Mikroelektronika w technice i medycynie

Podstawy programowania systemów wbudowanych

Instrukcja do ćwiczeń laboratoryjnych

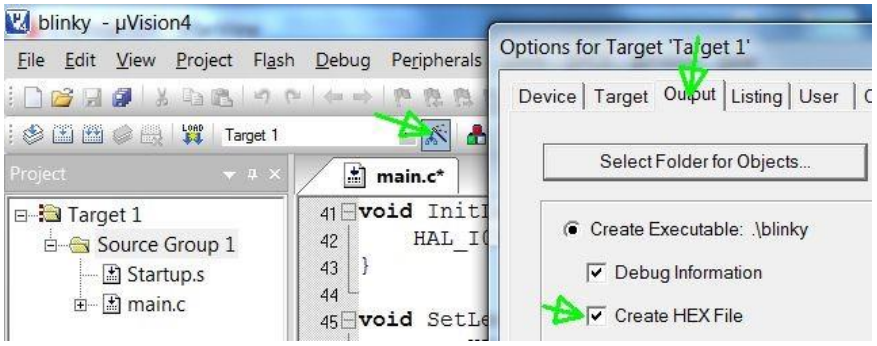
Mirosław Żołądź
2023-03-04

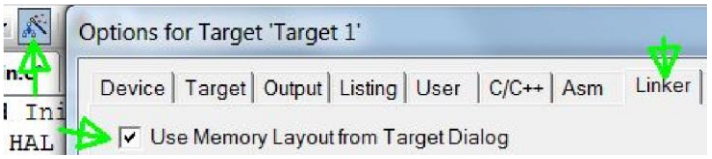
Spis treści

1 Obsługa środowiska Keil

1.1 Tworzenie projektu

1. Na pulpicie stworzyć katalog, w którym będzie trzymany projekt. Nazwa katalogu powinna składać się z nazwiska oraz wyrazu „TEST”.
2. Uruchomić środowisko KEIL (skrót w menu start).
3. Utworzyć nowy projekt: *MENU->Project->New µVision Project*, nazwa „TEST”
   Podczas tworzenia program zapyta się o Target - w naszym przypadku należy wybrać *NXP->LPC2129/01*) i o dołączenie "startup code" - należy potwierdzić okienko dialogowe,
4. Włączyć tworzenie pliku wynikowego (*.hex), potrzebnego do zaprogramowania modułu ewaluacyjnego, a następnie skonfigurować linker.





5. Stworzyć plik z kodem źródłowym programu (MENU->File->New) i zapisać go pod "main.c" do katalogu projektu. (należy pamiętać o rozszerzeniu ”.c”).
6. Nowo stworzony plik "main.c" należy następnie dołączyć do projektu.

Okienko „Projekt” (po lewej stronie) rozwinąć drzewo "Target 1 -> Source group 1", z menu kontekstowego "Source group" wybrać "Add files to group" i dodać stworzony wcześniej plik "main.c" (przycisk „Add” w okienku dialogowym).

7. Wstawić do main.c przykładowy program:

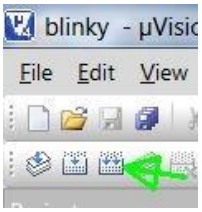
```
int iLoopCtr;

int iPlusCounter;
int iMinusCounter;

int main()
{
   iPlusCounter=10
   iMinusCounter = 10;

   for (iLoopCtr=0 ; iLoopCtr<5; iLoopCtr++)
   {
      iPlusCounter=iPlusCounter+3;
      iMinusCounter=iMinusCounter-3;
   }
}
```

8. Skompilować program – „Menu->Projekt->Build target” lub



Kompilator powinien zgłosić informację o błędzie (errors) lub ostrzeżenie (warning) w przypadku znalezienia „podejrzanej” instrukcji.

Należy wtedy znaleźć linijkę kodu, w której znajduje się instrukcja będąca przyczyną błędu lub ostrzeżenia (klikając dwa razy na linijkę z informacją o błędzie lub z ostrzeżeniem co spowoduje ustawienie się kursora w oknie programu w odpowiednim miejscu) a następnie poprawić kod (w tym przypadku dodać średnik).

9. Ponownie skompilować program.

Wyeliminować ewentualne błędy lub ostrzeżenia.

(Dodać inicjalizacje zmiennej iMinusCounter wartością 10)

10. Po wyeliminowaniu błędów i przyczyn ostrzeżeń kompilacja powinna kończyć się komunikatem:

```
FromELF: creating hex file...
"blinky.axf" - 0 Error(s), 0 Warning(s).
```

1.2 Debugowanie programu

Krokowanie:

11. Uruchomić debugger: Menu->Debug->Start
12. Zamknąć okienko z kodem assemblera: Menu->View->Diasembly Window
13. Krokować program za pomocą przycisku F10 aż wskaźnik z lewej strony okna (żółta strzałka) dojdzie do końca programu.
14. Zamknąć debugger: Menu->Debug->Stop

Podglądnie stanu zmiennych:

15. Uruchomić debugger: Menu->Debug->Start
16. Otworzyć okienko podglądu zmiennych: Menu->View->Watch Window->Watch1
    (Powinno pojawić się w prawym dolnym rogu ekranu)
17. Ustawić kursor na liczniku pętli (for).
    Z menu kontekstowego wybrać „Add 'nazwa zmiennej' to Watch #1"
    (Zmienna powinna pojawić się w okienku „Watch1")
18. Krokować program za pomocą przycisku F10 obserwując stan licznika pętli.

Użcie breakpointów:

19. Zamknąć, a następnie ponownie uruchomić debugger:
20. Ustawić kursor na instrukcji „iMinusCounter=iMinusCounter-3;"
    Z menu kontekstowego wybrać „Insert/Remove breakpoint"
    (Powinien pojawić się czerwony znacznik po lewej stronie linii)
21. Przekrokować (F10) jedną iteracje pętli, obserwując stan licznika pętli (Watch #1).
22. Uruchomić symulację (Menu->Debug->Run),
    Symulator wykona następną iteracje pętli i zatrzyma się na breakpoincie.
23. Powtarzać poprzedni punkt obserwując stan licznika pętli.

UWAGA: Debuger\Symulator pracują na skompilowanym kodzie programu dlatego po modyfikacji kodu programu a przed debugowaniem program należy ponownie skompilować.

2 Struktura i archiwizacja programów

Struktura programu (pliku main.c)

Wszystkie programy tworzone w ramach Ćwiczeń Laboratoryjnych powinny zawierać przynajmniej plik main.c:

- Na początku pliku main.c należy dołączyć plik nagłówkowy z nazwami rejestrów mikrokontrolera (patrz rozdział GPIO). W przeciwnym wypadku w przypadku użycia nazwy rejestru, która to nazwa nie stanowi słowa kluczowego języka C kompilator zgłosi błąd. W celu dołączenia wspomnianego pliku należy ustawić kursor na początku programu i z menu kontekstowego wybrać **„Insert #include <LPC21xx.H>".**
- W następnych linijkach powinny znaleźć się ewentualne deklaracje zmiennych i definicje funkcji.
- Na końcu programu powinna znajdować się funkcja „**main**".
- Na początku funkcji „main" powinny znaleźć się wszystkie instrukcje, które mogą być wykonane tylko raz (np. inicjalizacje).
- Na końcu funkcji „main" powinna znajdować **pętla główna** programu, która powinna wykonywać się nieskończoną ilość razy (Pętla główna ma zwykle postać instrukcji **while** z argumentem 1).
- W pętli głównej powinny znajdować się instrukcje programu, które powinny być wykonywane cyklicznie.

Archiwizacja

Prowadzący ma prawo zażądać od studenta przedstawienia każdego z dotychczas wykonanych ćwiczeń. Dlatego sugeruje się przechowywać wszystkie programy w jednym katalogu. Powinny znajdować się w nim znajdować katalogi: C*urrent* i A*rchive*. Katalog *Current* powinien zawierać cały projekt zawierający ćwiczenie, nad którym aktualnie pracujemy. Katalog A*rchive* powinien zawierać projekty odpowiadające ćwiczeniom, nad którymi zakończono pracę. Projekty powinny być przechowywane w oddzielnych katalogach/podkatalogach o nazwach zawierających: numer rozdziału, pełną lub skróconą nazwę rozdziału oraz numer ćwiczenia. Przykład pokazano poniżej. MITP:

- Current
- Archive\10_UART\ 1,2,3

Po zakończeniu pracy nad ćwiczeniem należy:

- usunąć pliki robocze z katalogu projektu: menu\project\clean_target
- zamknąć środowisko Keil
- skopiować katalog Current w odpowiednie miejsce w katalogu Archive
- odpowiednio zmienić nazwę katalogu Current

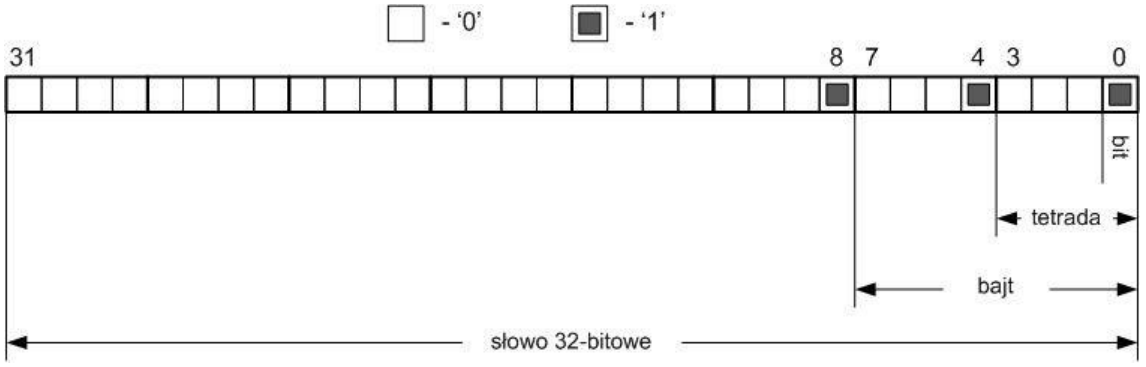
UWAGA: Wraz z wyłaczeniem komputera wszystkie pliki stworzone przez uzytkownika są usówane. Z tego powodu przed wyłaczeniem komputera należy zadbać o ich archiwizację. (pendrive, „chmura", poczta).

Jedną z możliwość jest kompresja\dekompresja całego katalogu zawierającego programy (katalog\menu_kontekstowe\send to compressed..).

3 Praca z rejestrami

Rejestr to układ służący do przechowywania i odtwarzania informacji w postaci bitów.

Podstawowym parametrem rejestru jest jego długość, czyli ilość bitów, które może zapamiętać.
Poszczególne bity w rejestrze są numerowane od zera. Przykładowo rejestr 32-bitowy zawiera bity od 0 do 31.



Rejestr ogólnego stosowania – służy do zapamiętywania danych (liczb, adresów itp.)

Rejestr konfiguracyjny – służy do określania parametrów pracy mikrokontrolera lub układów peryferyjnych.

Przykład: Bit 7 w rejestrze CLK_CTL może decydować z jaką częstotliwością pracuje mikrokontroler. Jeżeli bit równy 0 to jest to częstotliwość podstawowa a jeżeli 1 to dwa razy mniejsza.

Przykład: Bit 2 w rejestrze PIN_CTL może określać napięcie na jednej z nóżek mikrokontrolera. 0 – napięcie 0V, 1 – napięcie 3.3V.

Ustalanie bitów w rejestrze

Ustalanie wartości bitów w rejestrze odbywa się przez wpisanie do niego liczby. Rozmiar liczby odpowiada rozmiarowi rejestru.

Przykładowo, jeżeli rejestr jest 8-bitowy to liczba musi być bajtem a jeżeli 32-bitowy to liczba musi być słowem 32-bitowy. Inaczej mówiąc zawsze wpisujemy wartości wszystkich bitów.

Liczby w rejestrze zapisane są w kodzie binarnym. W odróżnieniu od kodu dziesiętnego, w którym cyfry na kolejnych pozycjach mają wagi będące potęgą dziesiątki: np. 245 = 2 x 10^2 + 4 x 10^1 + 5 x 10^0

W kodzie binarnym poszczególne cyfry (0 lub 1) mają wagi będące potęgą dwójki:
11001 [binarnie] = 1 x 2^4 + 1 x 2^3 + 0 x 2^2 + 0 x 2^1 + 1 x 2^0 = 1x16 +1x 8 +0x4 + 0x2 + 0x1 = 25 [dziesiętnie]

Z tego powodu bit zerowy nazywamy najmniej znaczącym lub najmłodszym bitem lub LSB (ang. Least Significant Bit) a ostatni bit liczby najbardziej znaczącym lub najstarszym lub MSB (Most Significant Bit).

Przykładowo żeby ustawić w rejestrze bity 0, 3 i 4 (00011001) należy do niego wpisać liczbę 25 (dziesiętnie) lub 11001 (binarnie).
Niestety żaden z zapisów nie jest zbyt czytelny. W większych liczb (np. 32-bitowych) trudno na podstawie liczby określić, na której pozycji są bity

Np.: 1497907208[dec], 0101 1001 0100 1000 0100 0000 0000 1000 [bin].

W przypadku kodu binarnego przyczyna jest duża liczba cyfr a w przypadku kodu dziesiętnego fakt ze pojedynczej cyfrze nie odpowiada całkowita liczba bitów.

Częściowym rozwiązaniem problemu jest kod heksadecymalny. W kodzie tym cyfry na poszczególnych pozycjach mogą przyjmować wartości 0-15 dziesiętnie, 0 – F heksadecymalnie (0,1..8,9, A,B,C,D,E,F). Jedna cyfra odpowiada dokładnie 4 bitom, co powoduje, że zapis jest bardziej zwięzły niż binarny. W praktyce można nauczyć się układu bitów dla poszczególnych cyfr na pamięć.

dziesiętnie	binarnie				heksadecymalnie
	MSB			LSB	
	3	2	1	0	
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	2
3	0	0	1	1	3
4	0	1	0	0	4
5	0	1	0	1	5
6	0	1	1	0	6
7	0	1	1	1	7
8	1	0	0	0	8
9	1	0	0	1	9
10	1	0	1	0	A
11	1	0	1	1	B
12	1	1	0	0	C
13	1	1	0	1	D
14	1	1	1	0	E
15	1	1	1	1	F

Przykład 1

1100 1001 [bin]	201 [dec]	C9 [hex]
1111 1010 0011 0001 [bin]	64049 [dec]	FA31[hex]

Przykład 2

Jeżeli chcemy żeby w słowie 32-bitowym jedynki były na pozycji **2**, **16**, **22** i **23,** odpowiednia liczba to **0C10004**.

Zapis do rejestru

Żeby poinformować kompilator ze używamy kodu heksadecymalnego a nie dziesiętnego dajemy przed liczba prefix „0x".

Przykładowo, jeżeli chcemy, żeby w rejestrze CTL były ustawione na jeden tylko bity 2, 16 i 23, a pozostałe były równe 0, piszemy instrukcję przypisania jak poniżej:

```
CTL = 0x00810004;
```

Ć**wiczenie**

Podać instrukcje, która ustawi w rejestrze INT bity 3, 5, 6, 7, 28, 29 i 31. Użyć zapisu heksadecymalnego.

(Podpowiedź: zapisać liczbę w kodzie binarnym grupując bity po cztery, następnie każda czwórkę/tetradę zamieniać na cyfrę w kodzie heksadecymalnym).

4 Układ GPIO

4.1 Opis

Jednostką, która pozwala na bezpośredni zapis i odczyt wejść mikrokontrolera jest jednostka **GPIO** (General Purpose Input/Output – Uniwersalne Porty Wejścia Wyjścia).

Umożliwia ona indywidualną kontrolę kierunku (wejście/wyjście) poszczególnych pinów mikrokontrolera. Użyty do ćwiczeń mikrokontroler (LPC 2129) posiada 2 porty we/wy, P0 i P1, każdy 32-bitowy.

Współpraca z portem odbywa się za pośrednictwem 4 rejestrów 32bitowych. Poszczególne bity w rejestrze odpowiadają poszczególnym wyprowadzeniom portu.

Do kontroli GPIO służą następujące rejestry:

- **IOxDIR** pozawala na kontrolę kierunku poszczególnych pinów portu x, zero ustawia pin jako wejściowy, jedynka jako wyjściowy
- **IOxPIN** służy do odczytu stanów pinów (wejść) portu,
- **IOxSET** wpisanie jedynki do danego bitu rejestru powoduje ustawienie jedynki na odpowiadającym mu pinie portu, wpisanie zera nie powoduje żadnej akcji, czyli rejestr służy tylko do ustawiania stanów wysokich („1”) na wyjściach,
- **IOxCLR** wpisanie jedynki do danego bitu rejestru powoduje ustawienie zera na odpowiadającym mu pinie portu, wpisanie zera nie powoduje żadnej akcji, czyli rejestr służy tylko do ustawiania stanów niskich („0”) na wyjściach.

Uwaga: „x” w nazwie rejestru oznacza numer portu, do którego należy rejestr, czyli przykładowo, jeżeli chcemy wpisać liczbę do rejestru DIR portu 1 robimy to instrukcją IO**1**DIR = 0x17;

4.2 Ćwiczenia

1. Utworzyć nowy projekt w katalogu *Current,* o nazwie składającej się ze swojego nazwiska.
2. Napisać program, który po kolei:
    - ustawi pin 16 i tylko 16 portu P1 jako wyjściowy (rejestr DIR)
    - ustawi pin 16 i tylko 16 portu P1 na „1” (rejestr SET)
    - ustawi pin 16 i tylko 16 portu P1 na „0” (itd.)
    - wejdzie w pętle nieskończoną „while(1){ }”

Program powinien mieć strukturę zgodną z opisem podanym w rozdziale 2.

Skompilować program, poprawić ewentualne błędy i wyeliminować przyczyny ostrzeżeń.

Wejść w tryb debugowania.

Otworzyć okno z rejestrami portu P1: Menu -> Peripherals-> GPIO Slow Interface -> P1.

Krokować program (F10) obserwując zachowanie bitów poszczególnych rejestrów.

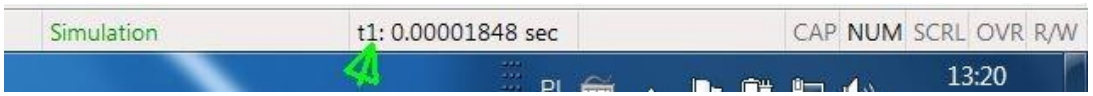
W celu powtórnego przekrokowania programu zatrzymać i powtórnie uruchomić debugowanie: Menu -> Start/Stop Debug Session.

3. Zmienić kod programu tak, żeby w nieskończoność zmieniał stan P1.16 na przeciwny.
   UWAGA: Każda modyfikacja programu wymaga jego powtórnej kompilacji, czyli ewentualnego wyjścia z trybu debugowania

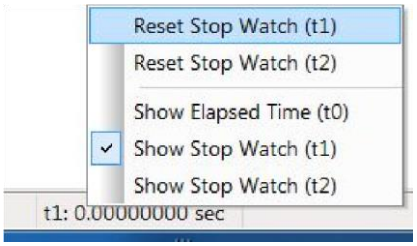
4. Napisać funkcję *Delay*, której zadaniem jest wprowadzanie opóźnienia.
   Funkcja nie powinna przyjmować ani zwracać żadnego argumentu.
   Funkcja powinna zawierać pętlę for. Czas wykonania funkcji będzie zależał od ilości iteracji pętli. Na początek można przyjąć wartość 1000.
   Wstawić funkcję *Delay* do programu między instrukcje ustawiającą a instrukcje kasującą bit na porcie P1.
   Skompilować program, usunąć przyczyny ewentualnych błędów i ostrzeżeń.

5. Dobrać ilość iteracji pętli w funkcji Delay tak żeby wprowadzała ona 1-sekundowe opóźnienie.

   W tym celu należy wykorzystać Stoper, który znajduje się na pasku statusu kompilatora. Mierzy on w sekundach czas wykonywania programu lub czas od ostatniego resetu Stopera:



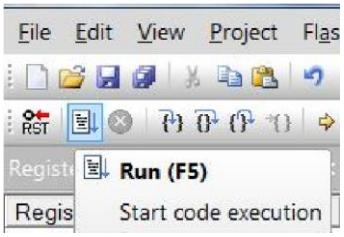
   Reset stopera wywołuje się z menu kontekstowego:



   W celu dobrania ilości iteracji pętli for tak żeby czas wykonania funcki Delay wynosił 1 sekundę należy w pierwszej kolejności zmierzyć ile czasu zajmuje wykonie fukcji Delay przy przy 1000 iteracji petli.

   Aby wykonać wspomniany pomiar należy:

   - wejśc w tryb debugowania,
   - ustawić breakpoint na instrukcji znajdującej się bezpośredniu za wywołaniem funkcji Delay,
   - przekrokować program do instrukcji znajdującej się bezpośrednio przed wywołaniem funkcji Delay, ● zresetować stoper,
   - uruchomić program:



     Program powinien zatrzymać się na breakpoincie.

   - Zarejestrować wskazanie stopera.
   - Na podstawie wskazań stopera dobrać z proporcji ilość iteracji petli która da opóxnienie 1-sekundowe.
   - Sprawdzić stoperem poprawność doboru.

UWAGA 2: Każda modyfikacja programu wymaga wyjścia z trybu debugowania i jego powtórnej kompilacji.

6. Przerobić funkcję Delay tak, aby czas opóźnienia był przekazywany w argumencie wywołania funkcji w milisekundach. Przykładowo, wywołanie „Delay(17)” ma wprowadzić 17-sto milisekundowe opóźnienie. Sprawdzić poprawność działania funkcji dla 1 sekundy i 1 milisekundy.

7. Przerobić program tak, aby P1.16 zmieniał stan na przeciwny raz na sekundę.
   Inaczej mówiąc P1.16 ma się cyklicznie przez sekundę równać 1, a następnie przez sekundę równać 0.
   Sprawdzić działanie programu na symulatorze.
   Sprawdzić działanie programu na mikrokontrolerze. W tym celu pobrać zestaw uruchomieniowy i zasilacz, a następnie wgrać program do mikrokontrolera według instrukcji znajdującej się w dokumencie „InstrukcjaObslugiArmEvm.pdf". Pin P1.16 mikrokontrolera jest podłączony do diody świecącej LED0, dlatego po poprawnym załadowaniu poprawnego programu dioda LED0 powinna zacząć migać.

8. Przerobić program tak, aby LED0 migała z częstotliwością 10 Hz.

9. Zmodyfikować poprzedni program tak, żeby zamiast dosłownych liczb heksadecymalnych przy wszystkich wpisach do wszystkich rejestrów używał masek bitowych, zdefiniowanych makrodefinicją #define, wstawioną po instrukcji „include". Dla bitu odpowiadającemu diodzie 0, maski powinna nazywać się LED0_bm.
   UWAGI:
   Po instrukcji *#define* nie daje się średnika.
   Postfiks „_bm" oznacza bitmask.
   Sprawdzić działanie programu na symulatorze.

UWAGA 1: W tym i w następnych programach przy wszystkich wpisach do wszystkich rejestrów używać masek bitowych

10. Przerobić program z poprzedniego punktu tak, aby pulsował diodą świecącą D3. Sposób podłączenia pinów portu 1 do diod świecących podano w dokumencie „InstrukcjaObsługiArmEVM.pdf" zamieszczonym jako załącznik do strony laboratorium.
    Działanie programu sprawdzić na symulatorze a następnie na zestawie uruchomieniowym. Instrukcja obsługi zestawu uruchomieniowego znajduje się w dokumencie podanym powyżej.
    UWAGA: Należy stworzyć nową maskę - LED3_bm.

11. Napisać program, który po kolei zapali diody LED0 do LED3.
    Na początku ma świecić pierwsza, a na końcu wszystkie cztery.
    Nie wprowadzać opóźnienia między zapalaniem diod.
    Program sprawdzić na symulatorze.

12. Napisać program, który będzie cyklicznie przesuwał punkt świetlny od LED0 do LED3 (0,1..3,0,1..3,…).
    Czas świecenia pojedynczej diody powinien wynosić 0.25 sekundy.
    UWAGA: W żadnym momencie wykonywania programu nie powinna się świecić więcej niż jedna dioda.
    Program sprawdzić na symulatorze i mikrokontrolerze.

13. Napisać funkcje *LedInit*, która będzie ustawiać piny podpięte do LED0-LED3 na wyjściowe oraz zapalać LED0.
    UWAGA: można użyć tylko masek bitowych zdefiniowanych we wcześniejszych punktach (użyć sumy bitowej).
    Działanie funkcji sprawdzić na symulatorze realizując funkcjonalność programu z punktu 12. (Zastąpić odpowiedni fragment kodu funkcją *LedInit*).

14. Napisać funkcje *LedOn(unsigned char ucLedIndeks)*, która będzie zapalać diodę LED o numerze podanym w argumencie. UWAGA: Po wywołaniu funkcji może świecić się co najwyżej jedna dioda.
    Działanie funkcji sprawdzić na symulatorze i mikrokontrolerze realizując funkcjonalność programu z punktu 12.

15. Zmodyfikować funkcje *LedInit* tak, aby ustawiała kierunek **tylko** pinów odpowiadających LED0-LED3. Kierunek pozostałych pinów powinien zostać niezmieniony.
    Pierwszy test przeprowadzić na symulatorze ustawiając zaraz po wejściu w tryb debugowania bit 23 P1 na 1 za pomocą okienka „Menu -> Peripherals-> GPIO Slow Interface -> P1". Po wykonaniu funkcji stan bitu powinien pozostać niezmieniony.
    Drugi test przeprowadzić realizując funkcjonalność programu z punktu 12 na symulatorze a następnie na mikrokontrolerze.

16. Napisać funkcję *ReadButton1()*, która będzie zwracać „1" jeżeli naciśniety jest przycisk „S0", a „0" w przeciwnym przypadku.
    Funkcję przetestować pisząc program, który będzie zapalał *Led1* jeżeli przycisk będzie naciśnięty, a *Led0* w przeciwnym przypadku. Test wykonać na symulatorze i mikrokontrolerze.

17. Przerobić funkcję *ReadButton1()* tak, żeby zwracała wynik typu *enum ButtonState {RELASED, PRESSED}*.
    Test jak w punkcie 16 z tym, że do wyboru, która dioda ma być zapalona **użyć instrukcji *switch***.
    Test wykonać na symulatorze i mikrokontrolerze.

18. Napisać funkcje *KeyboardInit*.
    Funkcja ta powinna ustawić na wejścia wszystkie piny portu odpowiadające przyciskom S0-S3, **nie modyfikując** przy tym pozostałych bitów rejestru. Test jak w punkcie 17.

19. Zamienić funkcję *ReadButton* na e*KeyboardRead*, która będzie zwracać *enum KeyboardState {RELASED, BUTTON_0, BUTTON_1, BUTTON_2, BUTTON_3}*.
    Jeżeli naciśnieto więcej niż jeden przycisk, funkcja powinna zwracać pierwszy naciśnięty (w przestrzeni, nie w czasie, np. jeśli naciśnięte są jednocześnie BUTTON_2 i BUTTON_3 funkcja powinna zwrócić BUTTON_2).

    Test: program, który zapala diodę o numerze takim jak naciśnięty przycisk. Jeżeli nie naciśnięto żadnego przycisku nie powinna palić się żadna dioda (użyć *LedOn* z argumentem 4).
    UWAGI:
    - W funkcji e*KeyboardRead* użyć sekwencji *if-elsów.*
    - M funkcji main użyć instrukcji *case.*

20. Napisać funkcję „StepLeft", której pojedyncze wywołanie będzie powodować przesunięcie punktu świetlnego w lewo (np. 0->1, 1->2, 2->3, 3->0).
    UWAGI:
    - Użyć funkcji „LedOn",
    - Użyć zmiennej globalnej unsigned int i operatora „modulo",
    - Docelowo funkcja powinna składać się z dwóch linii kodu,
    - Inkrementacje / dekrementacje zmiennej dać przed wywołaniem *LedOn*.
    - W programie testowym nie używać LedOn – zauważyć, że LedInit zapala diodę zerową.

    Test z punktu 12 na symulatorze i mikrokontrolerze.

21. Napisać funkcję StepRight. (np. 3->2, 2->1, 1->0, 0->3)
    Test z punktu 12 tylko w przeciwna stronę (symulator i mikrokontroler).

22. Zastąpić dwie funkcje StepLeft i StepRight jedną funkcją „LedStep", która będzie przesuwać punkt świetlny o jeden punkt w prawo lub w lewo w zależności od wartości przekazywanego do niej argumentu. Argument powinien by typu enum.
    Test: 9 kroków w prawo i 9 w lewo w nieskączońść (symulator i mikrokontroler).
    Usunąć funkcje StepLeft i StepRight z kodu.

23. Zmodyfikować funkcję LedStep tak, żeby nie używała zmiennych globalnych.
    Podpowiedź: „static".
    Test jak w poprzednim punkcie.

24. Napisać program, który będzie przesuwał punkt świetlny:
    - w prawo jeżeli naciśnięty jest BUTTON_1,
    - w lewo jeżeli naciśnięty jest BUTTON_2,
    - w ogóle jeżeli nie jest naciśnięty żaden przycisk.

    Uwaga: należy użyć instrukcji *case.*

25. Napisać funkcje *LedStepLeft(void)* i *LedStepRight(void)* wykorzystujące L*edStep.*

    Test jak w poprzednim punkcie

26. Upewnić się, że kod programu jest zgodny z konwencjami opisanymii w dokumencie "ProgramObowiazkowy.pdf"

    Zademonstrować działanie programu prowadzącemu .

5 Podział na moduły

5.1 Wprowadzenie

Wraz z dodawaniem funkcjonalności do program zaczyna wzrastać ilość funkcji I zmiennych. Trzymanie wszystkich w jednym pliku na dłuższą metę prowadzi do trudności z utrzymaniem przejrzystości kodu. Dlatego zaleca się dzielenie programu na mniejsze części.
Przykładowo ostatnią wersję programu można podzielić na trzy moduły: *main*, *led* i *keyboard*.

W modułach *led* i *keyboard* znajdować się będą funkcje i definicje związane z led-ami i klawiaturą. W module *main* znajdować się będzie funkcja główna część programu wraz z funkcją *main*.
Każdy moduł (oprócz *main*) powinien się składać z pliku źródłowego z rozszerzeniem „.c" oraz pliku nagłówkowego z rozszerzeniem „.h". Aktualny program będzie się składał więc z plików: main.c, led.c, led.h, keyboard.c, keyboard.h.

Należy dążyć do tego, aby w każdym module wyodrębnić tylko jedną konkretną funkcjonalność mikrokontrolera oraz, jeżeli to możliwe, aby moduły nie przenikały się między sobą (aby rejestry lub bity konfiguracyjne mikrokontrolera ustawiane w jednym module nie były modyfikowane przez inny). Znacznie upraszcza to wykorzystanie już raz napisanych modułów w innych projektach.

W plikach „.h" (nagłówkowych) powinny się znajdować deklaracje funkcji (nagłówek funkcji) oraz deklaracje typów danych stworzonych przez użytkownika:

- Deklaracja funkcji (nagłówek) zawiera nazwę funkcji oraz informację jak się daną funkcją posługiwać (argumenty wejściowe i wyjściowe), nie zawiera natomiast ciała funkcji, czyli informacji jak funkcja działa.
- Deklaracja typów danych stworzonych przez użytkownika, np. enum lub struct i używanych w nagłówkach zdeklarowanych funkcji.

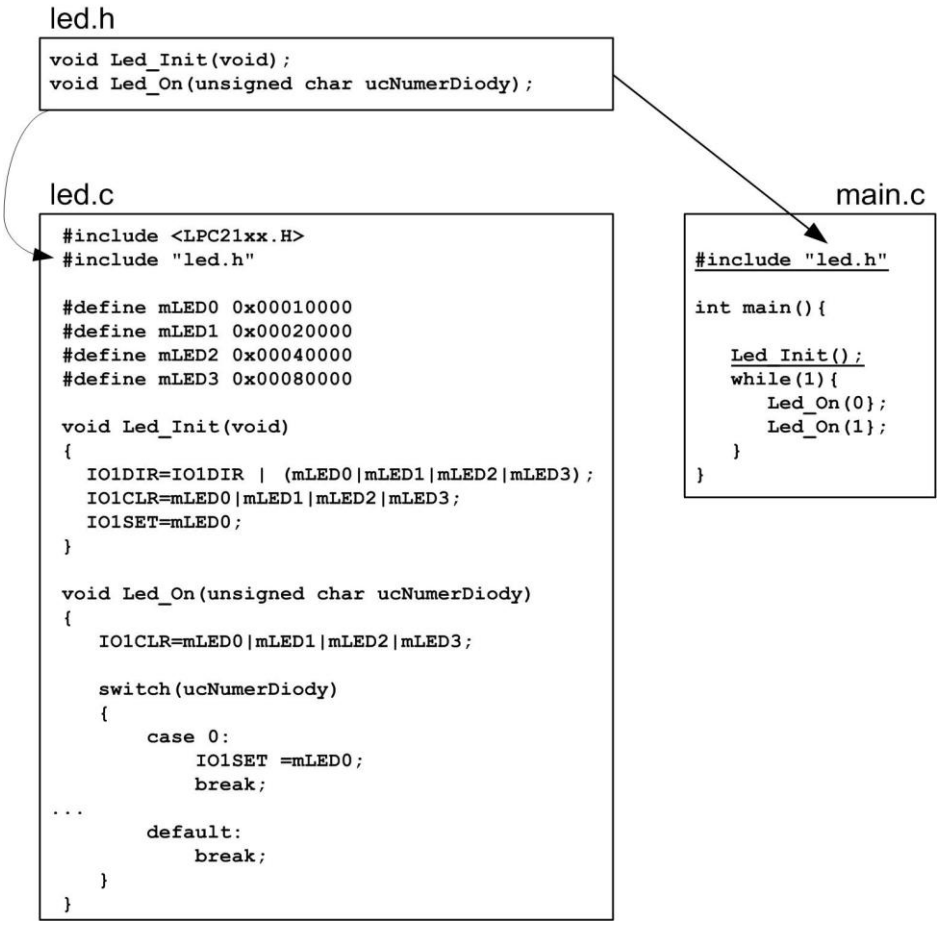
UWAGA: W pliku nagłówkowym umieszczamy deklaracje **tylko** tych funkcji i typów, które zamierzamy wykorzystać w innych modułach.

W plikach „.c" powinny się znajdować:

- dyrektywa dołączająca plik nagłówkowy o takiej samej nazwie jak plik.c (#include <xx.h>)
- informacje konieczne do działania funkcji, np.:
  - nazwy rejestrów mikrokontrolera (#include <LPC21xx.H>)
  - pliki nagłówkowe modułów zawierających funkcje, z których korzystamy w bieżącym module (#include <yy.h>)
  - definicje masek bitowych
- definicje zmiennych globalnych
- definicje funkcji (nagłówek „funkcja()" oraz ciało „{ }").

Przykład dla modułu „*Led*”:

Załóżmy, że moduł *led* składa się z dwóch funkcji - *LedInit i LedOn,* z których chcemy skorzystać w module *main*. Zawartość plików led.h led.c i main.c pokazano poniżej:



UWAGA: Jeżeli chcemy korzystać z funkcji znajdujących się w jakimkolwiek module należy plik „.c” tego modułu dołączyć do projektu.

5.2 Ćwiczenia

1. Wyodrębnić w ostatniej wersji programu (ostatnie ćwiczenie z GPIO) dwa moduły: *led* i *keyboard*.
   - Moduł *led* powinien udostępniać użytkownikom tylko funkcje: *LedInit, LedStepLeft, LedStepRight.* Podpowiedź: plik nagłówkowy.
   - Po podziale program powinien się kompilować i działać poprawnie..

2. Jeżeli maska bitowa zawiera tylko jedną jedynkę można uprościć zapis podając zamiast całej liczby w kodzie heksadecymalnym pozycję bitu w kodzie dziesiętnym.
   Zamiast „*#define LED2_bm 0x00020000*” należy wpisać „*#define (1 << 17)*” – wyrażenie po *define* koniecznie w nawiasach! (Dlaczego?)
   Przerobić wszystkie stałe i wszystkie wpisy do rejestrów tak, aby używać definicji określającej pozycje bitu.
   Upewnić się, że program kompiluje i wykonuje się poprawnie oraz zaprezentować go prowadzącemu.

6 Automaty

1. Napisać program, który będzie na przemian co 0.25s przesuwał punkt świetlny w prawo/lewo.

   W kodzie programu może być tylko jedno odwołanie do funkcji *Delay*.

   ROZWIAZANIE A: while(1){

   Delay(500);

   Led_StepRight();

   Delay(500);

   Led_StepLeft(); }

   To rozwiązanie nie jest poprawne, ponieważ wywołanie funkcji *Delay* ma miejsce dwukrotnie.

   ROZWIAZANIE B część 1:

   Można zauważyć, że w omawianym programie diody mogą znajdować się tylko w dwóch stanach – punkt świetlny przesówa się albo lewo albo w prawo. Deklarujemy zmienną, która reprezentuje stan diod – „*enum eLedState {LED_LEFT, LED_RIGHT}*" – i inicjalizujemy jej wartość na *LED_LEFT*, ponieważ po uruchomieniu programu świeci właśnie ta dioda. Następnie wstawiamy do pętli głównej fragment kodu, który w każdej iteracji pętli zmienia stan diod na przeciwny (LED_RIGHT -> LED_LEFT -> LED_RIGHT ->…):

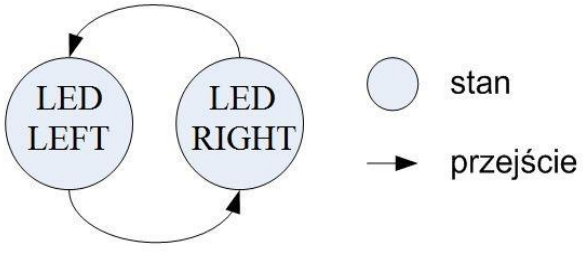
```
enum LedState{LED_LEFT, LED_RIGHT};
enum LedState eLedState = LED_LEFT;

while(1){
   switch(eLedState){

      case LED_LEFT:
      eLedState = LED_RIGHT;
      break;

      case LED_RIGHT:
      eLedState = LED_LEFT;
      break;
   }
}
```

   Działanie powyższego kodu można zilustrować za pomocą diagramu stanu.



   Zadanie: Wstawić kod do kompilatora i przy pomocy debuggera sprawdzić jego działanie, obserwując wartość zmiennej *eLedState*.

ROZWIAZANIE B część 2:

Poprzednia wersja program nie zmieniała rzeczywistego stanu diod oraz nie wprowadzała wymaganego opóźnienia między przełączeniami. Brakującą funkcjonalność wstawiono w wersji ostatecznej:

```
enum LedState{LED_LEFT, LED_RIGHT};
enum LedState eLedState = LED_LEFT;

while(1){
   switch(eLedState){

      case LED_LEFT:
      LedStepLeft();
      eLedState = LED_RIGHT;
      break;

      case LED_RIGHT:
      LedStepRight();
      eLedState = LED_LEFT;
      break;
   }
}
```

Działanie programu sprawdzić na symulatorze i mikrokontrolerze.

Należy zauważyć, że zmiany stanu zachodzą bezwarunkowo, tzn. gdy jesteśmy w stanie LED_LEFT to zawsze przechodzimy do LED_RIGHT.

2. Napisać program, który będzie przesuwał punkt świetlny 3 kroki w jedną stronę, a potem 3 kroki w drugą.

   WYMAGANIA:

   - Program powinien mieć taką samą strukturę jak program z poprzedniego punktu.
   - Program powinien mieć sześć stanów (STATE0, STATE1, …, STATE5).

   Program sprawdzić na symulatorze i zestawie uruchomieniowym.

Zasady 1: We wszystkich następnych programach z tego rozdziału:

- **pętla główna powinna składać się tylko z instrukcji switch i wywołania funkcji Delay,**
- **zmienna reprezentująca stany powinna być typu enum i mieć opisową nazwę.**
- **narysować diagram stanu (w dowolnej aplikacji lub na kartce), pamiętać o przejsciach wsobnych i opisach strzałek (przy przejściach wsobnych opsów nie dajemy)**
- **testy wykonywać tylko na zestawie uruchomieniowym, symulatora używać jeśli program nie działa i trzeba ustalić przyczynę niedziałania**

3. Przerobić poprzedni program tak, aby zmienna e*LedState* przyjmowała tylko dwa stany:
   - stany powinny reprezentować przesuwanie się w lewo i w prawo (dobrać odpowiednie nazwy),
   - użyć zmiennej pracującej jako licznik przesunięć.
   - narysować diagram stanu, opisać strzałki tj. pod jakim warunkiem zachodzi przejście między stanami.

Zasady 2: We wszystkich następnych programach:

- **poszczególne *case'y* mogą składać się tylko z:**
  - **instrukcji warunkowej if() {} else if () {} else ... ,**
  - **instrukcji *break.***
- **W szczególności przypisania do zmiennej reprezentującej stan oraz pozostałych zmiennych lokalnych powinny znajdować się wewnątrz instrukcji warunkowej.**
- **W kodzie należy odzwierciedlić również przejścia wsobne, czyli pozostawanie w tym samym stanie.**

4. Przerobić program tak, aby po uruchomieniu punkt świetlny przesunął się 3 kroki w prawo i się zatrzymał. Następnie zmodyfikować program w taki sposób, aby po starcie programu punkt świetlny stał w miejscu, a po każdym naciśnięciu *S0* przechodził 3 kroki w prawo i się zatrzymywał.
   Automat powinien składać się z 2 stanów.

5. Napisać program, który będzie przesuwał punkt świetlny w prawo aż do momentu naciśnięcia przycisku S0, co spowoduje zatrzymanie się punktu. Naciśniecie przycisku S1 ma spowodować ponowne przesuwanie się punktu świetlnego w prawo.

6. Napisać program, w którym naciśniecie przycisku:
   1. S0 spowoduje przesuwanie się punktu świetlnego w lewo,
   2. S1 spowoduje zatrzymanie się punktu świetlnego,
   3. S2 spowoduje przesuwanie się punktu świetlnego w prawo.

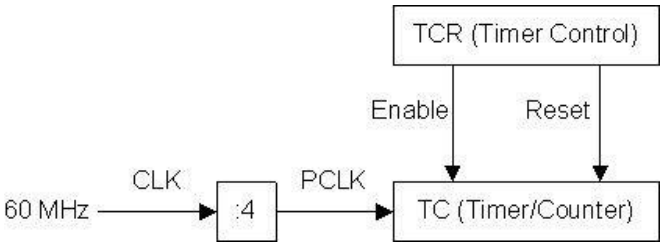
   Nie ma możliwość na przejście bezpośrednio od przesuwania w lewo do przesuwania w prawo i vice versa.
   Automat powinien składać się z trzech stanów.
   Automat powinien być wywoływany 10 razy na sekundę.

7. Upewnić się, czy programy z punktów 3-6 są napisane zgodne z konwencjami, a następnie poprosić prowadzącego o sprawdzenie.

7 Układ czasowo-licznikowy

7.1 Opis

W systemach mikroprocesorowych jednym z najbardziej podstawowych zagadnień jest pomiar czasu, ponieważ nawet proste systemy muszą wykonywać poszczególne sekwencje działań w ściśle określonych przedziałach czasu. Z tego powodu układy czasowo-licznikowe (ang. Timer), tak samo jak porty wejścia/wyjścia, są jednym z podstawowych układów peryferyjnych. Zadaniem Timera jest zliczanie impulsów podawanych na jego wejście. W zastosowanym mikrokontrolerze źródłem taktowania Timera jest zegar peryferiów PCLK. Jego częstotliwość stanowi jedną czwartą częstotliwości taktowania rdzenia procesora, która wynosi 60MHz (rysunek 7.1).



Rys 7.1 Uproszczony schemat układu licznika

Mikrokontroler LPC2129 posiada dwa identyczne Timery: TIMER0 i TIMER1. Tak jak w przypadku innych peryferiów, komunikacja z nimi odbywa się za pomocą rejestrów. Rejestrem reprezentującym bieżący stan Timera (aktualnie zliczona liczbę) jest rejestr „Timer Counter" - **TC**. Do włączania/wyłączania Timera służy bit „Counter Enable", a do jego zerowania - bit „Counter Reset". Oba bity znajdują się w rejestrze „Timer Control Register" **TCR** (user manual, str. 217).

7.2 Ćwiczenie 1

Dotychczas do zapewnienia odpowiednich odstępów czasowych w programach używano funkcji *Delay()* realizującej swoje zadanie za pomocą pętli opóźniającej. Z różnych powodów taki sposób nie gwarantuje opóźnień o pożądanym i stałym czasie trwania. Ponadto przekompilowanie tej samej funkcji *Delay()* za pomocą różnych kompilatorów może powodować zmianę opóźnień, co czyni koniecznym zmianę wartości stałej w pętli opóźniającej. Aby uniknąć wymienionych problemów można wykorzystać *Timer*.

W tym celu należy dodać do projektu nowy moduł – **timer** (plik .c oraz plik .h), a w nim:

- stworzyć funkcję *InitTimer0(void)*, której jedynym zadaniem będzie włączenie *Timera0*.
- stworzyć funkcję *WaitOnTimer0(unsigned int uiTime)*, która: o zresetuje *Timer0*, o poczeka do momentu osiągnięcia przez *Timer0* wartości podanej w argumencie wywołania funkcji.

UWAGI:

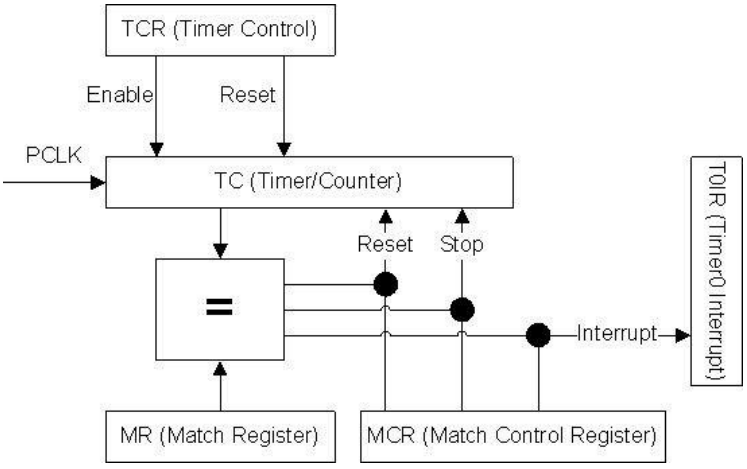
- argument funkcji *WaitOnTimer0* powinien być wyrażony w mikrosekundach; częstotliwość taktowania licznika została podana w opisie układu czasowo-licznikowego,
- do oczekiwania na osiągnięcie przez *Timer* oczekiwanej wartości można użyć pętli *while,*
- do ustawiania bitów w rejestrze konfiguracyjnym Timera należy używać masek bitowych, których nazwy należy zaczerpnąć ze specyfikacji „lpc21xx_user_manual".
- W pliku nagłówkowym .h należy zadeklarować obie funkcje.

Działanie funkcji przetestować realizując funkcjonalność programu testującego z punktu GPIO 20. Jako funkcji opóźniającej użyć *WaitOnTimer0*.

7.3 Blok porównujący

Działanie funkcji *WaitOnTimer0* polegało na każdorazowym resecie Timera oraz sukcesywnym porównywaniu jego stanu z zadaną liczbą zależną od pożądanego opóźnienia. W celu realizacji powyższych funkcjonalności można

wykorzystać bloki porównujące Timera. Działanie bloku porównującego (rysunek 7.2) polega na porównywaniu zawartości licznika Timera (TC) z wartością zapisaną w odpowiednim rejestrze MR0…MR3. W przypadku, gdy wartości w obu rejestrach będą identyczne, w zależności od konfiguracji, licznik może zostać zatrzymany, wyzerowany lub może zostać zapalona flaga przerwania (pojedynczy bit w rejestrze przerwań „T0IR").



Rys 7.2 Uproszczony schemat bloku porównującego

Przykład: jeżeli chcemy żeby licznik automatycznie resetował się po osiągnięciu pewnej wartości, należy tę wartość wpisać do rejestru porównującego (MR0 – Match Register 0) oraz ustawić bit „Reset on MR0" w rejestrze Match Control Register - MCR. Ponadto, jeżeli chcemy żeby osiągniecie przez licznik wartości równej MR0 powodowało ustawienie flagi przerwania, należy w rejestrze MCR ustawić bit „Interrupt on MR0".

7.4 Ćwiczenie 2

W celu wykorzystania układów porównujących do generowania opóźnień, należy do modułu timer dodać dwie kolejne funkcje:

- InitTimer0Match0(unsigned int iDelayTime), której zadaniem będzie:
  - ustawienie układu porównującego w sposób podany w przykładzie powyżej (czas w mikrosekundach),
  - wyzerowanie oraz włączenie Timera0.
- WaitOnTimer0Match0(), której zadaniem będzie:
  - oczekiwanie na osiągnięcie przez Timer0 odpowiedniej wartości, czyli na ustawienie flagi przerwania przez układu porównujący,
  - wyzerowanie flagi przerwania.

Program testowy taki sam jak w poprzednim punkcie, z opóźnieniem zrealizowany za pomocą WaitOnTimer0Match0.

8 Przerwania

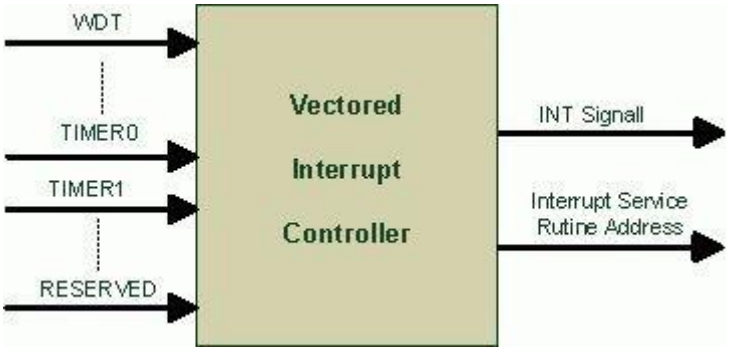
8.1 Opis

Mikrokontroler LPC2129 może przyjmować przerwania z 32 dwóch źródeł (kanałów). Mogą to być przerwania z pochodzące z zewnątrz (np. przerwanie spowodowane zboczem narastającym na jednym z pinów wejściowych – INT0-INT3) lub z jednego z wewnętrznych bloków, takich jak Timery, UART-y (układy transmisji szeregowej) lub układ Watchdog (WDT). Przykładowo, przerwanie z Timera może być spowodowane zrównaniem się zawartości licznika z wartością w rejestrze MATCH. W tabeli 8.1 zamieszczono fragment listy możliwych źródeł przerwań oraz przyporządkowanych do nich numerów kanałów.

Blok	VIC Channel #
WDT	0
-	1
ARM Core	2
ARM Core	3
TIMER0	4
TIMER1	5
UART0	6
UART1	7

Tabela 8.1 Przyporządkowanie poszczególnych źródeł przerwań do kanałów wejściowych kontrolera przerwań.

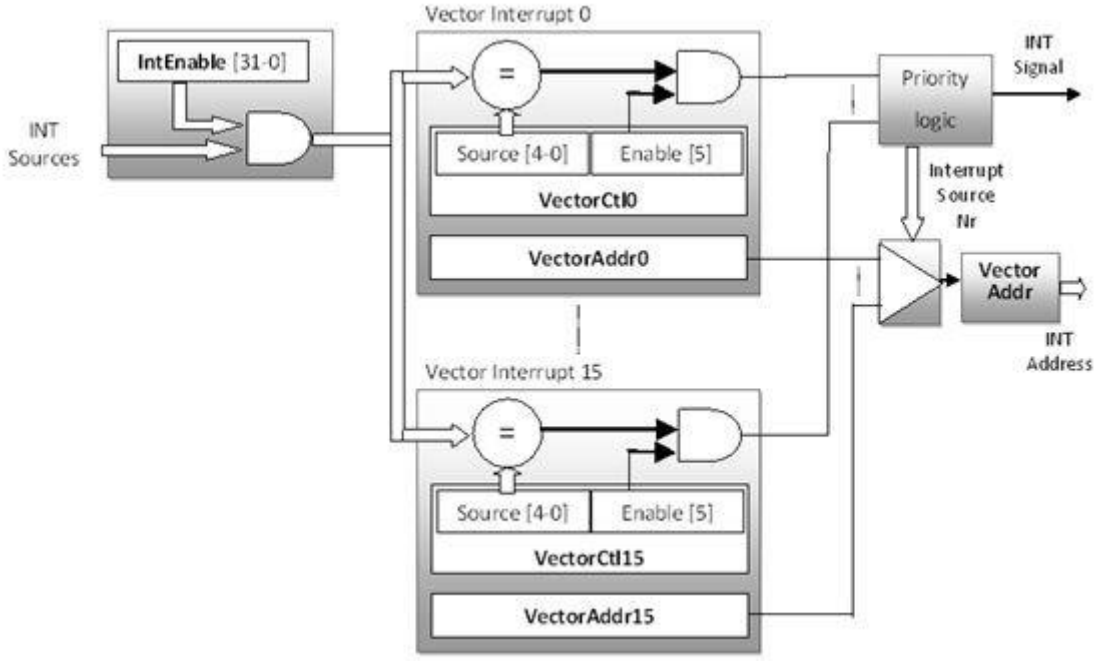
Zadaniem kontrolera przerwań VIC (Vectored Interrupt Controller) (rysunek 8.1) jest pobieranie zgłoszeń przerwań od poszczególnych modułów kontrolera, a następnie przydzielanie im odpowiedniego adresu procedury obsługi przerwania oraz, jeśli więcej niż jeden układ zgłosi przerwanie jednocześnie, decydowanie o tym, które z nich zostanie obsłużone w pierwszej kolejności (priorytetowanie).



Rys. 8.1 Schemat funkcjonalny układu kontroli przerwań

Schemat wewnętrzny układu VIC pokazano na rysunku 8.2. Na wejściu układu znajduje się układ, który pozwala na blokowanie/odblokowywanie przerwań z poszczególnych kanałów (patrz. tabela 8.1). Konfiguracji układu dokonuje się za pośrednictwem rejestru *IntEnable.* Poszczególne bity w rejestrze odpowiadają poszczególnym kanałom.

Główna częścią układu kontroli przerwań jest 16 slotów (VectorInterrupt0..15). Każdy slot może być włączany/wyłączony za pomocą bitu *Enable* w rejestrze *VectorCtl.* Każdy slot może być ustawiony na jeden z 32 kanałów (źródeł przerwań) za pomocą bitów *Source* w rejestrze *VectorCtl*. Każdemu slotowi można przypisać adres procedury obsługi przebrania wpisując go do rejestru *VectorAddr*.



Rys. 8.2 Uproszczony schemat układu kontroli przerwań

Działanie układu jest następujące: Po zgłoszeniu przerwania przez jeden z kanałów kontroler sprawdza, czy któryś z włączonych slotów jest ustawiony na ten kanał. Jeżeli tak, to kontroler pobiera z tego slotu adres procedury obsługi przerwania (rejestr *VectorAddr*) i uruchamia ją. Jeżeli więcej niż jeden kanał zgłosi przerwanie, to kontroler wybiera ten, którego slot ma niższy adres.

8.2 Ćwiczenia

Ć**wiczenie 1** Usunąć zawartość pliku *main.c*. Skopiować do pliku *main.c* zawartość pliku *Timer1Interrupts.c*. Program składa się z:

- funkcji main,
- funkcji inicjalizującej timer i przerwania (*Timer1InterruptsInit*), ● procedury obsługi przerwania timera (*Timer1IRQHandler*).

Zapoznać się z zawartością programu (wiersz po wierszu). W razie niejasności, sprawdzić w dokumentacji procesora.

Program przetestować na symulatorze sprawdzając czy wchodzi co milisekundę do procedury obsługi przerwania.

Ć**wiczenie 2** Przerobić program tak, żeby pracował z timerem 0, a procedura obsługi przerwania była wywoływana 4 razy na sekundę.
Uwaga: W pierwszej kolejności zamienić w nazwach wszystkich funkcji „Timer1" na „Timer0". Następnie zmienić nazwy odpowiednich rejestrów i zdefiniować maski bitowe.
Program przetestować na symulatorze i zestawie uruchomieniowym.

UWAGA: przed wykonaniem następnego ćwiczenia należy zapoznać się z podstawami deklarowania i użycia wskaźników na funkcje.

Ć**wiczenie 3** Obecna wersja funkcji inicjalizującej timer pozwala na ustawienie okresu przerwania. Nie pozwala natomiast na ustawianie funkcji wywoływanej w przerwaniu (obecnie LedStepRight). Zmiana programu tak, aby w przerwaniu wywoływana była np. LedStepLeft wymaga modyfikacji kodu funkcji obsługi przerwania.
Można to zmienić wykorzystując fakt, że w języku C można wywoływać funkcję za pośrednictwem wskaźnika. Celem ćwiczenia jest taka modyfikacja programu, aby podczas inicjalizacji timera można było ustawiać funkcję, która ma być wywoływana w przerwaniu.

W pierwszej kolejności należy stworzyć w pliku main zmienną globalną *ptrTimer0InterruptFunction* będącą wskaźnikiem na funkcję. Należy założyć, że funkcja nie przyjmuje żadnych argumentów i nic nie zwraca.
Następnie należy przerobić funkcję *Timer0InterruptsInit* tak, aby oprócz okresu przerwania przyjmowała jako argument wskaźnik na funkcje np. *ptrInterruptFunction* (takiego samego typu jak utworzona wcześniej zmienna globalna). W funkcji powinno następować przepisanie *ptrInterruptFunction* do zmiennej globalnej *ptrTimer0InterruptFunction.* Przykładowe wywołanie funkcji inicjalizującej powinno wyglądać następująco:

Timer0InterruptsInit(250000,&LedStepLeft);

Na koniec należy zastąpić w funkcji obsługi przerwania wywołanie funkcji LedStepLeft wywołaniem funkcji, na którą wskazuje *ptrTimer0InterruptFunction.*

Test programu przeprowadzić na symulatorze i zestawie uruchomieniowym.

Ćwiczenie 4 Utworzyć moduł *timer_interrupts:*

- Utworzyć plik *timer_interrupts*.c, przenieść do niego definicje wszystkich funkcji (oprócz *main*) oraz wszystko co jest potrzebne do ich kompilacji (pamiętać o wskaźniku na funkcje).
- Dołączyć *timer_interrupts*.c do projektu (patrz. rozdział 1 podpunkt 1.1.6).

- Skompilować *timer_interrupts*.c.

- Utworzyć plik headera (*timer_interrupts*.h), wstawić do niego deklarację funkcji inicjalizującej timer, a następnie dołączyć plik do *timer_interrupts*.c i do main.c
- Usunąć z pliku main.c „#include <LPC21xx.H>”
- Skompilować *timer_interrupts*.c, main.c a następnie cały projekt

Test programu przeprowadzić na symulatorze i zestawie uruchomieniowym.

Ćwiczenie 5

W pliku main.c stworzyć funkcje Automat() zawierającą automat z punktu 6, rozdział 6..

UWAGA: Nie deklarować ż**adnych dodatkowych zmiennych globalnych.**

Skompilować projekt.

Zmienić inicjalizację timera tak, aby wywoływała automat z częstotliwością 50 Hz.

Test programu przeprowadzić na zestawie uruchomieniowym.

Poprosić prowadzącego o sprawdzenie.

9 Sterownik serwomechanizmu

9.1 Opis urządzenia

Płytka serwomechanizmu składa się z silnika krokowego z zamocowana na jego osi tarczą oraz detektora w postaci transoptora refleksyjnego umieszczonego pod tarczą. Podawanie jedynek kolejno na wejścia W0 do W3 powoduje obrót osi silnia a co za tym idzie umieszczonej na nim tarczy. Zadaniem detektora jest wykrycie obecności (‘0’) „w polu widzenia” markera umieszczonego nad i pod tarczą.

W celu sprawdzenia poprawnego działania detektora należy połączyć płytkę serwomechanizmu z komputerem a następnie obrócić tarczę tak aby marker znalazł się nad detektorem. Powinno to spowodować zaświecenie się diody LED. Podczas testu płytka serwomechanizmu powinna być odłączona od płytki z mikrokontrolerem.

Do sterowania silnika należy używać pinów, które są używane do sterowania LED-ów, natomiast do odczytu detektora należy używać pinu 10 portu 0.

9.2 Ćwiczenia

Ćwiczenie 1

Celem ćwiczenia jest modyfikacja automatu z poprzedniego programu tak, aby zawsze po starcie programu serwomechanizm wykonywał procedurę kalibracji tj. ustawił tarczę w tej samej pozycji.

W tym celu, w pierwszej kolejności należy stworzyć dwie funkcje:

- void DetectorInit(), której zadaniem jest ustawienie pinu podłączonego do detektora (P0.10) jako wejściowego.
- enum DetectorState eReadDetector(), której zadaniem jest zwracanie stanu ACTIVE jeżeli marker na tarczy znajduje się nad detektorem lub INACTIVE w przeciwnym przypadku.

Następnie, do poprzedniego automatu należy dodać stan *CALLIB*. W stanie tym silnik powinien obracać się przeciwnie do ruchu wskazówek zegara do momentu przysłonięcia przez płetwę szczeliny w transoptorze.

Działanie programu sprawdzić na symulatorze i mikrokontrolerze podłączonym do serwomechanizmu.

Ć**wiczenie 2**

Celem ćwiczenia jest modyfikacja automatu tak, aby ustawiał lustro w pozycji określonej przez zawartość zmiennej *DesiredPosition.* Przykładowo, jeżeli gdziekolwiek w programie do zmiennej *DesiredPosition* zostanie wpisana wartość 10, to oś silnika powinna obrócić się o dziesięć kroków w prawo. Jeżeli następnie do zmiennej *DesiredPosition* zostanie wpisana wartość 7 to *oś* silnika powinna obrócić się o trzy kroki w lewo itd.

Najpierw należy dodać do programu zmienną globalną „*sServo*", przechowującą aktualny stan automatu oraz serwomechanizmu:

```
enum ServoState {CALLIB, IDLE, IN_PROGRESS};

struct Servo
{
  enum ServoState eState;    unsigned int
uiCurrentPosition;    unsigned int
uiDesiredPosition;
};  struct Servo sServo;
```

Automat powinien składać się z trzech stanów: **CALLIB**, **IDLE**, **IN_PROGRESS** oraz korzystać ze zmiennych ***CurrentPosition*** i ***DesiredPosition***. ***CurrentPosition*** reprezentuje bieżącą pozycję, a ***DesiredPosition*** pozycję, którą Serwo ma osiągnąć.

W stanie **CALLIB** następuje ustawienie się serwomechanizmu w pozycji zerowej oraz wyzerowanie ***CurrentPosition*** i ***DesiredPosition***.

Następnie automat przechodzi do stanu **IDLE**, w którym pozostaje dopóki ***CurrentPosition*** równa się ***DesiredPosition***.

Jeżeli ***DesiredPosition*** ulegnie zmianie *(**CurrentPosition*** != ***DesiredPosition)*** automat przechodzi do stanu **IN_PROGRESS**. W stanie **IN_PROGRESS** silnik wykonuje kroki w kierunku zależnym od tego, czy ***CurrentPosition*** < ***DesiredPosition*** *czy* ***CurrentPosition*** > ***DesiredPosition*** oraz odpowiednio aktualizuje (zmniejsza lub zwiększa) wartość ***CurrentPosition***.

Przejście ze stanu **IN_PROGRESS** do **IDLE** następuje w momencie zrównanie się zawartości zmiennych ***CurrentPosition*** i ***DesiredPosition.***

Program przetestować wstawiając do pętli głównej fragment kodu, który po naciśnięciu przycisku S1 spowoduje kalibracje serwomechanizmu, a po naciśnięciu przycisków S2, S3, S4 będzie powodował ustawienie serwomechanizmu w pozycjach odpowiednio 12, 24, i 36.

Ćwiczenie 3

Stworzyć funkcje:

- void ServoInit(unsigned int uiServoFrequency) – wprowadza automat w stan kalibracji, inicjalizuje Ledy, podłącza automat serwomechanizmu do przerwań timera;
- void ServoCallib(void) – wprowadza automat w stan kalibracji;
- void ServoGoTo(unsigned int uiPosition) – wymusza obrót osi silnika do zadanej pozycji;

UWAGA: Należy zwrócić uwagę, że zmienna *ServoFrequency* wyraża częstotliwość, podaną w Herzach, natomiast do inicjalizacji timera wymagany jest okres wyrażony w mikrosekundach.

Przerobić program tak, aby używał powyższych funkcji.

Przetestować program.

Ćwiczenie 4

Utworzyć moduł „servo” analogicznie jak utworzono moduł „timer_interrupts”.

Z kodu serwomechanizmu w main.c powinny pozostać tylko wywołania funkcji z ćwiczenia 3.

Przetestować program.

Poprosić prowadzącego o sprawdzenie.

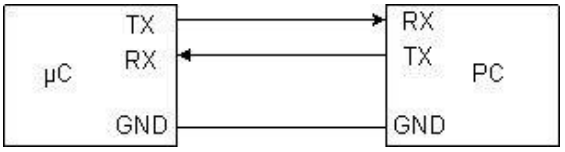
10 Układ Asynchronicznej Transmisji Szeregowej

10.1 Opis standardu

RS232C jest asynchronicznym interfejsem szeregowym, pracującym w trybie „Full-duplex”. Został zaprojektowany z myślą o transmisji danych między dwoma urządzeniami na niewielkie odległości (do 15 m w warunkach przemysłowych) z prędkością do 20 kb/s (obecnie najczęściej stosuje się prędkości do 115 kb/s). RS232C pierwotnie służył do komunikacji między terminalem znakowym a modemem.

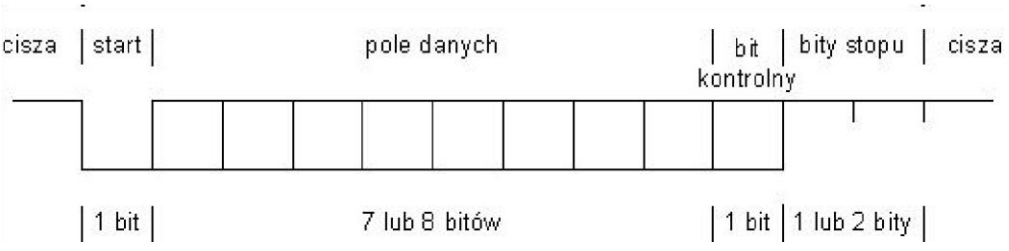
Standard RS232C określa zbiór zasad przeprowadzania transmisji danych między dwoma urządzeniami z wykorzystaniem 8-u sygnałów. Dwa z nich (Tx i Rx) służą bezpośrednio do transmisji bitów danych, reszta (DTR, RI, DSR, RTS, CTS, DCD) jest wykorzystywana do sterowania transmisją (nawiązywanie, podtrzymywanie, blokowanie, przerywanie). Sygnały sterujące były używane dopóki moc obliczeniowa oraz pojemności komputerów były niewielkie i często zachodziła konieczność wstrzymywania transmisji przez odbiornik. Obecnie w większości stosuje się rozwiązania korzystające tylko z sygnałów „Tx” i „Rx”.

Poziomy napięć dla logicznego „0” i „1” wynoszą odpowiednio +12V i –12V. Zakres poprawnego odbioru jest jednak szeroki i wynosi +3...15V dla stanu „0” i –15...-3V dla stanu „1”. Schemat połączenia dwóch urządzeń za pomocą interfejsu RS232C pokazano na rysunku 10.1.



Rys. 10.1 Schemat połączenia dwóch urządzeń za pomocą interfejsu RS232C.

Rysunek 10.2 pokazuje ramkę transmisji pojedynczego słowa danych. Transmisja zaczyna się od bitu „START”, następnie przesłanych zostaje 7 lub 8 bitów danych, po których może (ale nie musi) następować bit kontrolny (bit parzystości). Transmisje kończą jeden, półtora lub dwa bity stopu. Należy zwrócić uwagę, że odbiornik musi z góry znać parametry transmisji, tj. ilość bitów danych, obecność bitu kontrolnego, ilość bitów stopu oraz prędkość transmisji.



Rys. 10.2 Ramka transmisji pojedynczego słowa danych w standardzie RS232.

Jak wspomniano na początku rozdziału, interfejs RS232C został zaprojektowany do komunikacji między terminalem znakowym a modemem. Dzięki dużej popularności interfejsu został on zaimplementowany w komputerach IBM PC,

a potem w większości komputerów klasy PC. Również większość rodzin mikrokontrolerów posiada wersje z wbudowaną jednostką zdolna komunikować się w standardzie RS232C (chodzi tu o okrojoną wersję standardu korzystającą tylko z linii Tx i Rx). Omawiany interfejs jest wykorzystywany zarówno do komunikacji między mikrokontrolerami, jak między mikrokontrolerem a komputerem PC. Ponieważ poziomy logiczne mikrokontrolerów są zwykle związane z ich napięciem zasilania (Vcc) i wynoszą 0V dla logicznego „0" i Vcc dla logicznej „1" i nie są zgodne z poziomami interfejsu RS232C, konieczne jest stosowanie dodatkowych konwerterów w postaci gotowych układów scalonych.

10.2 Opis modułu UART

W mikrokontrolerach jednostkę używaną do komunikacji zgodną ze standardem RS232 nazywa się zwykle UART (ang. *Universal Asynchronous Receiver and Transmitter*, także USART - *Universal Synchronous and Asynchronous Receiver and Transmitter*) - obwód zintegrowany używany do asynchronicznego przekazywania i odbierania informacji poprzez port szeregowy. Zawiera on konwerter typu *parallel-to-serial* – służący do konwersji danych przesyłanych z komputera – oraz *serial-to-parallel* – do konwersji danych przychodzących do komputera poprzez port szeregowy. *UART* zawiera także bufor do tymczasowego gromadzenia danych w przypadku szybkiej transmisji. Przed rozpoczęciem transmisji danych za pomocą układu UART należy skonfigurować parametry transmisji, takie jak kształt ramki (ilość bitów danych, parzystości, stopu) oraz prędkość transmisji.

Tak jak w przypadku wykorzystania timera, można zastosować dwie techniki do współpracy z UART-em.

Pierwsza z nich zwana **pollingiem** polega na ciągłym sprawdzaniu status urządzenia/modułu. Przykładowo w przypadku timera było to ciągłe sprawdzanie flagi przerwania. W przypadku UART-a mogłoby to być ciągłe sprawdzanie flagi informującej o odebraniu znaku. Wadą tej techniki jest marnowanie mocy obliczeniowej procesora oraz brak gwarancji szybkiej reakcji na zdarzenie.

Drugą techniką współpracy programu z peryferiami jest technika oparta na **przerwaniach.** Przerwaniem jest sygnałem powodującym zapamiętanie bieżącego stanu procesora, wykonanie procedury obsługi przerwania, a następnie przywrócenie stanu oraz powrót do miejsca w programie wykonywanego przed nadejściem przerwania.

10.3 Ćwiczenia

Ćwiczenie 1

Stworzyć plik uart.c, wstawić do niego zawartość pliku UartRecieverOnInterrupts.c, dołączyć plik do projektu.

- W funkcji *UART_InitWithInt* uzupełnić brakujący fragment kodu, zaznaczony pytajnikami. Dotyczy on konfiguracji układu **PIN CONNECT BLOCK** (patrz – dokumentacja procesora), którego zadaniem jest konfigurowanie poszczególnych pinów do pożądanych funkcji. Należy skonfigurować pin P0.1 jako RxD (UART0).
- Uzupełnić brakujący komentarz w miejscu zaznaczonym pytajnikami.
- Sprawdzić czy przerwania Uarta i Timera nie pracują na tych samych slotach. Jeżeli tak, to poprawić.
- Skompilować plik uart.c.
- Wstawić do main-a inicjalizacje uart-a (*InitUART0WithInt*) – pamiętać o pliku nagłówkowym.
- Skompilować main.c, a następnie cały projekt.

Zadaniem programu jest odbieranie znaku ze złącza szeregowego i wpisywanie go do zmiennej cOdebranyZnak.
Program składa się z funkcji:

- inicjalizującej UART-a i „podpinającej" go do przerwań,
- funkcji obsługującej przerwanie, ● funkcji main.

W celu sprawdzenie działania programu należy:

- Wejść w tryb debugowania i wstawić break-point w linijce, w której odbywa się przepisywanie odebranego znaku do zmiennej *cOdebranyZnak.*
- Ustawić podgląd zmiennej *cOdebranyZnak* w okienku *Watch1.*
- Wysyłać znaki na UART0 używając okna terminala (patrz opis poniżej).
- Jeżeli program jest napisany poprawnie, każde wysłanie znaku do UARTA powinno powodować wejście do procedury obsługi przerwania (zatrzymanie na breakpincie).
- Usunąć breakpoint i ponownie uruchomić program.
- Wysłanie różnych znaków na UART-a powinno powodować zmianę zawartości zmiennej *cOdebranyZnak* (okno Watch1).

Wysyłanie znaku na UART symuluje się za pomocą znajdującego się w debuggerze okna terminala. Naciśniecie klawisz powoduje wpisanie do UART0 znaku odpowiadającego klawiszowi (w trybie debuggera: View -> Serial Windows -> UART #1).

Ćwiczenie 2

Przerobić kod programu tak, aby wysłanie znaków 1, 2, 3, 4 powodowało zapalenie diody 1, 2, 3, 4. Naciśnięcie przycisku „c" powinno powodować zgaszenie wszystkich LED-ów.

UWAGI:

- Zapalanie diod powinno odbywać się w pętli głównej, za pomocą funkcji *LedOn*.
- *„cOdebranyZnak",* żeby skorzystać ze zmiennej globalnej znajdującej się w innym module (innym pliku .c) należy ją zadeklarować dyrektywą „extern".
- Należy zablokować pracę serwomechanizmu żeby nie interferował z *LedOn oraz chwilowo „pod*łączyć*" moduł „led" do main-a*.

Program przetestować na symulatorze i mikrokontrolerze:

- Wgrać program do mikrokontrolera.
- **Przełączyć zestaw uruchomieniowy w tryb komunikacji (InstrukcjaObslugiArmEVM.pdf, rozdz. 3)**
- Uruchomić program Tera Term, wybrać komunikację po porcie szeregowym (Serial).
- Zresetować mikrokontroler (przycisk RST).
- Naciskać klawisze 1,2,3,4 w okienku terminala obserwując zachowanie się LEDów.
- (Nie należy oczekiwać literek w okienku terminala ponieważ okienko to wyświetla tylko znaki przychodzące; można to zmienić wybierając opcje *Echo*).

Ćwiczenie 3

Przystosować powyższy program do sterowania serwomechanizmem. Naciśnięcie przycisku C powinno powodować kalibrację, naciśnięcie przycisków 1,2,3 powinno powodować ustawienie serva na odpowiednio 90, 180 i 270 stopni od detektora krańcowego (jednemu obrotowi osi silnika odpowiada 200 kroków).

UWAGA: Nie usuwać fragmentu kodu odpowiedzialnego za sterowanie pozycją z przycisków.

Ćwiczenie 4

Przerobić poprzedni program tak, aby pojedyncze wysłanie znaku „1" powodowało pojedynczy obrót osi silnika **o** 90 stopni. Naciśnięcie przycisku C powinno powodować kalibrację.

11 Odbiór i dekodowanie łańcuchów znakowych

11.1 Wprowadzenie

W poprzednim podpunkcie sterowanie serwomechanizmem odbywało się za pomocą pojedynczych znaków przesyłanych za pomocą interfejsu RS232C. Od tego punktu sterowanie będzie odbywać się za pomocą komunikatów mających postać łańcuchów znakowych.

Poniższe ćwiczenia mają na celu:

- stworzenie i przetestowanie funkcji do odbierania łańcuchów znakowych,
- podłączenie jej do przerwań odbiornika układu UART, ● wykorzystanie dekodera komend.

UWAGA: Opisane poniżej zmienną i funkcje należy umieścić w module UART

11.2 Struktura danych

Znaki odbierane ze złącza szeregowego będą gromadzone **buforze odbiornika,** który ma posta ć struktury:

```
enum eRecieverStatus {EMPTY, READY, OVERFLOW};
  struct RecieverBuffer{ char cData[RECIEVER_SIZE];
  unsigned char ucCharCtr;
  enum eRecieverStatus eStatus;
};
```

cData służy do przechowywania odbieranego łańcucha znakowego.

ucCharCtr do iteracji po łańcuchu znakowym.

eStatus przechowuje status bufora:

EMPTY – nie odebrano jeszcze całego łańcucha, czyli nie przyszedł jeszcze znak terminatora,

READY – odebrano kompletny łańcuch znakowy, czyli odebrano znak terminatora,

OVERFLOW – odebrano więcej znaków niż może zmieścić się w buforze.

11.3 Funkcje

Zadaniem funkcji **void Reciever_PutCharacterToBuffer(char cCharacter)** jest wstawianie znaków do bufora odbiornika.

- Znaki powinny być wstawiane do tablicy cData do momentu napotkania znaku terminatora.
- W momencie napotkania znaku terminatora do bufora powinna zostać wpisana wartość NULL, a status bufora powinien zostać ustawiony na READY.
- Próba przepełnienia bufora, czyli wstawienia większej ilości znaków niż wynosi rozmiar bufora (RECIEVER_SIZE) powinna spowodować ustawienie statusu bufora na OVERFLOW oraz nie wstawienie znaku do tablicy.
- Do iterowaniu po tablicy cData użyć zmiennej cCharCtr. Należy przyjąć założenie, że przed pierwszym wywołaniem funkcji jest ona równa 0.

Zadaniem funkcji **enum eRecieverStatus eReciever_GetStatus(void)** jest odczyt status bufora odbiornika.

Zadaniem funkcji **void Reciever_GetStringCopy(char * ucDestination)** jest odczyt stringa znajdującego się w buforze odbiornika. Funkcja powinna skopiować stringa z RecieverBuffer.cData do tablicy ucDestination oraz ustawiać status bufora odbiornika na EMPTY.

11.4 Ćwiczenia

Ćwiczenie 1

Funkcję Reciever_PutCharacterToBuffer przetestować symulatorem przy pomocy następującej sekwencji komend:

```
//Zakładamy, że RECIEVER_SIZE równa się 4
Reciever_PutCharacterToBuffer ('k');
Reciever_PutCharacterToBuffer ('o');
Reciever_PutCharacterToBuffer ('d');
Reciever_PutCharacterToBuffer (TERMINATOR);

// w buforze powinien znaleźć się łańcuch znakowy “kod\0”,  // status powinien równać
się READY, a cCharCtr 0.
sRxBuffer.eStatus = EMPTY;
Reciever_PutCharacterToBuffer ('k');
Reciever_PutCharacterToBuffer ('o');
Reciever_PutCharacterToBuffer ('d');
Reciever_PutCharacterToBuffer ('1');
Reciever_PutCharacterToBuffer (TERMINATOR);
// status powinien równać się OVERFLOW
```

Ćwiczenie 2

Napisać program, który po odebraniu łańcuchów znakowych „callib”, „left” i „right” będzie odpowiednio: uruchamiał kalibrację, obracał oś serwomechanizmu do pozycji 50 i 150.

W tym celu należy:

- wstawić funkcję Reciever_PutCharacterToBuffer do przerwania odbiornika UARTA w miejsce gdzie odbieramy znak,
- w pętli głównej sukcesywnie sprawdzać status odbiornika (Reciever_GetStatus) i jeżeli przyjmie on wartość READY odczytać string z bufora odbiornika (Reciever_GetStringCopy) do pomocniczej tablicy,
- porównać odczytany string ze stringami „callib”, „left” i „right” i wpisać odpowiednia wartość do DesiredPosition.

UWAGI:

Nie modyfikować funkcji Reciever_PutCharacterToBuffer.

Nie odwoływać się bezpośrednio do bufora odbiornika.

Ćwiczenie 3

Napisać program reagujący na komendy „callib” i „goto 0xXXXX”.

„callib” – kalibruje serwomechanizm

„goto” - ustawia serwomechanizm w pozycji określonej przez liczbę znajdującą się po komendzie.

UWAGI:

Należy stworzyć dwa moduły „string” i „command_decoder”.

Należy zmodyfikować listę słów kluczowych.

W pętli głównej należy:

- sprawdzać status bufora odbiornika, jeżeli jest READY odczytać zwartość bufora do tablicy pomocniczej o rozmiarze takim samym jak rozmiar bufora odbiornika,
- zdekodować odebrany łańcuch ,
- jeżeli ilość tokenów jest różna od zera i pierwszy token jest słowem kluczowym to, w zależności od komendy (pierwszy token), uruchomić kalibracje albo wpisać odpowiednią wartość (drugi token) do *uiDsiredPosition* (użyć instrukcji switch/case).

12 Wysyłanie łańcuchów znakowych

Uwaga: Opisane poniżej strukturę danych i funkcje umieścić w module UART.

12.1 Struktura danych

Do wysyłania łańcucha znakowego za pomocą UART-a służyć będzie **bufor nadajnika**:

```
enum eTransmiterStatus {FREE, BUSY};
typedef struct TransmiterBuffer{
  char cData[TRANSMITER_SIZE];
  enum    eTransmiterStatus eStatus;
  unsigned char fLastCharacter;
  unsigned char cCharCtr;
};
struct TransmiterBuffer sTransmiterBuffer;
```

cData służy do przechowywania łańcucha znakowego, który ma być wysłany.
eStatus przechowuje status bufora. Pole może przyjmować stany BUSY albo FREE: o FREE - z bufora pobrano wszystkie znaki (patrz opis funkcji Transmiter_GetCharacterFromBuffer), o BUSY - z bufora nie pobrano jeszcze wszystkich znaków.
fLastCharacter to flaga (0,1), która służy do zapamiętania faktu, że z cData pobrano ostatni znak, tzn. NULL (sens tego pola będzie widoczny podczas implementacji funkcji *GetCharacterFromTxBuffer*)**.**
ucCharCtr służy do iteracji po łańcuchu znakowym.

12.2 Funkcje pomocnicze

Zadaniem funkcji **char Transmiter_GetCharacterFromBuffer()** jest pobieranie znaków z bufor nadajnika Funkcja powinna zwracać po kolei znaki z *CData*.
Po osiągnięciu znaku NULL funkcja powinna zwrócić znak terminatora czyli \r. (zamiast NULL).
W następnym wywołaniu funkcja powinna zwrócić NULL oraz odpowiednio zmienić status bufora.
Do iteracji po tablicy należy wykorzystać zmienną *ucCharCtr.*
Do wysłania po terminatorze znaku NULL należy użyć zmiennej (flagi) *fLastCharacter.*

Przykładowy test:
Jeżeli w buforze nadajnika znajduje się string „Ala" to następująca sekwencji wywołań:

```
cTxBuffer_GetCharacter (void);
cTxBuffer_GetCharacter (void);
cTxBuffer_GetCharacter (void);
cTxBuffer_GetCharacter (void);
cTxBuffer_GetCharacter (void);
```

zwróci po kolei: 'A', 'l', 'a', TERMINATOR, NULL.

Funkcja **void Transmiter_SendString(unsigned char cString[])** służy do inicjalizacji wysyłania stringa podanego w argumencie. Funkcja powinna:

- kopiować *cString* do pola *cData* bufora nadajnika*,*
- inicjalizować wysłanie pierwszego znaku przez wpisanie go do *U0THR* (Transmitter Holding Register układu UART0),
- ustawiać odpowiednio wartości status i licznik bufor nadajnika.

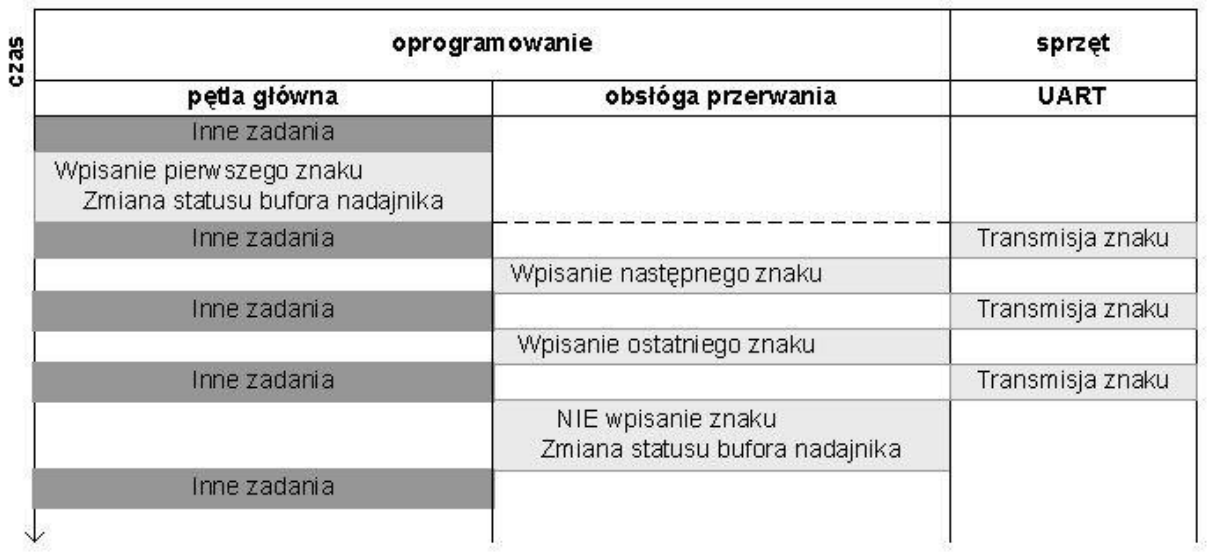
Funkcja **enum eTransmiterStatus Transmiter_GetStatus(void)** odczytuje status bufora.

12.3 Schemat transmisji

Wysyłanie łańcucha znaków poprzez UART za pomocą przerwań powinno odbywać się w sposób następujący.

1. W pierwszej kolejności należy zainicjalizować wysyłanie znaku poprzez wpisanie do rejestru *U0THR* UARTA-a pierwszego znaku z bufora. (Transmiter_SendString)
2. Następnie UART zajmuje się wysyłaniem znaku.
   UWAGA: Dla prędkości transmisji 9600 b/s trwa to ok. 1 ms. Dla porównania, wykonanie jednej instrukcji procesora ARM zajmuje mniej niż 1 µs.
3. Zakończenie wysyłania znaku (nie mylić z wpisaniem do rejestru – patrz punkt 2) powoduje wywołanie procedury obsługi przerwania.
4. W procedurze obsługi przerwania należy zainicjalizować wysyłanie następnego znaku z łańcucha (Transmiter_GetCharacterFromBuffer), w następnym przerwaniu następnego itd.
5. Zakończenie wysyłania znaków polega na nie inicjalizowaniu wysyłania następnego znaku podczas obsługi przerwania wygenerowanego przez UART, po zakończeniu wysyłania poprzedniego znaku.

Schemat transmisji pokazano poniżej:



12.4 Ćwiczenia

Ćwiczenie 1

Napisać program który będzie w sposób ciągły wysyłał następujące po sobie kody ASCII, czyli liczby od 0 do 255.

W pierwszej kolejności należy zmodyfikować inicjalizację UART-a, tj.: ustawić odpowiedni pin do pracy jako wyjście nadajnika oraz odblokować przerwania nadajnika (odpowiednia maska jest już zdefiniowana).

Następnie wstawić w odpowiednie miejsce w przerwaniu UARTA wysyłanie kolejnego znaku do rejestru *U0THR*. Inicjalizacja transmisji czyli wpisanie pierwszej wartości do rejestru *U0THR* powinno mieć miejsce w funkcji main.

W tym programie nie używamy jeszcze bufora nadajnika.

Test przeprowadzić na symulatorze.

Ćwiczenie 2

Napisać program, który wyśle raz łańcuch znakowy „test123”.

Użyć funkcji bufora nadajnika.

Test na symulatorze.

Ćwiczenie 3

Napisać program, który będzie wysyłał ciągle łańcuch znakowy „test123”.

W tym celu należy w głównej pętli cyklicznie sprawdzać status bufora nadajnika. Jeżeli jest FREE należ wysłać następny string.

Test na symulatorze i zestawie uruchomieniowym.

Ćwiczenie 4

Napisać program, który będzie wysyłał jeden po drugim łańcuchy „licznik 0x0000”, „licznik 0x0001”, „licznik 0x0002” itd. Do tworzenia łańcuchów do wysłania należy wykorzystać napisane wcześniej funkcje.

Test na symulatorze i zestawie uruchomieniowym.

Ćwiczenie 5

Napisać program pełniący funkcję zegara z minutami i sekundami.

Struktura danych zegara powinna wyglądać następująco:

struct Watch { unsignech char ucMinutes; ucSecconds; unsigned char fSeccondsValueChanged, fMinutesValueChanged }

Zegar powinien być aktualizowany co sekundę w przerwaniu Timera0.

W tym celu należy stworzyć funkcje *WatchUpdate()* i podpiąć ją do przerwań Timera0.

Zmianie sekund powinno towarzyszyć ustawienie flagi fSeccondsValueChanged, a zmianie minut flagi fMinutesValueChanged.

W pętli głównej należy sprawdzać status nadajnika, jeżeli jest wolny należy sprawdzić czy są do wysłania nowe sekundy albo minuty.

Każda zmiana sekund powinna spowodować wysłanie łańcucha znakowego „sec 0xSSSS", a każda zmiana minut „min 0xMMMM", gdzie SSSS i MMMM oznaczają odpowiednio ilość sekund i minut.

UWAGA:

Program nie powinien gubić minut ani sekund w momencie równoczesnej zmiany minut i sekund, np. sec 59->0, min 2->3, kolejność wysłania minut i sekund nie ma znaczenia.

Ćwiczenie 6

Do poprzedniego programu dodać funkcjonalność kalkulatora.

Po odebraniu komendy „calc" z argumentem liczbowym program powinien odesłać „calc 0xXXXX", gdzie 0xXXXX jest argumentem liczbowym podanym przy wywołaniu komendy pomnożonym przez 2 (czyli kalkulator, który mnoży liczby przez 2).

Program należy napisać w taki sposób żeby kalkulator i zegarek nie kolidowały ze sobą. Przykładowo, nie może zaistnieć sytuacja, gdy uC odsyła „calc" i jednocześnie w tym samym czasie zaczyna wysyłać sekundy, w efekcie czego wysłany string będzie wyglądać następująco: „calsek 0x0034".