Jest to zarejestrowany już odpowiednik `interruptAction()` z kodu 1, czyli zawiera odpowiednik wywołania `attachInterrupt()`.

W funkcji `loop()`, makro `ATOMIC_BLOCK()` jest odpowiednikiem sekwencji wywołań `noInterrupts()` i `interrupts()`. Jego użycie wymaga dołączenia pliku nagłówkowego `util/atomic.h`. Ma ono dwie zalety. Po pierwsze, jest to blok kodu. Jeśli otworzyliśmy nawias klamrowy, to musimy go zamknąć, bo inaczej wystąpi błąd kompilacji. Czyli unikamy sytuacji, gdzie wyłączyliśmy przerwania, a potem zapominamy je włączyć. Druga zaleta, to argument, jak skonfigurować stan mikrokontrolera po wykonaniu bloku. Tu argument wskazuje, że przywracamy stan z przed wywołania bloku. Ale wcale nie musi tak być. Zainteresowanych tematem odsyłam do strony „Atomically and Non-Atomically Executed Code Blocks" (link).

Pozostała część kodu 2 nie różni się zasadniczo od kodu 1, więc dalsze wyjaśnienia nie są wymagane.

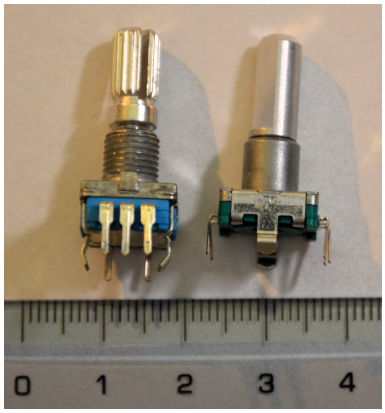
2 Enkoder

2.1 Enkoder i jego działanie

Użycie klasycznego potencjometru związane jest z pomiarem napięcia, do którego wykorzystuje się przetwornik analogowo-cyfrowy. Enkodery dostarczają sygnału, który może być traktowany jako sygnał cyfrowy. Omówimy jeden z najprostszych wśród enkoderów - enkoder kwadraturowy. Istota pomiaru polega na obserwowaniu stanów dwóch wyjść (wyprowadzeń) enkodera, które mogą być traktowane jako wyjścia cyfrowe.

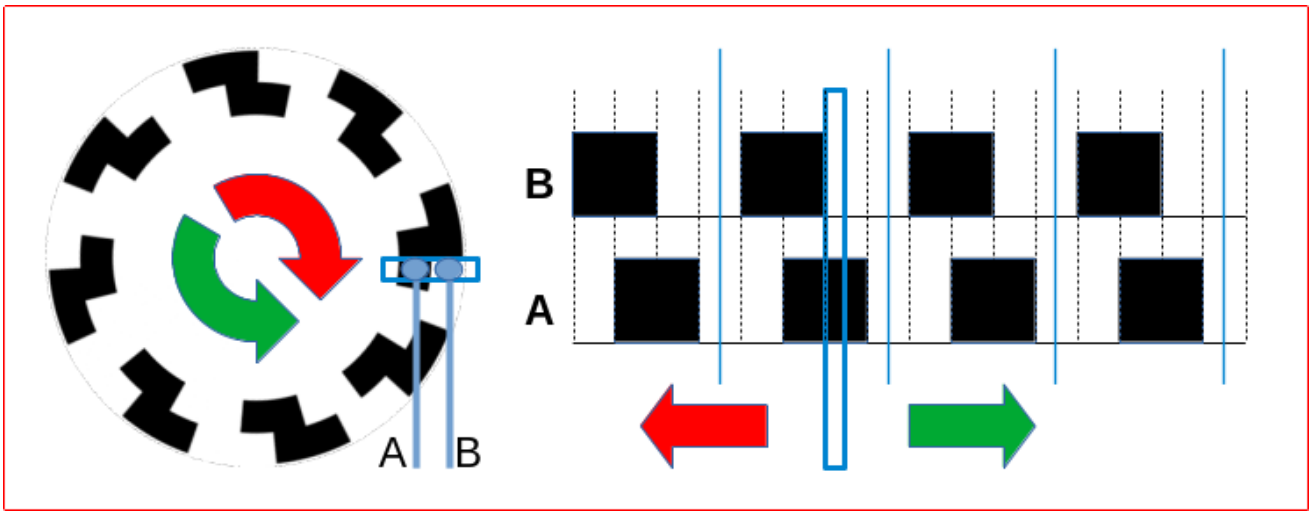
W układach potencjometrami analizuje się sygnały analogowe, w układach z enkoderami - cyfrowe. Stąd, klasyczne potencjometry wypierane są powoli przez enkodery, ponieważ użycie enkodera nie wymaga użycia przetwornika analogowo-cyfrowego. Enkoder nakłada jednak na programistę, obowiązek obsłużenia zdarzeń związanych ze zmiana stanów wyjść enkodera. Można to zrealizować, na przykład, z wykorzystaniem przerwań.

Enkoder, do sprzętu powszechnego użytku, jak wzmacniacze audio, kina domowe, itp., wizualnie nie odbiega bardzo wyglądem od potencjometru. Enkodery widzimy na fotografii 1.



Fotografia 1: Enkodery.

Niemniej jednak zasada jego działania jest zupełnie inna. Działanie jednego z rodzajów enkodera ilustruje rysunek 1.



Rysunek 1: Zasada działania enkodera kwadraturowego.

Zamiast ścieżki rezystancyjnej, w jego wnętrzu znajduje się niewielka okrągła płytka (lewa strona rysunku 1), która zawiera nadrukowane połączenia elektryczne (czarne pola). Pola te są elektrycznie połączone najczęściej z masą. Jeśli jet to

enkoder taki jak na fotografii 1, to jest to zazwyczaj środkowe z trzech wyprowadzeń i najczęściej łączymy je z masą. Pola są ułożone w okręgi przesunięte względem siebie o połowę pola. Nad każdym okręgiem pól znajduje się ślizgacz elektryczny podłączony do zewnętrznego wyprowadzenia (na rysunku oznaczone jako A i B). Jeśli pod ślizgaczem znajdzie się czarne pole, to zamykany jest obwód pomiędzy masą, a wyprowadzeniem ślizgacza (A lub B). Przypomina to działanie przełącznika. I rzeczywiście, aby dostarczać informacje do mikrokontrolera, podłączamy wyprowadzenia A i B do mikrokontrolera (płytki Arduino) i załączamy rezystory podciągające na pinach do których są podłączone.

Wyjaśnijmy jak interpretować sygnały enkodera. Aby łatwiej było prowadzić objaśnienia „rozwińmy" nasze okręgi pól stykowych do postaci diagramu jak po prawej stronie rysunku. Niebieski prostokąt („okienko") pokazuje umowną pozycję ślizgaczy A i B. Jeśli obracamy tarczą enkodera w prawo (czerwona strzałka), to po diagramie przesuwamy „okienko" w lewo (czerwona strzałka pod diagramem). Jeśli obracany tarczą w lewo, to wszystko dzieje się w przeciwnych kierunkach (zielone strzałki.

Obracając pokrętłem enkodera wyczuwamy, że przeskakuje z pozycji stabilnej na kolejną pozycję stabilną (wyczuwamy „ząbki"). Pionowe niebieskie linie na diagramie wyznaczają pozycje ślizgaczy w pozycji stabilnej. Jeśli założymy, że obracamy enkoderem w prawo, to wychodząc z pozycji stabilnej (niebieska pionowa linia), najpierw zwarty zostanie ślizgacz A, następnie zwarty B, dalej rozwarty zostanie A, następnie rozwarty B, i dochodzimy do położenia stabilnego. Przy obrocie w lewo jest analogiczna, ale zdarzenia na ślizgaczu B wyprzedzają zdarzenia ślizgacza A.

Typowa obsługa enkodera wygląda tak że przyjmujemy jeden ze ślizgaczy jako ten, którego impulsy będziemy liczyć, a drugi wskaże kierunek obrotu. Przyjmijmy, że na ślizgaczu A będziemy zliczali impulsy, a dokładniej, będziemy zliczali ile razy został zwarty do masy poprzez czarne pole. To uściślenie jest bardzo ważne, bo gdy wykryjemy zwarcie ślizgacza A do masy (wejście na czarne pole) to sprawdzamy stan ślizgacza B. Zauważmy, że zawsze, gdy obracamy pokrętłem enkodera w prawo (czerwona strzałka), to w momencie zwarcia ślizgacza A do masy ślizgacz B nie jest zwarty (jest tam otwarty obwód). Natomiast zawsze, gdy obracamy pokrętłem enkodera w lewo (zielone strzałki), w momencie zwierania ślizgacza A ślizgacz B jest już zwarty. Jeśli teraz będziemy sprawdzać stany pinów odpytując odpowiednio często, lub skorzystamy z przerwań, łatwo wykryjemy ruch enkodera i kierunek jego obrotu.

Wykrycie ruchu i kierunku pozwala nam zwiększać, lub zmniejszać wartość zmiennej, która może reprezentować nastawę enkodera.

2.2 Programowa obsługa enkodera

Enkoder jest elementem stosunkowo delikatnym. W sprzęcie domowego użytku można spotkać go jako element pozwalający regulować siłę głosu, na przykład, we wzmacniaczach audio, kinach domowych, itp. Najczęściej często wyposażony jest w duże pokrętło (potocznie, „gałkę"). Duże pokrętło stosowane celowo i nie chodzi tu jedynie o względy estetyczne. Ma ograniczać prędkość obracania enkoderem. Duże pokrętło trzeba obrócić ruchem całej dłoni. Małe pokrętło, jak w naszym zestawie, można obrócić bardzo szybko miedzy kciukiem i palcem wskazującym. Ze względu na niewielkie rozmiary i złożoność struktury na płytce elektrycznej wewnątrz, należy też spodziewać się błędnych odczytów z enkodera, szczególnie, gdy obracamy zbyt szybko pokrętło. Wobec powyższego, **przy ćwiczeniu z enkoderem w naszym zestawie laboratoryjnym wymaga się przekręcania go z umiarkowaną prędkością**.

Poznaliśmy już zasadę działania enkodera, ale ponieważ nie ma ogólnie przyjętych i ścisłych standardów ich budowy, to dobrze jest umieć zidentyfikować samodzielnie działanie posiadanego enkodera. W tym celu został przygotowany prymitywny program zamieszczony jako kod 3. Pokazuje on świeceniem diod stan pinów, do którego podłączone są wyprowadzenia enkodera. Zapalona dioda oznacza na wyprowadzeniu stan HIGH, zgaszona LOW. Enkoder jest podłączony tak jak przycisk. Dla każdego wyprowadzenia enkodera jest włączony w mikrokontrolerze rezystor podciągający, i, podobnie jak przycisk, w stanie spoczynkowym ma rozwarte styki, czyli panuje na nich stan wysoki napięciowy (HIGH). Stąd po załadowaniu i uruchomieniu programu diody zaświecą się. Pokrętłem enkodera trzeba obracać bardzo powoli, ponieważ cały cykl zmian zachodzi pomiędzy każdymi dwoma stanami stabilnymi („ząbkami") pokrętła enkodera.

Kod 3: Program identyfikujący wyprowadzenia i kolejność przełączania na wyprowadzeniach enkodera.

```
1  #define LED_RED 6
2  #define LED_BLUE 3
3
4  #define ENCODER1 A2
5  #define ENCODER2 A3
6
7  void setup()
8  {
9      pinMode(LED_RED, OUTPUT);
10     pinMode(LED_BLUE, OUTPUT);
11     pinMode(ENCODER1, INPUT_PULLUP);
12     pinMode(ENCODER2, INPUT_PULLUP);
13 }
14
15 void loop()
16 {
17     digitalWrite(LED_RED, digitalRead(ENCODER1));
18     digitalWrite(LED_BLUE, digitalRead(ENCODER2));
19 }
```

Zadanie 1: Identyfikacja i obserwacja funkcjonowania enkodera w zestawie laboratoryjnym

Korzystając z programu zamieszczonego jako kod 3, proszę zaobserwować w zestawie laboratoryjnym jak zachowują się wyprowadzenia enkodera przy obrocie w lewo i przy obrocie w prawo o jedną pozycję. Odnotować, a wręcz narysować sobie, sekwencję stanów przy obrocie do kolejnej pozycji spoczynkowej (o jeden „ząbek"), zarówno w lewo, jak i w prawo. Proszę odpowiedzieć jak, patrząc tylko na przebieg stanów wyprowadzeń w czasie, można stwierdzić, w którą stronę obrócono enkoder i o ile pozycji. Proszę opisać krótko słownie algorytm, który to realizuje.

Sformułujmy teraz problem programistyczny, który enkoder ma rozwiązać. Program ma zmieniać, przy obrocie enkoderem, łagodnie kolor diody RGB z niebieskiej na czerwoną przy obrocie enkodera w prawo, i przeciwnie przy obrocie w lewo. Wartość zmiennej sterującej, ustawianej przez enkoder, ma być wyświetlana na wyświetlaczu LCD. Należy wyeliminować zjawisko drgania styków, które występuje tak samo jak w przypadku przycisków.

Można zrealizować wiele algorytmów odczytu pozycji enkodera. W kodzie 4 zawarta jest propozycja odczytywania. Opis programu znajduje się pod kodem.

Kod 4: Program obługujący enkoder metodą odpytywania (*pooling*).

```
#include <LiquidCrystal_I2C.h>

#define LED_RED 6
#define LED_BLUE 3

#define ENCODER1 A2
#define ENCODER2 A3

#define DEBOUNCING_PERIOD 100

LiquidCrystal_I2C lcd(0x27, 16, 2);

void printResults(int val)
{
    char buffer[40];
    sprintf(buffer, "Encoder: %3d", val);
    lcd.setCursor(2, 0);
    lcd.print(buffer);
}

void myAction(int val)
{
    printResults(val);
    analogWrite(LED_RED, val);
    analogWrite(LED_BLUE, 255 - val);
}

void setup()
{
    pinMode(LED_RED, OUTPUT);
    pinMode(LED_BLUE, OUTPUT);
    pinMode(ENCODER1, INPUT_PULLUP);
    pinMode(ENCODER2, INPUT_PULLUP);
    lcd.init();
    lcd.backlight();
    myAction(0);
}

int encoderValue = 0;
int lastEn1 = LOW;
unsigned long lastChangeTimestamp = 0UL;
void loop()
{

    int en1 = digitalRead(ENCODER1);
    int en2 = digitalRead(ENCODER2);

    unsigned long timestamp = millis();
    if (en1 == LOW && lastEn1 == HIGH && timestamp > lastChangeTimestamp + DEBOUNCING_PERIOD)
    {
        if (en2 == HIGH)
        {
            if (encoderValue < 255)
                encoderValue += 15;
        }
        else
        {
            if (encoderValue > 0)
                encoderValue -= 15;
        }
```

```
61          lastChangeTimestamp = timestamp;
62
63          myAction(encoderValue);
64      }
65      lastEn1 = en1;
66  }
```

Akcja wykonywana przy zmianie wartości enkodera jest zdefiniowana w funkcji `myAction()` i wywoływanej przez nią funkcji `printResults()`. Zawartość funkcji `setup()` to inicjalizacja pinów sterujących diodami, pinów wejściowych enkodera, inicjalizacja wyświetlacza i wykonanie akcji dla stanu zerowego.

Odczyt pozycji enkodera i jej zmian zrealizowany jest w funkcji `loop()` wykonywanej cyklicznie. Zmienna `encoderValue` pełni rolę licznika enkodera. Będziemy zliczać zmiany stanu z wysokiego na niski wyprowadzenia oznaczonego „1”, natomiast wyprowadzenie „2” użyjemy do określenia kierunku obrotu pokrętła enkodera. Zatem, najpierw odczytujemy stany wyprowadzeń enkodera. Następnie sprawdzamy, czy pojawiła się interesująca nas zmiana stanu wyprowadzenia „1” enkodera, która nie jest spowodowana drganiem styków. Jeśli tak to odczytujemy stan wyprowadzenia „2” enkodera. W przypadku obrotu w prawo zwiększamy licznik enkodera, w lewo zmniejszamy. Inkrementacja i dekrementacja działają w zakresie od 0 do 255, ponieważ takie wartości pozwalają sterować wprost diodą RGB za pomocą modulacji szerokości impulsu (PWM). Zmiany licznika enkodera ustalono na 15, ponieważ zmiany o warość 1 wymagają bardzo wielu obrotów enkoderem, aby zmienić stan diody na przeciwny, na przykład, z niebieskiego na czerwony. Liczba 15 to dzielnik 255 i regulacja diody zachodzi w pełnym wyznaczonym zakresie w rozsądnej liczbie obrotów enkodera. Na koniec wykonywana jest akcja związana ze zmianą stanu enkodera.

Jak już to wspomniano, pracę enkodera można obsłużyć w ramach obsługi przerwań. Prezentuje to kod 5. Jego opis znajduje się pod nim.

Kod 5: Program obługujący enkoder z wykorzystaniem przerwania.

```
1  #include <util/atomic.h>
2  #include <LiquidCrystal_I2C.h>
3
4  #define LED_RED 6
5  #define LED_BLUE 3
6
7  #define ENCODER1 A2
8  #define ENCODER2 A3
9
10 #define DEBOUNCING_PERIOD 100
11
12 LiquidCrystal_I2C lcd(0x27, 16, 2);
13
14 void printResults(int val)
15 {
16     char buffer[40];
17     sprintf(buffer, "Encoder: %3d", val);
18     lcd.setCursor(2, 0);
19     lcd.print(buffer);
20 }
21
22 void myAction(int val)
23 {
24     printResults(val);
25     analogWrite(LED_RED, val);
26     analogWrite(LED_BLUE, 255 - val);
27 }
28
29 void setup()
30 {
31     pinMode(LED_RED, OUTPUT);
32     pinMode(LED_BLUE, OUTPUT);
33     pinMode(ENCODER1, INPUT_PULLUP);
34     pinMode(ENCODER2, INPUT_PULLUP);
35     lcd.init();
36     lcd.backlight();
37     myAction(0);
38
39     PCICR |= (1 << PCIE1);
40     PCMSK1 |= (1 << PCINT10);
41 }
42
43 volatile int encoder1 = HIGH;
44 volatile int encoder2 = HIGH;
45 volatile unsigned long encoderTimestamp = 0UL;
46
47 ISR(PCINT1_vect)
48 {
49     encoder1 = digitalRead(ENCODER1);
```

```
50      encoder2 = digitalRead(ENCODER2);
51      encoderTimestamp = millis();
52  }
53
54  int encoderValue = 0;
55  int lastEn1 = LOW;
56  unsigned long lastChangeTimestamp = 0UL;
57  void loop()
58  {
59
60      int en1;
61      int en2;
62      unsigned long timestamp;
63
64      ATOMIC_BLOCK(ATOMIC_RESTORESTATE)
65      {
66          en1 = encoder1;
67          en2 = encoder2;
68          timestamp = encoderTimestamp;
69      }
70
71      if (en1 == LOW && timestamp > lastChangeTimestamp + DEBOUNCING_PERIOD)
72      {
73          if (en2 == HIGH)
74          {
75              if (encoderValue < 255)
76                  encoderValue += 15;
77          }
78          else
79          {
80              if (encoderValue > 0)
81                  encoderValue -= 15;
82          }
83          lastChangeTimestamp = timestamp;
84
85          myAction(encoderValue);
86      }
87      lastEn1 = en1;
88  }
```

W porównaniu z kodem 4, zasadnicza idea nie zmieniła się. Wprowadzono jedynie metodę obsługi przerwania zaprezentowaną wcześniej w kodzie 2. Musimy reagować na zmianę stanu pinu 1 enkodera, który jest podłączony do pinu `A2` płytki Arduino. Z pomocą schematu pinów (wyprowadzeń) Arduino i karty katalogowej ATmega328P odnajdujemy, że interesujący nas sygnał to `PCINT10`, który spowoduje wyzwolenie przerwania PCI1. Włączenie wyzwolenia `PCI1` przez sygnał `PCINT10` odbywa się poprzez rejestr `PCMSK1`. I ta konfiguracja została wprowadzona w funkcji `setup()`. Procedura obsługi przerwania polega na zapisaniu aktualnych stanów wyprowadzeń enkodera i znacznika czasowego, kiedy wystąpiło przerwanie, do ulotnych (`volatile`) zmiennych globalnych. Podobnie jak w kodzie 2, nie musimy identyfikować dokładnie pinu, który zgłosił przerwanie, bo tylko jeden w pin w tej grupie (porcie) może je zgłosić

W funkcji `loop()` przepisujemy wartości zmiennych ulotnych do lokalnych i dalsza część zasadniczo funkcjonuje jak w programie bez przerwań (kod 4), więc dalsze wyjaśnienia nie są wymagane.

Zadanie 2: Sterowanie programem przy pomocy enkodera

Przygotuj program, który będzie pozwalał sterować świeceniem diody RGB. Program ma być wyposażony w przewijalne menu na ekranie wyświetlacza LCD (w kolejnych liniach wyświetlane są dwie pozycje z menu). Samodzielnie zaprojektuj strukturę menu programu. Nawigacja po menu (przemieszczanie się pomiędzy pozycjami) ma odbywać się za pomocą enkodera. Wybór pozycji menu odbywa się za pomocą przycisku. Minimalna funkcjonalność to zapalanie i gaszenie wybranej diody. Można też zaimplementować ustawianie jasności każdego kolory diody enkoderem. Warto też program oprzeć na przerwaniach zgłaszanych przy zmianie stanu enkodera.