Kurs programowania mikrokontrolerów PIC (3)

Konfigurowanie portów I/O i timerów. Odmierzanie opóźnień

W tym odcinku kursu podajemy, w jaki sposób wykonać konfigurowanie projektu w środowisku MPLAB oraz pokażemy przykłady prostych programów, które pozwolą zacząć pracę z mikrokontrolerami PIC.

Dodatkowe materiały na CD/FTP:
ftp://ep.com.pl, user: 12147, pass: 2e7u6a2a
- pierwsza i druga część kursu

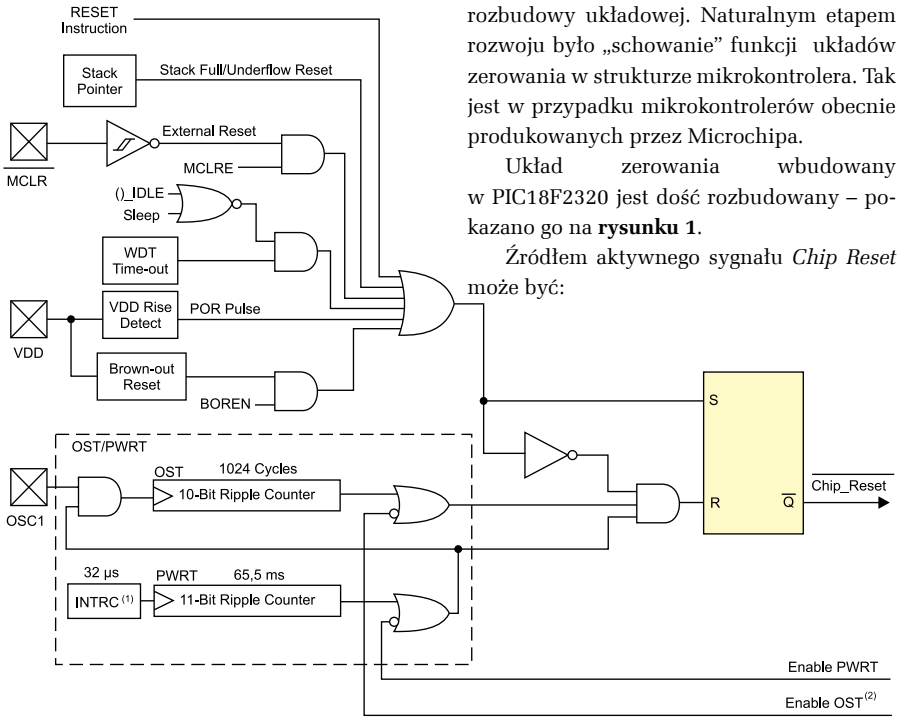
Zanim przejdziemy do programu zatrzymamy się przez chwilę przy zerowaniu mikrokontrolera. Prawidłowo wykonanie zerowanie jest jednym z warunków prawidłowej pracy mikrokontrolera i nie wolno go bagatelizować.

Zerowanie mikrokontrolera

W starszych mikrokontrolerach zerowanie było wykonywane przez wymuszenie stanu aktywnego (wysokiego lub niskiego) na wejściu oznaczanym jako *Reset*. Po włączeniu zasilania niezbędny impuls był generowany przez prosty układ RC, a zerowanie na żądanie realizowano przez zwarcie styku podłączonego do wejścia zerowania. Szybko się okazało, że te proste środki to za mało w bardziej wymagających aplikacjach i pojawiły się specjalizowane układy scalone potrafiące generować impuls zerowania po włączeniu zasilania, nieoczekiwanym obniżeniu napięcia zasilania oraz z funkcją watchdog'a. Chociaż spełniały swoją funkcję bardzo dobrze, to ich stosowanie wiązało się z dodatkowymi kosztami i koniecznością rozbudowy układowej. Naturalnym etapem rozwoju było „schowanie" funkcji układów zerowania w strukturze mikrokontrolera. Tak jest w przypadku mikrokontrolerów obecnie produkowanych przez Microchipa.

Układ zerowania wbudowany w PIC18F2320 jest dość rozbudowany – pokazano go na **rysunku 1**.

Źródłem aktywnego sygnału *Chip Reset* może być:



Rysunek 1. Układ zerowania mikrokontrolera PIC18F2320

- wykrycie włączenia napięcia zasilającego (Power On Reset, POR).
- obniżenie się napięcia zasilającego (Brown Out Reset, BOR).
- wykrycie stanu niskiego na wejściu zerowania MCLR,
- przepełnienie się licznika watchdoga WDT,
- przepełnienie się stosu,
- wykonanie instrukcji *RESET*.

Układ zerowania jest konfigurowany w rejestrach konfiguracyjnych i można jego działanie dostosować do wymogów aplikacji.

Jednym z trudniejszych momentów pracy mikrokontrolera jest włączanie zasilania. Microchip dopuszcza tradycyjne generowanie impulsu zerującego po włączeniu zasilania na wyprowadzeniu MCLR przez układ RC (aktywny stan niski). To proste i skuteczne rozwiązanie nie może być stosowane, kiedy chcemy programować pamięć w układzie (ICSP). Programator podaje na wejście MCLR napięcie +12...13 V programujące pamięć Flash. Układ RC powoduje, że impuls przełączający jest zniekształcany i programowanie nie jest możliwe, ale za to możliwe jest uszkodzenie programatora. W takim przypadku trzeba połączyć wejście MCLR do plusa zasilania przez rezystor, aby można było wykorzystywać wewnętrzne mechanizmy zerowania i programować układ w systemie.

Zerowania POR jest wspomagane przez 2 liczniki *Power Up Timer* (PWRT) i *Oscillator Start Up Timer* (OST). PWRT jest licznikiem 11-bitowym, zliczającym impulsy o częstotliwości ok. 32 kHz z wewnętrznego generatora RC. Licznik zaczyna odliczanie po włączeniu zasilania, a jego przepełnienie następuje po około 65 ms. Ten czas jest wystarczający dla wygenerowania prawidłowego impulsu

Listing 1. Definicje atrybutów makra __CONFIG
```
// Configuration register 1
 #define IESOEN 0xFFFF  // Internal/External switch over mode enabled
 #define IESODIS 0x7FFF // Internal/External switch over mode disabled
 #define FCMEN 0xFFFF   // Fail-safe clock monitor enabled
 #define FCMDIS 0xBFFF  // Fail-safe clock monitor disabled
 #define EXTCLKO 0xFFFF // External RC, RA6=CLKOUT
 #define RCCLKO 0xF9FF  // Internal RC, RA6=CLKOUT
 #define RCIO 0xF8FF    // Internal RC, RA6=IO
 #define EXTIO 0xF7FF   // External RC, RA6=IO
 #define HSPLL 0xF6FF   // HS with PLL enabled
 #define ECIO 0xF5FF    // EC, RA6=IO
 #define ECCLKO 0xF4FF  // EC, RA6=CLKOUT
 #define HS 0xF2FF      // HS osc
 #define XT 0xF1FF      // XT osc
 #define LP 0xF0FF      // LP osc
```

Fotografia 2. Umieszczenie mikrokontrolera PIC18F2320 w podstawce

zerowania przy minimalnej prędkości narastania napięcia zasilającego określonego w danych technicznych, a w praktyce dla zdecydowanej większości stabilizatorów liniowych i impulsowych. Licznik można włączyć lub wyłączyć bitem PWRTEN w rejestrze konfiguracyjnym. Ponieważ w naszym module ewaluacyjnym nie ma układu RC dołączonego do MCLR, to licznik będzie zawsze włączony przez wyzerowanie bitu PWRTEN.

Działanie licznika OST jest uwarunkowane rodzajem wybranego oscylatora. Po wybraniu oscylatora kwarcowego jest automatycznie włączany, a dla oscylatorów RC jest wyłączany. OST odlicza 1024 okresy drgań oscylatora kwarcowego o amplitudzie zdolnej wysterować wejście cyfrowe.

Zerowanie POR jest najważniejszym elementem zerowania interesującym nas w tej chwili. Nie będziemy wykorzystywać licznika watchdoga WDT i zerowania po obniżeniu napięcia zasilania. Warto jednak zapoznać się z tymi elementami dokładniej, bo są bardzo użyteczne na przykład w aplikacjach pracujących w trudnych warunkach przemysłowych. W rejestrze konfiguracyjnym CONFIG2L wyzerowane są bity ! PWRTEN (odblokowanie PWRT) i BOR (BOR wyłączony). Musimy tez wyłączyć WDT przez wyzerowanie bitu WDTEN w rejestrze CONFIG2H.

Wybór oscylatora i konfigurowanie mikrokontrolera

Płytka ewaluacyjna ma możliwość dołączenia rezonatora kwarcowego i taktowania mikrokontrolera za pomocą wbudowanego generatora kwarcowego. Ma to swoje dobre strony, głównie stabilny przebieg taktujący z częstotliwością niewiele zależną od temperatury. Wadą jest konieczność zmiany rezonatora kiedy zaistnieje potrzeba zmiany częstotliwości. W naszych przykładach wykorzystamy wewnętrzny oscylator RC. Taki wybór jest podyktowany głównie możliwością elastycznego przeprogramowywania częstotliwości taktującej przez zapisywanie rejestru OSCCON omawianego w pierwszej części kursu. W pierwszym rejestrze konfiguracyjnym wybieramy oscylator RC z RA6 do wykorzystania jako linię portu I/O. Ponieważ został wybrany oscylator RC, to IESO i FSCM będą wyzerowane.

Makro definiujące zawartość pierwszego rejestru konfiguracji będzie wyglądało tak: *__CONFIG(1, IESODIS&FCMDIS&RCIO)*.

Atrybuty makra są zdefiniowane w pliku nagłówkowym pic18F2320.h – fragment definicji dla rejestru 1 (CONFIG1H) pokazano na **listingu 1**.

Rejestr drugi (CONFIG2Li CONFIG2H) konfiguruje działanie licznika PWRT, zerowania BOR i watchdoga:

__CONFIG(2,BORDIS&PWRTEN&WDTDIS)

Pozostałe rejestry możemy pozostawić dla potrzeb kursu w stanie domyślnym (same jedynki po kasowaniu układu).

Program sterujący portami

Jak już wiemy do potrzeb testu został wybrany mikrokontroler PIC18F2320 w obudowie DIP28 do montażu przewlekanego. W płytce ewaluacyjnej należy go umieścić w podstawce J3 jak pokazano na **fotografii 1**.

Zakładam, że mamy utworzony nowy projekt i w jego katalogu plik tekstowy *main.c*. Pierwszą rzeczą, którą zrobimy będzie dołączenie pliku nagłówkowego z definicjami rejestrów właściwą dla wybranego mikrokontrolera PIC18F2320. W kompilatorze HiTech dołączanie plików nagłówkowych jest stosunkowo proste. Niezależnie od wybranego układu dołącza się plik o nazwie *htc.h*. Kompilator na jego podstawie i informacji o wybranym mikrokontrolerze sam dołącza niezbędny plik nagłówkowy – w naszym przypadku *pic18f2320.h*. Jeżeli wybierzemy inny typ mikrokontrolera, to wystarczy poprawić dane projektu i nic nie trzeba zmieniać w plikach źródłowych. Po dołączeniu plików nagłówkowych definiujemy makra konfigurujące, aby nie trzeba było za każdym razem ustawiać bitów konfiguracji w okienku *Configuraton bits*.

W tym momencie mamy wszystko co potrzebne do napisania własnego, krótkiego programu i co ważniejsze, jest duże prawdopodobieństwo, że ten program zadziała.

Każdy program napisany w języku C wymaga zdefiniowana funkcji *main*. Od niej zaczyna się wykonywanie instrukcji po zerowaniu mikrokontrolera. W najprostszym, wykorzystywanym tutaj przypadku, *main* nie może niczego zwracać i nie przyjmuje żadnych argumentów. Musi też kończyć się pętlą niekończoną, której program nie może opuścić, ponieważ nie ma systemu operacyjnego, który przejąłby kontrolę nad CPU. Na początku funkcji umieszczony jest wpis do rejestru OSCCON ustalający taktowanie z częstotliwością 4 MHz za pomocą wbudowanego generatora RC.

Początkowy „szkielet" programu pokazano na **listingu 2**. Program nie będzie wypisywał nieśmiertelnego „Hello Word", ale zaświeci jedną z diod LED umieszczoną w module. Żeby było to możliwe, trzeba się najpierw zapoznać z budową portów I/O mikrokontrolerów z rodziny PIC18F.

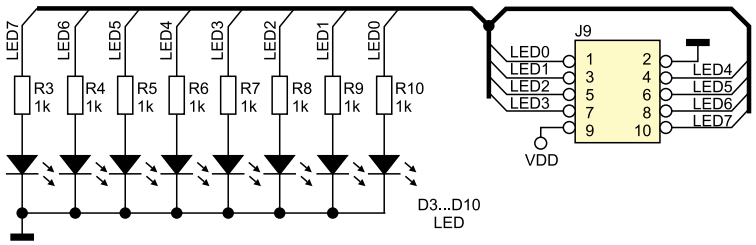
Zależnie od typu mikrokontrolera (w praktyce, od zastosowanej obudowy) dostępnych jest do pięciu 8-bitowych portów I/O. W układ PIC18F2320 wbudowano 3 porty: PORTA, PORTB i PORTC. Trzeba pamiętać, że prawie wszystkie linie portów są współdzielone z wyprowadzeniami modułów peryferyjnych. Alternatywne funkcje linii portów są uaktywniane po włączeniu modułu peryferyjnego, poza jednym wyjątkiem: przetwornika analogowo-cyfrowego. Z trochę niezrozumiałych dla mnie powodów część linii portów PORTA i PORTB po zerowaniu domyślnie jest wejściami przetwornika. Aby były portami cyfrowymi, trzeba je przeprogramować za pomocą rejestru ADCON1. Często zapominają o tym początkujący programiści i nawet proste programy z użyciem portów nie chcą działać prawidłowo.

Żeby wszystkie porty układu były portami cyfrowymi trzeba do ADCON1 wpisać 0x0F. Dokładniej temu rejestrowi przyjrzymy się przy okazji omawiania przetwornika A/C.

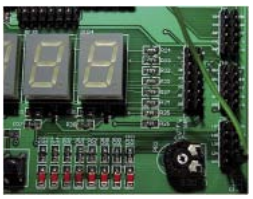
Każdy z portów ma trochę inną budowę wewnętrzną, ale nie będziemy jej dokładnie omawiać. Dla nas ważne jest, że wszystkie porty są dwukierunkowe. Kierunek przepływu informacji jest programowany rejestrem TRISx. Dla portu PORTA będzie to TRSA, portu PORTB, TRISB itd.

Wyzerowanie bitu TRIS ustalenie funkcji linii jako wyjścia, natomiast jego ustawienie

Listing 2. Szkielet programu
```
#include <htc.h>
__CONFIG(1, IESODIS&FCMDIS&RCIO);
__CONFIG(2,BORDIS&PWRTEN&WDTDIS);

void main (void){
  OSCCON=0x63; //częstotliwość 4MHz
  while(1);
}
```



Rysunek 3. Schemat podłączenia diod LED



Fotografia 4. Umiejscowienie złącza J9 i diod na płytce

Fotografia 5. Połączenie linii RA0 złącza J13

jako wejścia. Na przykład wyzerowanie bitu TRISA.0 załączenie funkcji linii PORTA.0 jako wyjściowej. Usatwienie TRISA.1 powoduje, że PORTA.1 staje się linią wejściową. W ten sposób można dowolnie ustawić kierunek każdej z linii wszystkich portów.

Oprócz rejestru TRISx z portami są związane jeszcze 2 rejestry PORTx i LATx. Rejestr PORTx jest przeznaczony do odczytywania stanów linii zaprogramowanych jako wejścia (odczytywanie PORT) i zamiany stanów linii wyjściowych (zapisywanie PORT). W starszej rodzinie PIC16F do operacji czytania wejścia i wysyłania stanu na wyjście stosuje się tylko rejestr PORT. Sterowanie stanami wyjść przy wykorzystaniu tylko tego rejestru powoduje problemy przy wykonywaniu instrukcji *czytaj-modyfikuj-zapisz* i zmianie kierunku linii z wejściowej na wyjściową. Żeby ten problem wyeliminować w PIC18F wprowadzono dodatkowy rejestr LATx. Aby zmienić stan linii wyjściowych można zapisywać rejestr PORT lub rejestr LAT. Jednak bezpieczniej jest się posługiwać rejestrem LAT. Odczytanie stanów linii wejściowych będzie polegało na odczytaniu rejestru PORT.

Na początek zaświecimy jedną diodę LED sterowaną z linii RA0. Schemat sterowania diod LED został pokazano na **rysunku 3**, natomiast umiejscowienie złącza J9 na płytce pokazano na **fotografii 4**. Ponieważ diody są połączone anodami przez rezystory ograniczające prąd do złącza sterującego J9, to żeby je zapalić trzeba podać na piny złącza napięcie dodatnie (stan wysoki). Do połączeń pomiędzy pinami zastosujemy gotowe przewody ze specjalnymi nasadkami pasującymi do szpilek goldpinów.

Aby dioda świeciła się, trzeba na linii portu PORTA0 wystawić stan wysoki i połączyć pin LD0 złącza J9 z pinem RA0 złącza J13 (fotografia 4 i **fotografia 5**). Program sterujący diodą LED zamieszczono na **listingu 3**.

Jak się okazuje, żeby zaświecić jedną diodę LED trzeba przyswoić dość sporo wiadomości wstępnych: wybrać oscylator i częstotliwość taktowania (rejestr OSCCON), ustawić bity konfiguracyjne(makro __CONFIG), dołączyć plik nagłówkowy z definicjami rejestrów (*htc.h*), wyłączyć wejścia przetwornika A/C i ustawić porty jako cyfrowe (rejestr ADCON1), zaprogramować rejestr TRISA, tak aby RA0 była wyjściowa i wysłać na RA0 stan wysoki. To proste ćwiczenie jest doskonałym treningiem przed bardziej skomplikowanymi zadaniami. Bardziej dociekliwy Czytelnik może w tym momencie dokładniej przestudiować strukturę rejestrów konfiguracyjnych i poznać dokładniej wewnętrzną budowę portów.

Odliczanie opóźnień – miganie diody LED

Jeżeli ktoś uważa, ze świecąca dioda LED wygląda mało atrakcyjnie, to można pokaz ożywić zmuszając ja do migania. Gdybyśmy wysyłali stan wysoki (zapalanie diody) i stan niski (gaszenie diody) zaraz po sobie, to będzie to następowało tak szybko po sobie, ze będzie postrzegane jako ciągłe świecenie. Żeby zobaczyć miganie musimy pomiędzy zmianami odczekać jakiś interwał czasu.

Odliczanie opóźnień polega na zajęciu mikrokontrolera odliczaniem czasu przez żądany czas. Żeby tematu nie sprowadzać tylko do migania diody spróbujemy spojrzeć na to zadanie szerzej.

Po pierwsze określmy jakie czasy nas interesują. Paradoksalnie im krótszy czas opóźnienia, tym większe problemy możemy napotkać. Mikrokontroler taktowany z częstotliwością Fosc najczęściej wykonuje jeden rozkaz w czasie jednego cyklu rozkazowego równego 4 okresom zegara taktującego. W rodzinie PIC18F są tez rozkazy wymagające dwu cykli rozkazowych. Dla częstotliwości taktowania wynoszącej 4 MHz czas wykonania rozkazu w jednym cyklu maszynowym wyliczamy następująco:

- częstotliwość cykli rozkazowych 4 MHz/4=1 MHz – bo 4 okresy zegara na 1 cykl rozkazowy,
- czas trwania jednego cyklu 1/1000000= 1 μs.

Tak taktowany mikrokontroler może odliczyć minimalne opóźnienie równe 1 μs i wielokrotność tej wartości. Zmniejszenie taktowania do 2 MHz wydłuża ten czas do 2 μs, a taktowanie 8 MHz skraca go do 500 ns. Najczęściej takie małe opóźnienia odlicza się wstawiając w program instrukcję NOP. Jeżeli chcemy odliczyć 10 μs, to trzeba wstawić 10 instrukcji NOP. Sprawa się komplikuje kiedy chcemy wstawić opóźnienie na przykład 200 μs. Nie możemy zrobić tego w pętli na przykład takiej:

```
for(i=0;i<200;i++)
asm(„nop”);
```

bo do odliczania czasu dodadzą się instrukcje dekrementacji zmiennej i sprawdzania warunku czy i jest równa 0. W rezultacie zostanie odliczonych nie 200 μs, a dużo więcej. Czy da się policzyć ile? Da się, ale trzeba zobaczyć jak kompilator zamieni to na instrukcje asemblera i policzyć cykle rozkazowe. Zadanie niezbyt wdzięczne, ale wykonalne.

Dużo większej pracy wymagałoby napisanie procedury, która miałaby w ten sposób odliczać ściśle określone opóźnienia wyrażane w μs lub w ms. Oczywiście jest to wykonalne. Na przykład kompilator C firmy CCS ma wbudowaną bibliotekę programowego odliczania opóźnień przez zliczanie cykli rozkazowych. W kompilatorze Hi-Tech takiej biblioteki nie ma i trzeba sobie radzić inaczej. W sytuacjach, kiedy potrzebne jest dokładne odliczanie opóźnień, naszym wielkim sprzymierzeńcem będą wbudowane moduły liczników, ale o tym powiemy później.

Wróćmy do ulepszania naszego programu. Do migania dioda nie będą potrzebne opóźnienia rzędu mikrosekund, czy pojedynczych milisekund, ale raczej setek milisekund. Możemy sobie pozwolić na odliczanie „jakiegoś” opóźnienia tak by dioda migała tak jak się nam podoba. W takiej sytuacji nie ma znaczenia jaki jest czas odliczania w sekundach, czy milisekundach, bo dobierzemy go sobie „na oko”. Do tego celu zatrudnimy mikrokontroler, żeby sobie pracował w pętli inkrementując 16 bitową zmienną. Do tego celu napiszemy prostą funkcję i nazwiemy ją na przykład delay (**listing 2**).

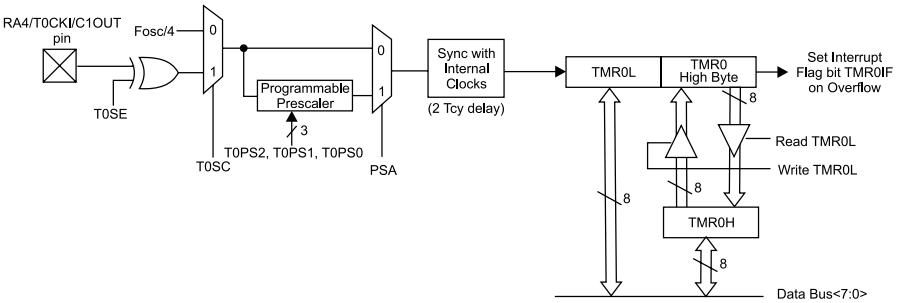
Listing 3. Zapalenie jednej diody LED

```
#include <htc.h>
__CONFIG(1, IESODIS&FCMDIS&RCIO);
__CONFIG(2,BORDIS&PWRTEN&WDTDIS);

void main (void){
  OSCCON=0x63; //czestotliwość 4MHz
  ADCON1=0x0f; //wyłączenie wejść przetwornika ADC
  TRISA=0xFE;  //RA0 wyjściowa, reszta wejścia
  LATA=0x01;   //RA=1
  while(1);
}
```

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0

```
Bit 7: TMR0ON
1-odblokowanie licznika
0-zablokowanie licznika
Bit 6 :T08BIT
1-licznik 8 bitowy
0-licznik 16bitowy
Bit 5:T0CS
1-zliczane są impulsy z wejścia T0CKI
0-zliczane są impulsy o częstotliwości
Fosc/4 (cykle rozkazowe)
Bit 4:T0SE
1 – inkrementacja licznika przy
opadającym zboczu T0CKI
0 – inkrementacja licznika przy
narastającym zboczu T0CKI

Bit 3: PSA
1-preskaler nie jest przypisany
0-preskaler  jest przypisany do Timer0
Bity 2:0 T0PS2:T0PS0 podział
częstotliwości przez preskaler
111  - 1:256
110  - 1:128
101  - 1:64
100   -1:32
011  -1:16
010  -1:8
001  -1:4
000  -1:2
```

Rysunek 6. Rejestr T0CON



Rysunek 7. Licznik Timer0 w trybie 16-bitowym

Zmieniając wartość początkową nadawaną zmiennej i możemy w szerokich granicach zmieniać czas jaki musi przeznaczyć mikrokontroler na wykonanie funkcji *delay*. Na **listingu 3** przedstawiono fragment programu sterującego miganiem diody LED.

Ten program może posłużyć do ciekawego eksperymentu polegającego na przeprogramowaniu dzielnika ustalającego częstotliwość taktowania mikrokontrolera w rejestrze OSCCON i obserwacji jak to wpływa na częstotliwość migania diody LED.

Co zrobić, kiedy będziemy potrzebowali opóźnień o ściśle określonych czasach na przykład raz 230 ms, a zaraz 1,23 s? Pomysł żmudnego liczenia cykli rozkazowych nie jest najlepszy. Wykorzystamy do tego celu wbudowany w mikrokontroler sprzętowy licznik i mechanizm przerwań. Pomimo, że sam program odliczający w ten sposób opóźnienia będzie zadziwiająco prosty, to do jego napisania musimy poznać budowę licznika Timer0 i powiedzieć co nieco o mechanizmach przerwania. Jednak trud ten nam się opłaci, bo większość problemów z odliczaniem czasu rozwiążemy raz na zawsze.

Idea odliczania czasu będzie się opierać na zbudowaniu wewnętrznego zegara „tykającego" co 1 ms. To „tykanie" będzie się odbywało prawie niezauważalnie dla wykonywanego programu, głównego dlatego, że wykorzystamy mechanizm przerwań.

Jak już wspomniałem, do odliczania interwału 1 ms będzie zastosowany licznik Timer0 skonfigurowany przez rejestr TCON (**rysunek 6**).

Licznik może być skonfigurowany jako 8 bitowy (tryb zgodności z rodzina PIC16F), lub jako 16 bitowy. Do naszych celów skonfigurujemy go tak by w trybie 16-bitowym (T8BIT=0) zliczał cykle rozkazowe o częstotliwości Fosc/4 (T0CS=0). Preskaler nie będzie nam potrzebny i dlatego nie przypiszemy go do zliczania cykli (PSA=0). Na rysunku 7 jest pokazana konfiguracja licznika w trybie 16 bitowym.

Licznik zlicza w przód i po przepełnieniu z wartości 0xFFFF do 0x0000 jest ustawiany wskaźnik przerwania TMR0IF. Ustawienie TMR0IF może spowodować zgłoszenie przerwania, jeżeli na to pozwolimy.

Aby odliczać opóźnienie z rozdzielczością 1 ms potrzebne będzie zgłaszanie przerwań co 1 ms. Licznik TMR0 ma zliczać impulsy zegarowe o częstotliwości 4 MHz/4=1 MHz. Jeżeli podzielimy częstotliwość przez 1000, to na wejściu licznika otrzymamy 1 MHz/1000=1 kHz. Licznik będzie się przepełniał z częstotliwością 1 kHz, czyli co 1 ms. Trzeba pamiętać, że 16-bitowy licznik liczy w górę i aby zliczył 1000 impulsów trzeba do niego wpisać jako wartość początkową 65536–1000=64536 (szesnastkowo 0xFC18).

W 8-bitowym mikrokontrolerze 16-bitowy licznik musi być zapisywany w 8-bitowych „porcjach". Dlatego przez system licznik jest widziany jako dwa 8-bitowe rejestry TMR0L i TMR0H. Uważny Czytelnik zauważy, że może to być przyczyną potencjalnych problemów. Jeżeli na przykład będziemy odczytywać pracujący licznik w dwóch krokach, to możemy otrzymać zafałszowaną wartość. Pokażę to na przykładzie. Odczytujemy wartość licznika 0x23FF. Po odczytaniu młodszej części mamy 0xFF, ale licznik w trakcie odczytywania liczy dalej i zmienia wartość na 0x2400. Odczytujemy teraz starszą część i mamy 0x24, czyli odczytaliśmy 0x24FF zamiast 0x23FF. Żeby temu zapobiec, w momencie odczytywania młodszego bajtu TMR0L sprzętowo jest zapisywany TMR0H wartością rejestru licznika TMR0. TMR0 liczy dalej, ale w TMR0H mamy wartość odczytana równo w momencie odczytania TMR0L. Podobnie jest z zapisywaniem: zapisanie TMR0L powoduje automatyczne uaktualnienie TMR0 wartością z TMR0H. Jest to dla nas ważna wskazówka: najpierw trzeba zapisać TMR0H, a dopiero potem TMR0L.

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF

```
bit 7:   GIE/GIEH
  kiedy bit IPEN =0
   1 odblokowanie wszystkich
niezamaskowanych przerwań
   0 zablokowanie wszystkich przerwań
  Kiedy bit IPEN=1
  1 odblokowanie wszystkich przerwań
wysokiego priorytetu
  0 zablokowanie wszystkich przerwań
wysokiego priorytetu
bit 6:   PEIE /GIEL
kiedy bit IPEN =0
   1 odblokowanie wszystkich
niezamaskowanych przerwań od peryferii
   0 zablokowanie wszystkich przerwań
od peryferii
  Kiedy bit IPEN=1
  1 odblokowanie wszystkich przerwań
niskiego priorytetu
  0 zablokowanie wszystkich przerwań
niskiego priorytetu

bit 5:   TMR0IE
1 odblokowane przerwanie generowane
w momencie przepełnienia licznika/
timera TIMER0
  0  zablokowane przerwanie generowane
w momencie przepełnienia licznika/
timera TIMER0
bit 4:   INT0E
   1 odblokowane przerwanie zewnętrzne
z linii INT
   0 zablokowane przerwanie zewnętrzne
z linii INT
bit 3:   RBIE
lodblokowane przerwanie zgłaszane przy
zmianach sygnału na liniach RB4...RB7
portu PORTB
0 zablokowane przerwanie zgłaszane
przy zmianach sygnału na liniach
RB4...RB7 portu PORTB
bit 2:   TMR0IF  znacznik zgłoszenia
przerwania
 1 wystąpiło przepełnienie licznika
TIMER0, czyli zmiana stanu licznika
z FFFFh na 0000h
  (musi być zerowany programowo)
  0 licznik TIMER0 nie przepełnił się
bit 1:   INT0IF  znacznik zgłoszenia
przerwania
  1 wystąpiło przerwanie zewnętrzne
INT, czyli zmiana stanu na wejściu INT
(RB0)
  (znacznik musi być zerowany
programowo)
  0 przerwanie INT nie wystąpiło
bit 0:   RBIF znacznik zgłoszenia
przerwania
  1 nastąpiła zmiana stanu na liniach
RB7...RB4 portu PORTB
  (znacznik musi być zerowany
programowo)
  0 na liniach RB7...RB4 portu PORTB
nie wystąpiła zmiana stanu
```

Rysunek 8. Rejestr INTCON

Wiemy już jak zapisać licznik, by odliczył 1000 impulsów o częstotliwości Fclk/4 i się przepełnił ustawiając znacznik TMROIF. W tym momencie mamy prosty sposób odliczania impulsów o czasie trwania równym 1/1000 000=1 μs. Ładujemy licznik potrzebną wartością pamiętając o zasadzie zapisywania rejestrów TMR0L i TMR0H oraz o tym, że licznik liczy w górę i testujemy ustawienia wskaźnika TMR0IF. Po jego ustawieniu wiemy, że licznik odliczył zadaną liczbę μs. Trzeba jeszcze pamiętać, że flaga TMR0IF musi być wyzerowana programowo przed załadowaniem licznika.

Wróćmy jednak do dalszego konstruowania mechanizmu „tykania" co 1 ms. Do tego celu potrzebne nam będzie zgłaszanie przerwania przy każdym przepełnieniu się licznika Timer0. Żeby przerwanie mogło być zgłaszane po ustawieniu wskaźnika TMR0IF, trzeba je jawnie odblokować, bo domyślnie po włączeniu zasilania wszystkie przerwania są zablokowane. Konfigurowanie układu przerwań od licznika Timer0 polega na zapisywaniu rejestru INTCON (**rysunek 8**).

Układ przerwań w mikrokontrolerach PIC18F może być zaprogramowany z możliwością programowania dwu (niski i wysoki) priorytetów, lub bez priorytetów. Decyduje o tym stan bitu IPEN z rejestru RCON. Domyślnie po włączeniu zasilania IPEN=0, co oznacza, że układy priorytetów przerwań są wyłączone. Dla naszych potrzeb, gdzie przerwanie będzie zgłaszane z jednego źródła zostawimy to domyślne ustawnie.

W rejestrze INTCON bit 7 (GIE) odblokowuje cały system przerwań, a bit 6 (PEIE) odblokowuje zgłaszanie przerwań od układów peryferyjnych.

Żeby przepełnienie licznika Timer0 mogło zgłosić przerwanie trzeba: ustawić bit TMR0IE zezwolenia na przerwanie od ustawienia flagi TMR0IF , ustawić bit PEIE i ustawić bit GIE.

Tak zaprogramowany system przerwań po ustawieniu TMR0IE zgłosi przerwanie i od wektora przerwania 0x00008 zacznie się wykonywać procedura obsługi przerwania.

W każdym kompilatorze dla mikrokontrolerów jest przygotowana specjalna składnia dla funkcji przerwania, tak by kompilator mógł ją zidentyfikować i umieścić jej wykonywanie od wektora przerwania. Programista piszący procedurę obsługi w asemblerze musi pamiętać o tym, żeby na początku zapisać wartość wszystkich rejestrów używanych (niszczonych) przez procedurę obsługi i na jej końcu te wartości odtworzyć. Pisząc w C jesteśmy w tej komfortowej sytuacji, że o to wszystko troszczy się kompilator. W kompilatorze Hi-Tech procedura obsługi przerwania musi mieć kwalifikator interrupt: void interrupt nazwa_procerdury(void)

i nie może mieć żadnych argumentów ani niczego zwracać. Dla konfiguracji z odblokowanymi priorytetami dla niskiego priorytetu kwalifikator jest uzupełniany o wyrażenie low_priority. Jest jeszcze jedna ważna rzecz, o której już wspomniałem. Ustawiany wskaźnik przerwania TMR0IF musi być wyzerowany programowo. Jeżeli tego nie zrobimy, to po zakończeniu obsługi zostanie natychmiast zgłoszone nowe przerwanie i mikrokontroler nie będzie nic innego robił tylko wykonywał obsługę przerwania. Mając te wszystkie wiadomości możemy się pokusić o napisanie fragmentu programu głównego inicjującego licznik i układ przerwań – listing 3.

Listing 4. Funkcja delay
```
void delay(void){
int i;
  for(i=0;i<20000;i++);
}
```

Listing 5. Program sterujący miganiem LED
```
void delay();
void main (void){
  OSCCON=0x63; //częstotliwość 4MHz
  ADCON1=0x0f;//wyłączenie wejść przetwornika ADC
  TRISA=0xFE; //RA0 wyjściowa, reszta wejścia
  while(1){
    LATA=0;
    delay();
    LATA=1;
    delay();
  }
  while(1);
}

void delay(void){
unsigned int i;
  for(i=0;i<40000;i++);
}
```

Listing 6. Inicjalizacja Licznika Timer0 i układu przerwań
```
TMR0H=0xFC;       //wartość początkowa TMR0
TMR0L=0x18;       //dla odliczania 1000 impulsów
T0CON=0x88;       //włącz Timer0, 16bitow, bez preskalera
TMR0IE=1;         //odblokuj przerwania od przepełnienia Timer0
PEIE=1;           //odblokuj przerwania od peryferii (opcja)
GIE=1;            //odblokuj systemu przerwań
```

Po wykonaniu inicjalizacji w programie musi być umieszczona funkcja obsługi przerwania. Szkielet takiej funkcji dla przerwania od licznika Timer0 został pokazany na listingu 4.

Listing 7. Szkielet funkcji obsługi przerwania
```
static void interrupt int_tmr0(void){
  TMR0H=0xFC; //ładowanie licznika
  TMR0L+=0x18;
  TMR0IF=0; //zerowanie flagi przerwania
}
```

Listing 8. Deklarowanie zmiennych i obsługa przerwania
```
//definicje zmiennych bitowych
struct{
  char tms:1;
}bits;

#define tms bits.tms
volatile unsigned int mSek=0;              //odliczanie milisekund
static void interrupt int_tmr0(void){
  TMR0H=0xFC;
  TMR0L=0x18;
  if(tms){                                 //odliczanie czasu kiedy tms=1
  --mSek;
  if(mSek==0)
  tms=0;}                                  //koniec odliczania tms=0
  TMR0IF=0;
}
```

Na początku ładujemy licznik ponownie wartością tak by zliczał 1000 impulsów do przepełnienia. Jeżeli zależy nam na jak najdokładniejszym odliczaniu trzeba pamiętać, o tym, że od momentu przyjęcia przerwania do momentu kiedy procedura obsługi załaduje nowa wartość do TMR0H i TMR0L mija określony czas. Jeżeli wpiszemy do licznika wartość odpowiadającą zliczeniu 1000impulsów, to ten czas doda się do czasu odliczania 1 milisekundy. Żeby to skorygować do młodszej części TMR0L nie wpisujemy 0x18, tylko dodajemy 0x18. W ten sposób odejmujemy do doliczenia taki czas jaki licznik zliczył od przepełnienia do załadowania nowa wartością. Załóżmy, że w czasie od przepełnienia do momentu załadowania do TMR0L licznik zliczył 23 impulsy, czyli upłynął czas 23 μs (dla Fosc=4 MHz). Do TMR0L zapiszemy 23 dzies.=0x17+0x18=0x2F. Licznik do przepełnienia zliczy więc o te 23 impulsy mniej. Ten sposób wymaga sprawdzenia czy po korekcji nie ma przeniesienia po dodaniu TMR0L (czy TMR0L się nie przepełnił). Jeżeli tak, to trzeba zwiększyć o 1 licznik TMR0H. W naszym przypadku wartość wpisywana do TMR0L jest mała i tego sprawdzenia tu nie robię. Ostatecznie można ładować licznik bezpośrednio godząc się z pewną niedokładnością.

Listing 9. Procedura opóźnienia
```
void delay(unsigned int msek){
unsigned int i;
  mSek=msek;
  TMR0H=0xFC;  //wartość początkowa TMR0
  TMR0L=0x18;
  tms=1;
  while(tms);
}
```

W ten sposób utworzyliśmy „mechanizm tykającego zegarka", który zostanie zatrudniony do odliczania czasu z rozdzielczością 1 ms. Można tę rozdzielczość zwiększyć, ale przerwania zgłaszane zbyt często powodu-

Listing 10. Program migający diodą LED

```
#include <htc.h>
__CONFIG(1,IESODIS&FCMDIS&RCIO);
__CONFIG(2,BORDIS&PWRTEN&WDTDIS);
struct{
  char tms:1;
}bits;
#define tms bits.tms

void delay(unsigned int msek);
volatile unsigned int mSek=0;
static void interrupt int_tmr0(void){//obsługa przerwania
  TMR0H=0xFC;//ładowanie licznika
  TMR0L=0x18;
  if(tms){
  --mSek;
  if(mSek==0)  //odliczanie skończone
  tms=0;}
  TMR0IF=0;
}

void main(void){
  OSCCON=0x63;  //Fosc=4MHz
  ADCON1=0x0f;  //wszystkie porty cyfrowe
  TRISA=0;  //PORTA Wyjściowy
  TMR0H=0xFC;  //wartość początkowa TMR0
  TMR0L=0x18;
  T0CON=0x88;  //włącz Timer0, 16bitow, bez preskalera
  TMR0IE=1;  //odbl. przerwnia od przepełnienia Timer0
  PEIE=1;  //odbl. przerwań od peryferii
  GIE=1;  //odblokowanie systemu przerwań
  PORTA=0;
  while(1){
    LATA=0;  //RA0=0
    delay(500);  //czekaj 500msek
    LATA=1;  //RA0=1
    delay(500);
  }
}

void delay(unsigned int msek){
unsigned int i;
  mSek=msek;  //liczba milisekund
  TMR0H=0xFC;  //wartość początkowa TMR0
  TMR0L=0x18;
  tms=1;
  while(tms);

}
```

ją, że mikrokontroler zaangażowany w ich obsługę nie ma czasu na program główny. W naszym przypadku pomiędzy kolejnymi przerwaniami mija 1000 cykli maszynowych. To wystarczy, by obsłużyć dobrze napisaną procedurę obsługi przerwania i mieć dość czasu na program główny. Dobrą zasadą jest tak projektować procedury obsługi przerwań, by wykonywały się tak szybko, jak to możliwe.

Odmierzanie czasu będzie polegało na zliczaniu przerwań. Najlepiej żeby zajęła się tym procedura obsługi przerwania dekrementując globalną, najlepiej 16 bitową zmienną (**listing 4**).

Oprócz deklaracji zmiennej *mSek* zliczającej milisekundy opóźnienia zadeklarujemy bitowy znacznik *tms*. Może to deklaracja „na wyrost" dla pojedynczego bita, ale pozwala na użycie w programie większej liczby znaczników. Zmienna *mSek* jest zdeklarowania jako *volatile*, żeby kompilator nie poddawał jej procesowi optymalizacji. To jest dobry zwyczaj deklaracji zmiennych globalnych używanych w procedurach przerwania.

Modyfikacja zmiennej (dekrementacja) jest wykonywana tylko wtedy, kiedy jest ustawiony znacznik *tms*. Po wyzerowaniu zmiennej *mSek* znacznik *tms* jest zerowany automatycznie przez procedurę obsługi przerwania. Mamy już wszystko, żeby napisać procedurę opóźnienia odliczająca czas z rozdzielczością 1 ms (umieszczono ją na **listingu 5**).

Na początku do zmiennej *mSek* jest wpisywana wartość argumentu. Potem do licznika jest ładowana wartość początkowa do zliczania 1 ms. Po ustawieniu bitu *tms* procedura czeka na jego wyzerowanie przez procedurę obsługi przerwania. A jak wiemy, *tms* jest automatycznie zerowany po wyzerowaniu *mSek*, czyli po odliczeniu zadanej liczby przerwań, każde co 1 ms.

Ten sposób odmierzania czasu ma jeszcze jedną zaletę – umożliwia proste budowanie funkcji timeout'ów. Procedura inicjalizacji *timeout* jest taka, jak na listingu 5, trzeba tylko usunąć nieskończoną pętlę oczekiwania na wyzerowanie *tms*. To czy ta zmienna jest wyzerowana jest sprawdzane w miejscu, w którym zależy nam na określeniu czy już upłynął czas określony na wykonanie jakiejś czynności, na przykład, czy w czasie 200 ms odebrano kompletną wiadomość przez interfejs RS232. Odliczanie czasu odbywa się w tle i nie przeszkadza w odbieraniu znaków z modułu USART.

Kompletny program migający diodą LED jest pokazany na **listingu 6**. Proste zadanie polegające na zmuszeniu mikrokontrolera do cyklicznego zapalania i gaszenia diody LED pozwoliło nam na poznanie sposobu taktowania mikrokontrolera, budowy portów, działania licznika Timer0 i przynajmniej częściowo działania mechanizmu przerwań.

Tomasz Jabłoński
tomasz.jablonski@ep.com.pl