Temat 8. Programowanie mikrokontrolerów z rodziny PIC16 w języku C przy użyciu HI-TECH C® for PIC10/12/16

Spis treści do tematu 8

8.1. Porównane języków C i C++

Język C++ został zaprojektowany jako rozszerzenie języka C.

Najważniejsze właściwości języka C++, których brak w języku C:

1). Brak programowania obiektowego - brak słów kluczowych m.in.:
class, **private**, **protected**, **public**, **this**, **virtual**.
W strukturach w języku C nie można definiować metod.

2). Brak strukturalnej obsługi wyjątków z udziałem słów kluczowych:
catch, **try**, **throw**.
Kody błędów są przekazywane typowo przez wartości funkcji i zmienne globalne.

3). Brak przeładowywania function i operatorów. W konsekwencji nie istnieją przeładowane operatory **>>** oraz **<<** do operacji na wejściu **cin** i wyjściu **cout**.
Używamy funkcji z biblioteki **stdio.h** takich jak **scanf**, **printf**...

4). Brak operatorów **new** oraz **delete**. Do dynamicznego zarządzania pamięcią używa się funkcji bibliotecznych, np.: **malloc**, **free**.

5). Brak szablonów klas i funkcji definiowanych słowem kluczowym **template**.

Programy w języku C zazwyczaj mogą być skompilowane także kompilatorem C++.
W nowszych standardach języka C wprowadzono jednak słowa kluczowe nieobecne w standardzie języka C++, np. restrict, _Complex.
Ponadto w nowszych standardach języka C++ słowa kluczowe auto i register zmieniły znaczenie, co nie nastąpiło w języku C.

c.d. - porównane języków C i C++

Do programowania najprostszych mikroprocesorów i mikrokontrolerów najczęściej używany jest język C, natomiast język C++ nie jest odpowiedni. Główne powody:

1). Język C++ ukrywa przed programistą odwołania do danych w klasach i strukturach przez wskaźnik **this**. W prostych urządzeniach odwołania przez wskaźnik wymagają znacznie więcej instrukcji niż bezpośrednie adresowanie danych. Język C wymaga od programisty bardziej świadomego używania wskaźników.

2). Duży rozmiar kodu bibliotek standardowych języka C++ w porównaniu do bibliotek języka C realizujących analogiczne operacje.

3). Kompilator C++ często dodaje do programu kod, który nie ma widocznego odpowiednika w tekście źródłowym programu – niejawne wywołania konstruktorów, destruktorów, operatorów konwersji. W języku C kod programu wynika bardziej bezpośrednio z tekstu źródłowego programu.

c.d. - porównane języków C i C++

Oba języki C i C++ posiadają **preprocesor**, tzn. zestaw dyrektyw do wstępnej obróbki kodu źródłowego zanim rozpocznie się kompilacja. Dyrektywy:

#include … - wstawienie tekstu innego pliku w miejscu wywołania tej dyrektywy,
składnia: #include<nazwa_pliku> lub #include "nazwa_pliku"

#define … - definiuje stałe i makroinstrukcje

1). Przykład definicji stałej:

```
#define powitanie "Witamy w Instytucie Fizyki PL"
puts(powitanie);   // powitanie zostanie zastąpione tekstem
```

2). Przykład definicji bez podania wartości:

```
#define empty      // identyfikator empty będzie rozwijany jako brak znaków
```

3). Przykład definicji makroinstrukcji:

```
#define dzialanie(a, b, c) ((a)*(b) + (c))
int wynik = dzialanie(1+1, 3, 1); // obliczenie (1+1)*3+1 = 7 przed kompilacją
```

#undef … - usuwa defnicję stałej lub maroinstrukcji

Przykład:

```
#undef powitanie
// Od teraz identyfikator powitanie może być użyty do czegoś innego
#define powitanie "Witamy w Laboratorium elektroniki"
```

c.d. - porównane języków C i C++

Preprocesor, c.d. …

#ifdef … #else … #endif – warunkowe wstawienie tekstu źródłowego do programu

Przykład:

część opcjonalna

```
#define test

char nazwa[50] = "";
#ifdef test
   // jeśli testowanie programu, to o nic nie pytaj – użyj domyślnej nazwy
   strcpy(nazwa, "nazwa doyślna");
#else
   // zapytaj użytkownika o nazwę
   puts("Podaj nazwę: ");
   gets(nazwa);
#endif
// nazwa – zwiera domyślny łancuch znaków albo wprowadzony
…
```

#ifndef … #else … #endif – działa odwrotnie niż #ifdef, tzn. wstawia pierwszy blok tekstu jeśli stała jest niezdefiniowana.

c.d. - porównane języków C i C++

Preprocesor, c.d. …

#if … #else … #endif – obliczenie warunku stojącego po #if i warunkowe wstawienie tekstu źródłowego do programu

Przykład:

```
#define rozmiar_domyslny 10

#if (0 < rozmiar_domyslny && rozmiar_domyslny <= 1000)
   double wyniki[rozmiar_domyslny];
#else
   #error rozmiar_domyslny musi wynosic od 1 do 1000
#endif
…
```

#error … – przerywa dalsze przetwarzanie programu i wyświetla podany komunikat.

c.d. - porównane języków C i C++

Preprocesor, c.d. …

Problem:
W wieloplikowych projektach często dochodzi do wielokrotnego wstawienia tego samego pliku nagłókowego komendą #include.

Rozwiązanie:
Żeby uniknąć ponownego przetwarzania wstawionych już definicji stosuje się:

```
#ifndef __PIC16F84A_H
#define __PIC16F84A_H
  #define CONFIG              0x2007
  #define FOSC_XT             0xFFFD
  //… tutaj treść pliku nagłówkowego, która nie powinna być wstawiana wielokrotnie
#endif
```

c.d. - porównane języków C i C++

Oba języki C i C++ posiadają ten sam zestaw **operatorów bitowych i logicznych**

typ operacji	operator bitowy	op. bitowy z przypisaniem	operator logiczny
AND	&	&=	&&
OR	|	|=	||
XOR	^	^=	
NOT	~	^= -1	!

Operatory bitowe wykonują równolegle operacje na poszczególnych bitach argumentów, np.:

```
  0b00001100
& 0b00001010
= 0b00001000

~ 0b00001100
= 0b11110011
```

Operatory logiczne traktują cały niezerowy argument jak true a zerowy jak false, np.:

```
  0b00001100        true
&& 0b00001010    && true
 = 0b00000001     = true

 ! 0b00001100     !true
 = 0b00000000     = false
```

Wynikiem jest zawsze 1 (true) albo 0 (false)

c.d. - porównane języków C i C++

W języku C++ istnieją słowa kluczowe:
bool – typ danych tylko do przechowywania wartości logicznych,
false – oznacza stałą fałsz,
true – oznacza stałą prawda.

Słów kluczowych nie można użyć w innym znaczeniu, np. nazwy zmiennych, funkcji.

W języku C wyżej wymienione słowa nie są kluczowe.
Można skorzystać z pliku nagłówkowego stdbool.h który definiuje:

```
#define true  1
#define false 0
#define bool char   // może być także unsigned char, int...
```

W obu językach C i C++ można używać stałych i zmiennych typów liczbowych w charakterze wartości logicznych bez potrzeby jawnych konwersji typów, np.:

```
bool wynik = true;
int i = wynik;
wynik = wynik && 77;
char c = i || 77;
```

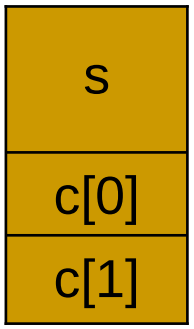
c.d. - porównane języków C i C++

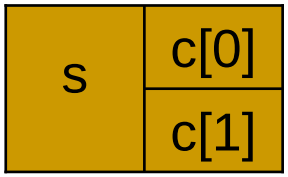
Oba języki C i C++ posiadają struktury i unie definiowane jako:

```
struct [nazwa_typu] {
  // lista pól struktury, np.:
  short s;
  char c[2];
} [zmienna1, zmienna2,...];
```

```
union [nazwa_typu] {
  // lista pól struktury, np.:
  short s;
  char c[2];
} [zmienna1, zmienna2,...];
```

układ zmiennej w pamięci:





UWAGI:

1). Tylko w języku C++ w strukturze można definiować metody (jak w klasie).

2). Tylko w języku C++ struktura może dziedziczyć po innej strukturze/klasie.

3). W języku C++ można modyfikować dostęp do składników struktury słowami kluczowymi public, protected i private, których nie ma języku C.

Przykład wykorzystania unii w programowaniu mikrokontrolerów:

Przykładowo, w pliku nagłówkowym pic16f84a.h można znaleźć m.in. unię:

```
volatile union {
    struct {
        unsigned    C           : 1;
        unsigned    DC          : 1;
        unsigned    Z           : 1;
        unsigned    nPD         : 1;
        unsigned    nTO         : 1;
        unsigned    RP          : 2;
        unsigned    IRP         : 1;
    };
    struct {
        unsigned                : 5;
        unsigned    RP0         : 1;
        unsigned    RP1         : 1;
    };
} STATUSbits @ 0x003;
```

Wniosek: do bitów wyboru bieżącego banku pamięci danych można odwołać się:
STATUSbits.RP0 – dostęp tylko do bitu RP0 (analogicznie dostęp do RP1),
STATUSbits.RP – zmienna dwubitowa złożona z RP1 i RP0.

8.2. Używanie kompilatora

Kompilator HI-TECH C® for PIC10/12/16 w wersji LITE (z wyłączonymi niektórymi optymalizacjami) może być używany za darmo bez ograniczeń jako:

1) kompilator wywoływany przez środowisko MPLAB IDE, co wymaga utworzenia w tym środowisku projektu (tworzenie projektu opisano dokładniej w instrukcji do ćwiczenia laboratoryjnego E59).

2) aplikacja wywoływana z linii komend:

PICC [opcje] pliki [biblioteki]

gdzie typ pliku jest rozpoznawany po rozszerzeniu:

.c - plik źródłowy w języku C,

.as, .asm - plik źródłowy w asemblerze,

.obj, .lib - pliki ze skompilowanym kodem.

Wygenerowany kod programu trafia do pliku .hex

Wśród opcji trzeba podać przynajmniej typ urządzenia, np. dla układu PIC16F819:

PICC --CHIP=16F819 program.c

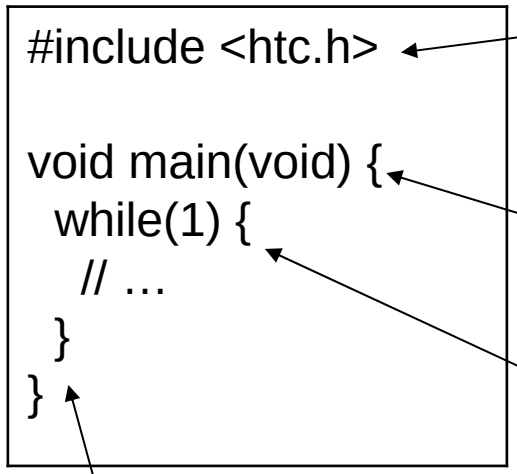
Wszystkie opcje kompilacji są podane w literaturze [1], rozdział 2.7.

W celu uzyskania listy możliwych opcji wywołać: PICC --HELP

Argumenty wywołania można umieścić w pliku, np..: PICC @xyz.cmd

8.3. Struktura najprostszego programu

Typowa struktura programu w języku C dla mikrokontrolerów PIC10/12/16:



Plik htc.h dzięki wykorzystaniu kompilacji warunkowej włącza do programu plik dedykowany wybranemu mikrokontrolerowi (wybrany np. w opcjach kompilatora).

2 x void, bo nie ma żadnego systemu, który mógłby przesłać argument lub odebrać wynik.

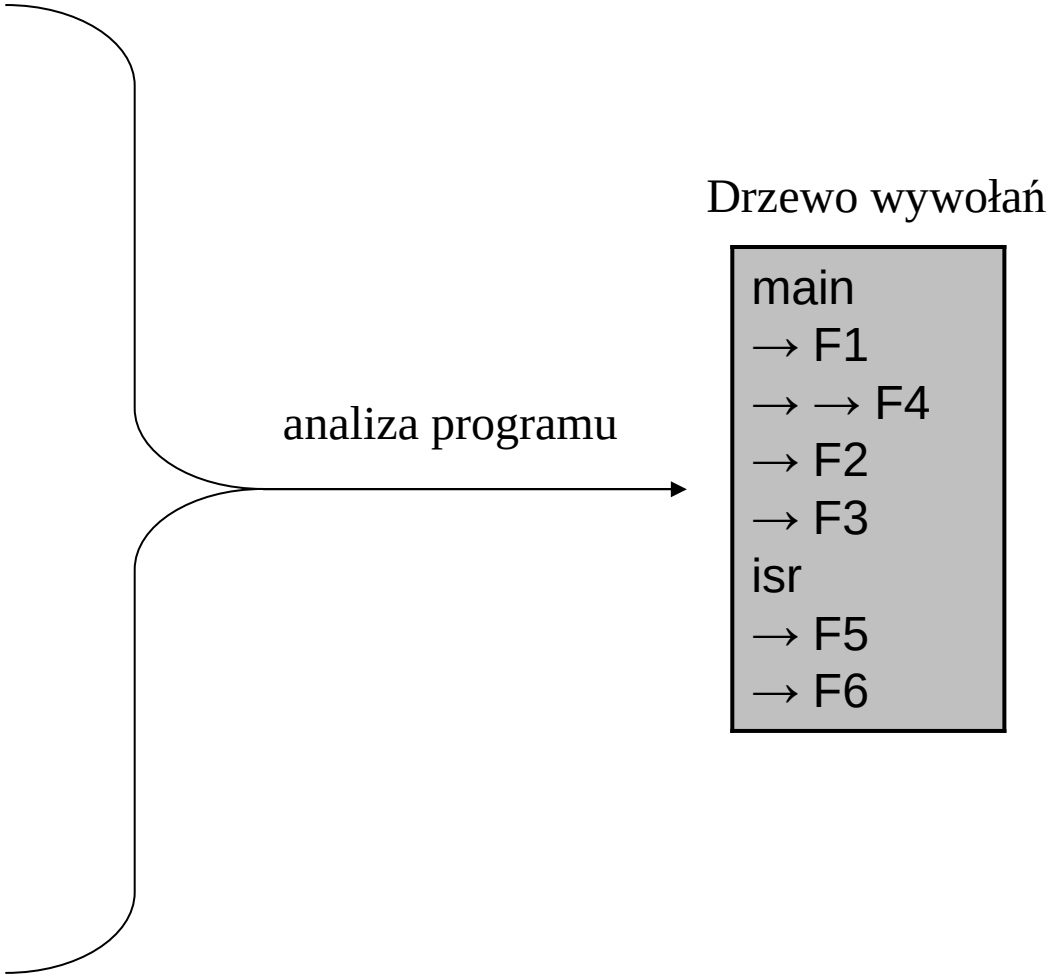
Potrzebna nieskończona pętla, bo nie ma żadnego systemu operacyjnego, do którego można wyjść z programu.

Można pominąć return, bo funkcja main nie powinna nigdy zakończyć się.

8.4. Stos kompilowany

Stos sprzętowy PIC10/12/16 pozwala tylko na przechowywanie adresów powrotów z funkcji. Lokalne dane są umieszczane na „kompilowanym stosie”, tzn. pod ustalonymi adresami, które kompilator przydziela na podstawie analizy drzewa wywołań, np.:

```
F4(...) { ... }
F5(...) { ... }
F6(...) { ... }

void interrupt isr(void) {
  F5(...);
  F6(...);
}

F1(...) {  F4(…);  }
F2(...) { ... }
F3(...) { ... }

void main(void) {
  F1(...);
  F2(...);
  F3(...);
}
```

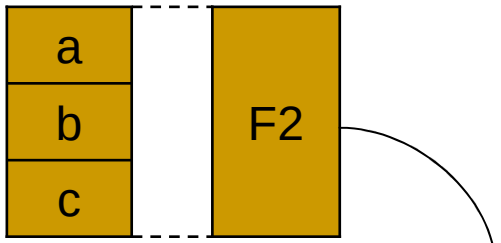


c.d. – stos kompilowany

1). Każdej funkcji jest przydzielany blok pamięci dla jej parametrów oraz lokalnych i tymczasowych zmiennych, np.:

```
void F2(int a, int b ) {
   char c;
   ...
}
```

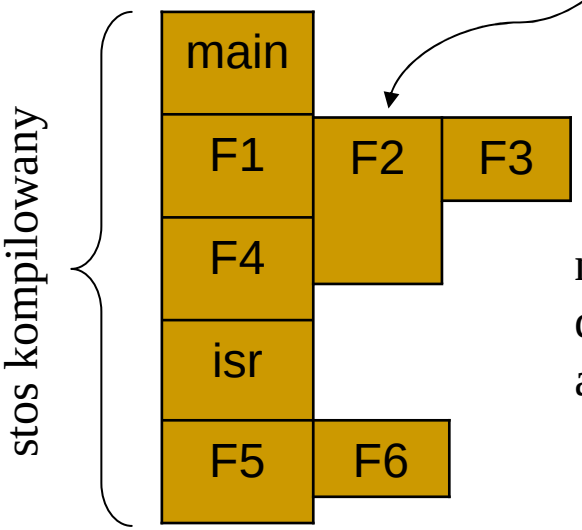
blok pamięci funkcji F2



2). Analiza drzewa wywołań

```
main
→ F1
→ → F4
→ F2
→ F3
isr
→ F5
→ F6
```

3). Przydział adresów



nakładanie się bloków dla niejednocześnie aktywnych funkcji

UWAGA: rekurencyjne wywołania funkcji nie są wspierane przez kompilator.

8.5. Rozszerzenia języka ANSI C

Kompilator HI-TECH C® for PIC10/12/16 oprócz standardu ANSI C oferuje szereg rozszerzeń dostosowanych do architektury mikrokontrolerów.

8.5.1. Literały stałe

Oprócz formatów wartości stałych przewidzianych w standardzie ANSI C kompilator HI- TECH C® umożliwia także zapis stałych binarnych:

system liczbowy	Format	Przykład
binarny	0b*liczba*, 0B*liczba*	0b10011010
ósemkowy	0*liczba*	0763
dziesiętny	*liczba*	129
szesnastkowy	0x*liczba*, 0X*liczba*	0x2F

kolor czerwony – format niestandardowy

8.5.2. Podstawowe typy danych

Typ danych	Rozmiar - liczba bitów	Typ arytmetyczny
bit	1	całkowity bez znaku
signed char	8	całkowity ze znakiem
unsigned char	8	całkowity bez znaku
signed short	16	całkowity ze znakiem
unsigned short	16	całkowity bez znaku
signed int	16	całkowity ze znakiem
unsigned int	16	całkowity bez znaku
signed short long	24	całkowity ze znakiem
unsigned short long	24	całkowity bez znaku
signed long	32	całkowity ze znakiem
unsigned long	32	całkowity bez znaku
float	24 albo 32	zmiennoprzecinkowy
double	24 albo 32	zmiennoprzecinkowy

kolor czerwony – typy niestandardowe

8.5.3. Rejestr konfiguracyjny

Wartość rejestru konfiguracyjnego można ustawić na etapie programowania przy użyciu makrodefinicji _ _CONFIG zdefiniowanej w pliku nagłówkowym htc.h, np.:

```
#include<htc.h>
_ _CONFIG(WDTDIS & HS & UNPROTECT);
```

Uwaga: ten rejestr nie jest dostępny wewnątrz wykonywanego programu.

Plik htc.h dzięki wykorzystaniu kompilacji warunkowej włącza do programu plik dedykowany wybranemu mikrokontrolerowi (wybrany np. w opcjach kompilatora).
Pliki dedykowane nazywają się jak model mikrokontrolera, np.:
pic16f84a.h - dla modelu PIC16F84A
pic16f819.h - dla modelu PIC16F819

Stałe symboliczne używane jako argumenty _ _CONFIG(…) są zdefiniowane razem z opisami w plikach nagłówkowych dedykowanych dla poszczególnych modeli.

8.5.4. Dostęp do rejestrów specjalnych podczas wykonywania programu

Dostęp do specjalnych rejestrów leżących w przestrzeni adresowej wej./wyj. jest możliwy poprzez zmienne o ustalonych adresach absolutnych, np.:

```
volatile unsigned char TRISA    @ 0x085;
volatile unsigned char PORTA  @ 0x005;
volatile bit TRISA4                 @ ((unsigned)&TRISA*8)+4;
volatile bit RA4                      @ ((unsigned)&PORTA*8)+4;
```

Uwaga: operator @ jest rozszerzeniem specyficznym dla kompilatora Hi-Tech C.

Zamiast definiować samodzielnie takie zmienne warto skorzystać z gotowego pliku nagłówkowego htc.h, np:

```
#include<htc.h>
PORTA = 0;                  // wszystkie wyjścia w stanie 0
TRISA = 0b11110000;     // ustaw linie RA0...RA3 jako wyjścia, reszta wejścia
If (RA4) { … }              // wykonaj jeśli stan 1 na wejściu RA4
If (PORTA & 0b00010000) { … } // działa jak wyżej
```

Dla poszczególnych typów mikrokontrolerów istnieją pliki nagłówkowe opisujące ich specyficzne właściwości, np.: pic16f84a.h, pic16f819.h. Plik odpowiedni dla wybranego urządzenia zostanie wstawiony pośrednio poprzez #include<htc.h>.

8.5.5. Wejście i wyjście standardowe

Wejście standardowe stdin oraz wyjście standardowe stdout nie są zdefiniowane! Jeżeli programista chce używać funkcji operujących na standardowym wejściu, to musi sam zdefiniować elementarne funkcje pobierające jeden znak:

```
char getch(void) { ... }
char getche(void) { ... }
```

Funkcja getch() odczytuje 1 bajt z wejścia standardowego, natomiast getche() dodatkowo zapisuje echo wprowadzanych znaków na wyjście standardowe. Przykładowo funkcja char * gets(char * s) jest implementowana jako powtarzające się wywołania getche().

Wykorzystanie funkcji zapisujących standardowe wyjście, np. printf(), wymaga zdefiniowania przez programistę własnej funkcji putch(), która musi mieć postać

```
void putch(char c) { ... }
```

8.6. Optymalizacja kodu programu

Oprócz wykorzystania przełączników kompilatora dotyczących optymalizacji kodu można podać kilka reguł dotyczących przygotowania tekstu źródłowego programu.

8.6.1. Wybór optymalnego typu danych

Mikrokontrolery PIC16Fxxx zawierają 8-bitową jednostkę arytmetyczno logiczną, posiadającą także specjalne instrukcje do operacji na bitach. Ponieważ wszystkie operacje na zmiennych dłuższych niż 1 bajt (8 bitów) są realizowane jako sekwencja operacji na bajtach, należy preferować typy całkowite o najmniejszym możliwym zakresie, np.:

1). wymagany zakres **0...0xFFFFFF** → optymalny typ **unsigned short long**
(zamiast standardowego typu **unsigned long**),

2). wymagany zakres **-8388608...8388607** → optymalny typ **short long**
(zamiast standardowego typu **long**),

3) wymagany zakres **0...255** → optymalny typ **unsigned char** (zamiast **int**),

4) wymagany zakres **-128...127** → optymalny typ **signed char** (zamiast **int**),

5) wymagane **tylko wartości 0 oraz 1** → optymalny typ **bit** (zamiast **bool** / **int**).

Ponadto typy całkowitoliczbowe należy preferować przez typami zmiennoprzecinkowymi.

8.6.2. Wykorzystanie operatorów

Mikrokontrolery PIC16xxx nie posiadają instrukcji do bezpośredniego wykonywania operacji mnożenia, dzielenia i modulo. Jeżeli potrzebna jest wysoka szybkość działania programu, w miarę możliwości należy ograniczać użycie operatorów języka C rozwijanych w kodzie programu jako wywołanie funkcji złożonych z wielu instrukcji:

***, /, %, *=, /=, %=**

Przesunięcia bitów w prawo oraz w lewo są wykonywane bezpośrednio tylko dla przesunięć o 1 bit. Należy ograniczać użycie przesunięć o wiele bitów oraz przesunięć o liczbę bitów daną poprzez zmienną, np.:

```
zmienna1 >> 5;                  // powolne
zmienna1 << przesuniecie;       // zmienne
 zmienna1 << 1;                 // szybkie
```

8.6.3. Deklarowanie stałych

Deklarowanie wartości stałych zamiast zmiennych (tam gdzie to możliwe) pozwala ograniczyć wykorzystanie niewielkich zasobów pamięci RAM:

```
int zmienna = 3;      // zmienna w pamięci RAM,
const int stala = 3;  // stała może być umieszczona kodzie programu.
#define stala 3       // stała będzie umieszczona w kodzie programu.
```

Stałe łańcuchy znaków są przechowywane w pamięci kodu programu, a ich typem jest const char * (a nie char *). Tak więc definicja

```
char tablica[] = "znaki";
```

spowoduje wygenerowanie kodu inicjalizującego tablicę w pamięci RAM znakami skopiowanymi z pamięci kodu programu.

Jeśli tablica będzie tylko odczytywana, to można uniknąć zajmowania pamięci danych przez jej kopię używając wskaźnika do danych const, np.:

```
const char * tablica = "znaki";  // tablica pozostaje tylko w pamięci kodu
```

8.6.4. Wykorzystanie wskaźników

Dostęp do zmiennej lub stałej poprzez wskaźnik generuje dodatkowe instrukcje maszynowe w porównaniu do odwołania bezpośredniego. Ponadto kompilator rozróżnia kilka typów wskaźników o różnym zasięgu i odpowiednio różnym narzucie na kod programu:

- 8-bitowe wskaźniki do danych w dwóch kolejnych bankach, np. banki 0 i 1,
- 8-bitowe wskaźniki do kodu programu o zakresie do 256 bajtów,
- 16-bitowe wskaźniki do danych w całej przestrzeni adresowej danych,
- 16-bitowe wskaźniki do kodu programu w całej przestrzeni adresowej programu,
- 16-bitowe wskaźniki do całej przestrzeni adresowej danych oraz kodu programu.

Programista nie może bezpośrednio zadeklarować zasięgu wskaźnika, ale może go przewidzieć. Kompilator automatycznie wybiera zasięg na podstawie analizy użycia wskaźnika.

c.d. - wykorzystanie wskaźników

Przykład 8.1:

8-bitowy wskaźnik „ip” do danych	16-bitowy wskaźnik „ip” do danych i kodu

// dwie zmienne w tym samym banku
char i;
char j[4];
 	
void main(void) {
  char * ip = &i;
  // jakieś operacje na *ip
  ip = &j[0];
  // jakieś operacje na *ip
}
	// zmienna w banku 0 oraz stała w kodzie
	char i;
	const char * j = "Ala";
	
	void main(void) {
	  const char * ip = &i;
	  // jakieś operacje na *ip
	  ip = &j[0];
	  // jakieś operacje na *ip
	}

8.6.5. Funkcje z rodziny printf

Funkcje z rodziny printf z biblioteki standardowej są generowane ze specjalnego wzorca po analizie kodu źródłowego programu. Przykładowo, jeśli program zawiera tylko jedno wywołanie:

```
int var = 77;
printf("input is: %d", var);
```

to zostanie dołączony tylko kod funkcji do konwersji liczby całkowitej typu int na łańcuch znaków z liczbą w postaci dziesiętnej.

Uwaga: jeżeli łańcuch znaków formatujących nie jest dany bezpośrednio jako stały łańcuch, to zostanie dołączony bardziej ogólny i obszerniejszy kod funkcji.

8.7. Pamięc EEPROM

Większość mikrokontrolerów PIC16Fxxx posiada 3 typy pamięci:
1) **FLASH** – dla kodu programu, słowa 14 bitowe,
2) (**S**)**RAM** – szybka pamięć dla danych, ulotna po zaniku zasilania, bajty po 8 bitów,
3) **EEPROM** – wolna ale trwała pamięć dla danych, bajty po 8 bitów,
   UWAGA: komórki EEPROM są nieobecne w przestrzeni adresowej danych, dostęp przez specjalne rejestry EEDATA, EEADR i EECON1 lub funkcje biblioteczne.

Początkowa zawartość pamięci EEPROM może być ustawiona na etapie programowania:

```
#include<htc.h>
_ _EEPROM_DATA(0, 1, 2, 3, 4, 5, 6, 7);
_ _EEPROM_DATA(8, 9, 10, 11, 12, 13, 14, 15);
```

Makrodefinicja przyjmuje 8 parametrów typu bajt. Wywołanie tej makrodefinicji może być powtarzane wiele razy w celu zdefiniowania wymaganej liczby początkowych bajtów w pamięci EEPROM.

c.d. - dostęp do pamięci EEPROM

Dostęp do pamięci EEPROM podczas wykonywania programu jest możliwy przy użyciu funkcji eeprom_write() oraz eeprom_read() do zapisu oraz odczytu jednego bajtu pod wskazanym adresem, np:

```
#include<htc.h>

void eetest(void) {
   unsigned char value = 1;
   unsigned char address = 0;

   // zapis do pamięci EEPROM
   eeprom_write(address, value);

   // odczyt z pamięci EEPROM
   value = eeprom_read(address);
   // ...
}
```

8.8. Dwustanowe porty wej./wyj.

Dwustanowe wejścia/wyjścia są pogrupowane w porty A, B, C, …, gdzie w jednym porcie można kontrolować do 8 linii. Z każdym portem związane są po dwa 8-bitwe rejestry:

TRIS*x* – bity w tym rejestrze decydują o kierunku: 0 – wyjście, 1 – wejście,

PORT*x* – bity w tym rejestrze symbolizują odczytany/zapisany stan logiczny, gdzie *x* = A, B, C, D albo E.

Szczegółowy opis tych rejestrów był podany w Temacie 7, rozdział 7.5.

W kompilatorze HI-Tech C włączenie pliku nagłówkowego

```
#include<htc.h>
```

umożliwa dostęp do każdego z tych rejestrów na 3 sposoby:

PORTA	– dostęp do całego bajtu (8-bitów),
RA0, RA1, RA2, …	– dostęp do wybranego bitu w PORTA,
PORTAbits.RA0, PORTAbits.RA1, …	– działa jak wyżej.

TRISA	– dostęp do całego bajtu (8-bitów),
TRISA0, TRISA1, TRISA2, …	– dostęp do wybranego bitu w TRISA,
TRISAbits.TRISA0, TRISAbits.TRISA1, …	– działa jak wyżej.

Analogicznie dostęp do rejestrów PORTB, TRISB, …

c.d. - Dwustanowe porty wej./wyj.

Fragment z pliku nagłówkowego pic16f84a.h:

```
// Register: PORTA
volatile unsigned char          PORTA           @ 0x005;
// bit and bitfield definitions
volatile bit RA0                @ ((unsigned)&PORTA*8)+0;
volatile bit RA1                @ ((unsigned)&PORTA*8)+1;
volatile bit RA2                @ ((unsigned)&PORTA*8)+2;
volatile bit RA3                @ ((unsigned)&PORTA*8)+3;
volatile bit RA4                @ ((unsigned)&PORTA*8)+4;
#ifndef _LIB_BUILD
volatile union {
    struct {
        unsigned        RA0 : 1;
        unsigned        RA1 : 1;
        unsigned        RA2 : 1;
        unsigned        RA3 : 1;
        unsigned        RA4 : 1;
    };
} PORTAbits @ 0x005;
```

c.d. - Dwustanowe porty wej./wyj.

Przykład 8.2 – próba zapisu do wejścia:

```
#include <htc.h>

void main(void) {
    PORTA = 0;  // początkowo wyjście w stanie 0
    nRBPU = 0;  // włącz rezystory do +5 V w porcie B
    unsigned char wewn_q = 0;
    do{
        unsigned char wejscie_A = !RB0;
        unsigned char wejscie_B = !RB1;

        wewn_q = (!wejscie_A && !wejscie_B) || (!wejscie_B && wewn_q);
        RA0 = wewn_q;
    }while(1);
}
```

Zgubione:
TRISA = 0b11111110;

Brak skutku, bo domyślnie wszystkie linie są ustawione jako wejścia

c.d. - Dwustanowe porty wej./wyj.

Przykład 8.3 – niekontrolowane zmiany wejść:

```
#include<htc.h>

void main(void) {
    TRISA = 0b11111110;// linia RA0 jako wyjście
    PORTA = 0; // początkowo wyjście w stanie 0
    nRBPU=0; // włącz rezystory do +5V w porcie B
    unsigned char wewn_q = 0;
    do{
        wewn_q = (RB0 && RB1)
                    || (RB1 && wewn_q);
        RA0 = wewn_q;
    }while(1);
}// main
```

Problem!

Dwa odczyty RB1 mogą dać inny wynik.

Trudno przeanalizować skutki.

```
do{
    unsigned char wejscie_A = RB0;
    unsigned char wejscie_B = RB1;

    wewn_q = (wejscie_A && wejscie_B)
                || (wejscie_B && wewn_q);
    RA0 = wewn_q;
}while(1);
```

Poprawione

Buforowanie wejść w pamięci RAM upraszcza analizę.

c.d. - Dwustanowe porty wej./wyj.

Przykład 8.4 – utrwalanie błędnych stanów przejściowych na wyjściach:

```
#include<htc.h>

void main(void) {
    TRISA = 0b11111100;// linie RA0 i RA1 jako wyjścia
    PORTA = 0; // początkowo wszystkie wyjścia w stanie 0
    nRBPU=0; // włącz rezystory do +5V w porcie B
    unsigned char Q1 = 1;
    unsigned char Q2 = 0;
    unsigned char BuforWej, newQ1;
    do{
        BuforWej = PORTB;
        newQ1 = !((BuforWej & 1) && Q2);
        Q2 = !((BuforWej & 2) && Q1);
        Q1 = newQ1;
        RA0 = Q1;
        RA1 = Q2;
    }while(1);
}// main
```

Problem!

Ryzyko utrwalenia błędnego stanu przejściowego na RA0. Wyjaśnienie na Rys. 7.18.

```
do{
    BuforWej = PORTB;
    newQ1 = !((BuforWej & 1) && Q2);
    Q2 = !((BuforWej & 2) && Q1);
    Q1 = newQ1;
    PORTA = Q1 | (Q2 << 1);
}while(1);
```

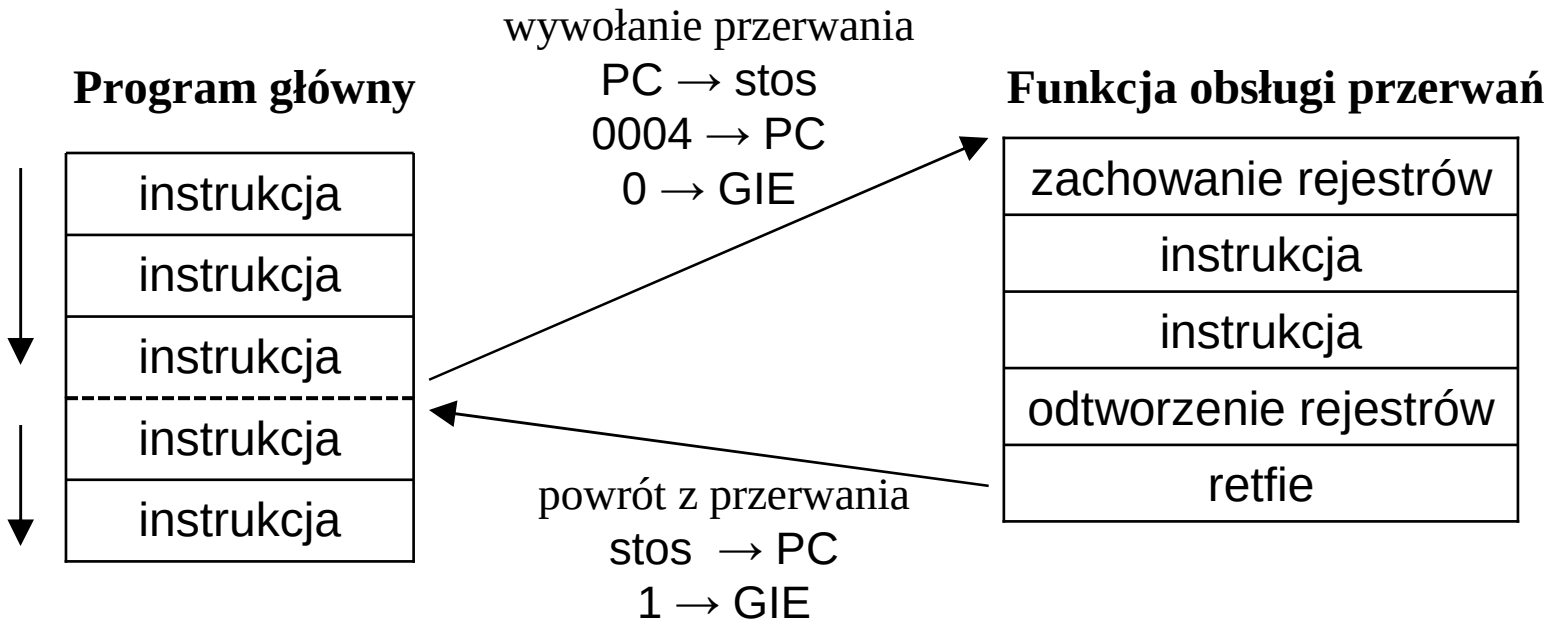
Poprawione

Ryzyko usunięte przez zapis całego bajtu.

8.9. Układ przerwań

Mikrokontrolery z rodziny PIC16Fxxx mają jednopoziomowy układ przerwań. Oznacza to, że wszystkie źródła przerwań mają jednakowy priorytet i powodują wywołanie jednej wspólnej procedury obsługi przerwań.

Schemat obsługi przerwania

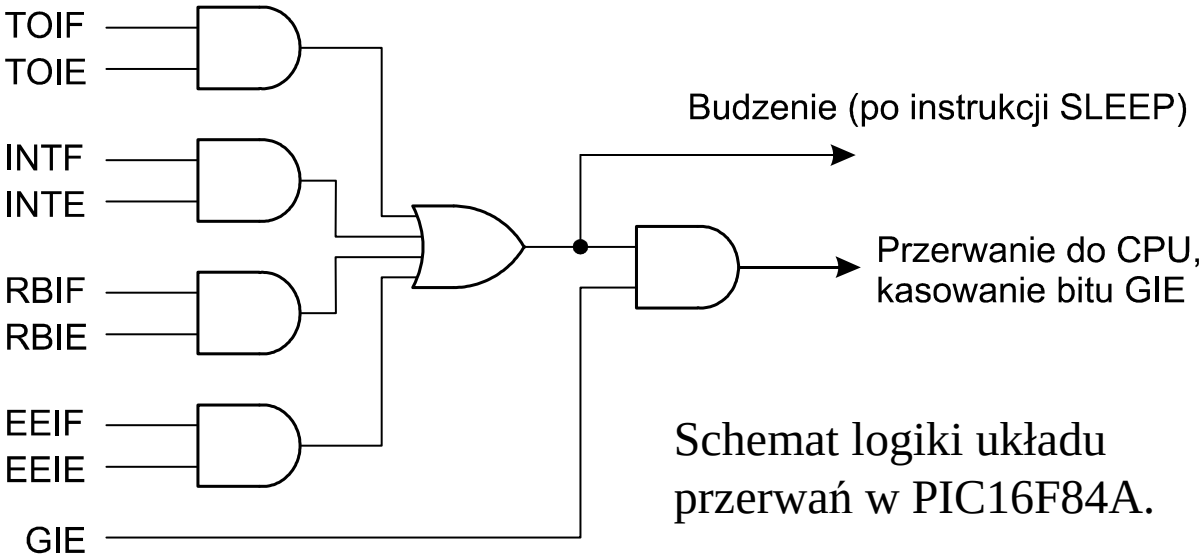


Kompilator HI-Tech C śledzi, które rejestry są używane w funkcji obsługi przerwań i w wywoływanych funkcjach (np. W, STATUS, FSR) i automatyczni generuje kod zachowujący oraz odtwarzający wartości tych rejestrów.

c.d. – układ przerwań

Lista przyczyn przerwań zależy od modelu mikrokontrolera.
Przykładowo, w PIC16F84A są to cztery źródła:

1. TIMER 0
2. Wejście INT (RB0)
3. Wejścia RB4…RB7
4. Koniec zapisu do EEPROM



Schemat logiki układu przerwań w PIC16F84A.

Z każdą przyczyną przerwania związane są dwa bity w rejestrach specjalnych:

- xxxIE (ang. *interrupt enable*) – bit zezwolenia na generowanie przerwania z danej przyczyny; przerwanie jest aktywne gdy bit ten jest równy 1,
- xxxIF (ang. *interrupt flag*) – znacznik ustawiany sprzętowo w stan 1 po każdym zgłoszeniu przerwania; zerowanie znacznika musi odbywać się w programowo.

Powyżej „xxx” symbolizuje różne oznaczenia poszczególnych przyczyn przerwań.
Bit GIE (ang. *global interrupt enable*) w stanie 0 blokuje wszelkie przerwania, natomiast w stanie 1 dopuszcza przerwania nie zamaskowane przez bity …IE.

c.d. – układ przerwań

Ponieważ system przerwań jest jednopoziomowy, wewnątrz funkcji obługi przerwań należy sprawdzać przyczyny wystąpienia przerwania, np.:

```
#include<htc.h>

volatile int tick_count = 0;

void interrupt tc_int(void) {
  if (T0IE && T0IF) {  // jeżeli przerwania TIMER 0 są włączone
                       // i przerwanie zostało zgłoszone
    T0IF = 0;          // wytrzyj flagę zgłoszenia przerwania
    ++tick_count;   // jakaś akcja
    return;
  }
  if (INTE && INTF) {  // jeżeli przerwania RB0 są włączone
                       // i przerwanie zostało zgłoszone
    INTF = 0;          // wytrzyj flagę zgłoszenia przerwania
    tick_count = 0;
    return;
  }
  // ... obsługa innego źródła przerwań
}
```

Słowo kluczowe interrupt konieczne w nagłówku funkcji obsługi przerwań.

c.d. – układ przerwań

Przerwania domyślnie są wyłączone. Żeby je włączyć należy ustawić maski odpowiednio wybranych przerwań oraz globalnie włączyć przerwania:

```
#include<htc.h>

T0IF = 0;   // wytrzyj maskę przerwania zegara 0
T0IE = 1;   // przerwania zegara 0 będą używane
ei();       // ogólne włączenie niezamaskowanych przerwań
// ...
di();       // ogólne wyłączenie przerwań
```

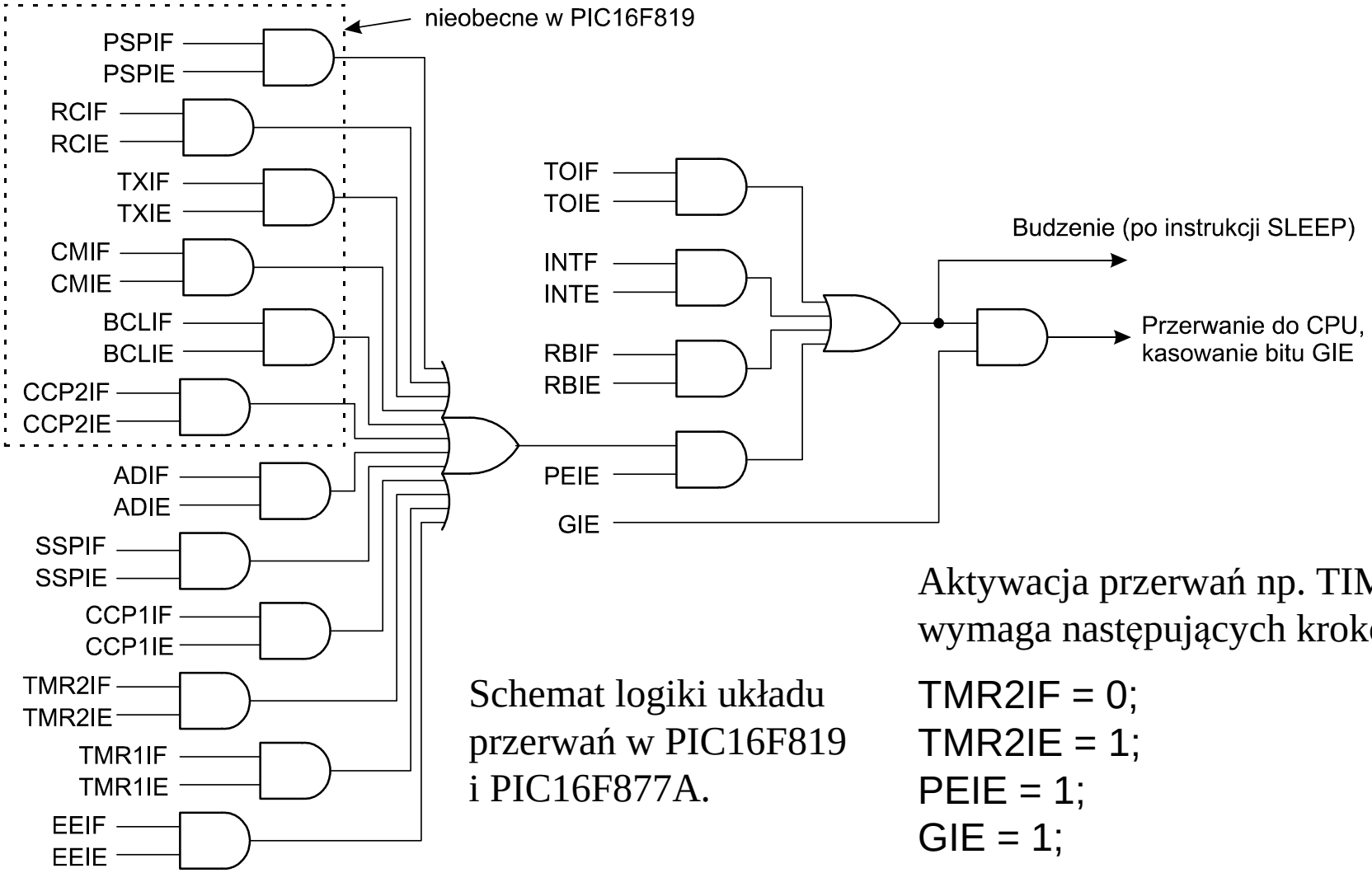
Uwaga 1: maski przerwań (np. T0IE) i flagi przerwań (np. T0IF) są specyficzne dla poszczególnych modeli mikrokontrolerów. Należy je sprawdzić w dokumentacji dla odpowiedniego urządzenia (tzw. „Data sheet").

Uwaga 2: Zamiast użycia ei() oraz di() można bezpośrednio zmieniać bit GIE z rejestru INTCON, który jest obecny w przestrzeni adresowej danych:

```
GIE = 1;    // równoważne ei();
// ...
GIE = 0;    // równoważne di();
```

c.d. – układ przerwań

W nowszych mikrokontrolerach istnieje więcej przyczyn przerwań. Oprócz indywidualnego oraz globalnego maskowania przerwań, cześć przerwań od układów peryferyjnych jest maskowana także bitem PEIE (w rej. INTCON).



Schemat logiki układu przerwań w PIC16F819 i PIC16F877A.

Aktywacja przerwań np. TIMER2 wymaga następujących kroków:

```
TMR2IF = 0;
TMR2IE = 1;
PEIE = 1;
GIE = 1;
```

8.10. Zegary

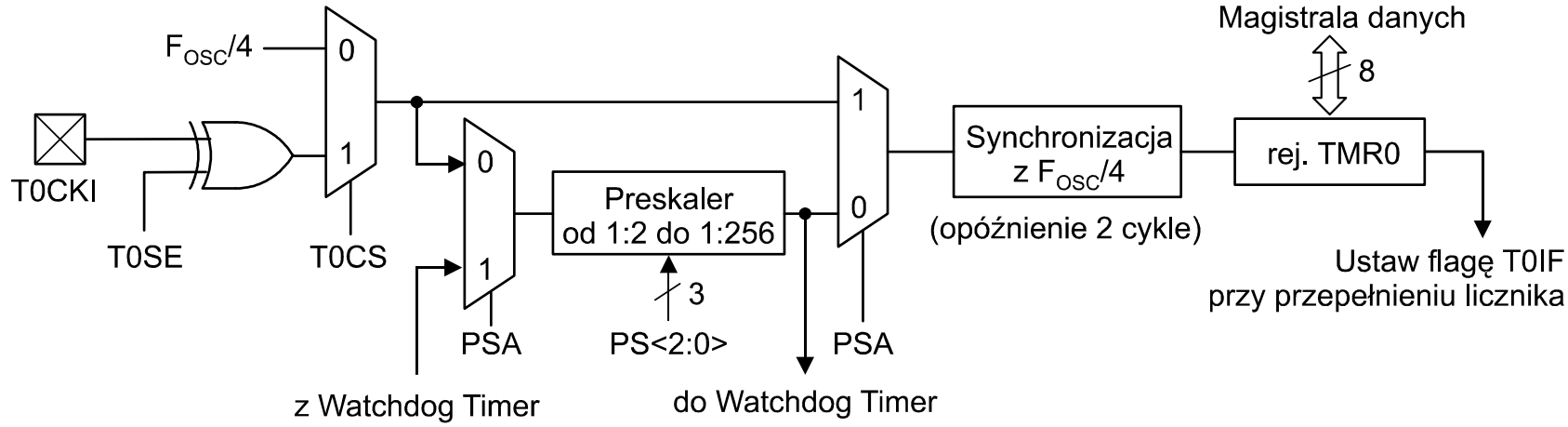
Mikrokontrolery z rodziny PIC16Fxxx mają od 1 do 3-ech zegarów sprzętowych, które można wykorzystać do odmierzania okresów czasu albo do zliczania impulsów.

Wyposażenie	Mikrokontroler dostępny w lab. techniki cyfrowej		
	PIC16F84A	PIC16F819	PIC16F877A
Timer0	+	+	+
Timer1	brak	+	+
Timer2	brak	+	+

Jeżeli zegar jest obecny w danym modelu mikrokontrolera z rodziny Mid-range, to jego działanie musi być zgodne z dokumentacją producenta:

PICmicro™ Mid-Range MCU Family Reference Manual, data sheet **DS33023A**
Dostępne: https://ww1.microchip.com/downloads/en/devicedoc/33023a.pdf

8.10.1. Timer0



Timer0 posiada 8-bitowy licznik dostępny jako TMR0, który jest automatycznie zwiększany o 1. Przepełnienie tego licznika, tzn. przejście 0xFF → 0x00, powoduje ustawienie flagi przerwań T0IF (alternatywna nazwa TMR0IF).
Sygnał taktujący może pochodzić z zewnątrz (TOCS = 1) albo z tego samego zegara (TOCS = 0), który taktuje wykonywanie kolejnych instrukcji (1 instrukcja = 4 cykle).
Sygnał taktujący inkrementację TMR0 opcjonalnie może być dzielony od 1:2 do 1:256 w preskalerze - tylko podziały 1:(2^*n*), gdzie *n* jest całkowite.
Bity kontrolujące Timer0 zebrano w rejestrze OPTION_REG:

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
nRBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

Timer0 – c.d.

Przykład procedury wprowadzqającej opóźnienie o daną liczbę cykli zegara:

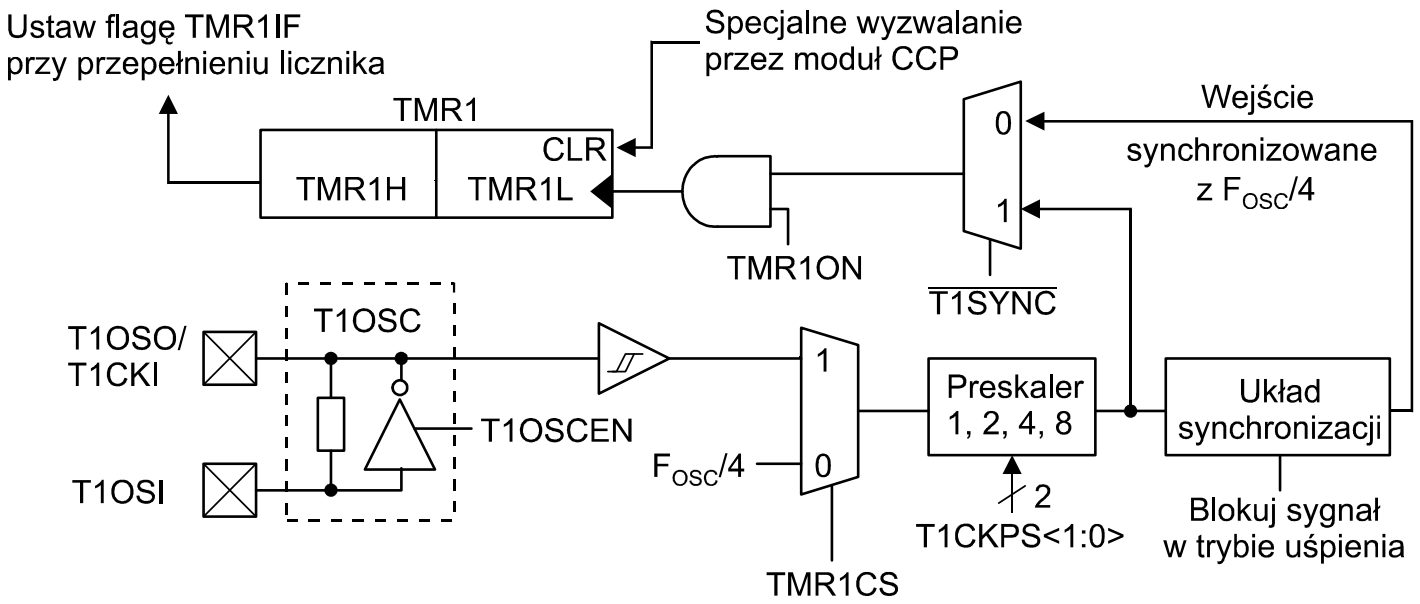
```
#include<htc.h>

void Delay_Tmr0(unsigned short cykle) {
  // 1 cykl = 4 okresy zegara
  if (cykle > 0) {
    TMR0 = 0; // zeruje licznik zegara oraz prescaler
    TMR0IF = 0; // zeruj flagę zgłoszenie przerwania zegara 0
    if (cykle <= 256)
      OPTION_REG = 0B00001000;  // wyłącz preskaler
    else {
      // potrzebny prekskaler od 1:2 do 1:256
      unsigned char PodzialPreskaler = 0;
      while(cykle > 256) {
        cykle >>= 1;
        PodzialPreskaler++;
      }
      OPTION_REG = PodzialPreskaler - 1;
    }
    TMR0 = 256 - cykle;
    // zaczekaj na zgłoszenie przerwania zegara
    while (TMR0IF == 0);
  }
}
```

Licznika preskalera nie można wyzerować niezależnie, tylko razem z zapisem do TMR0

Przy okazji wyzerowanie bitu nRBPU, co powoduje załączenie rezystorów podciągających wejścia w porcie B do napięcia zasilania.

PS<2:0>	podział
000	1:2
001	1:4
…	…
110	1:128
111	1:256

8.10.2. Timer1



Timer1 posiada 16-bitowy licznik dostępny jako para bajtów TMR1H:TMR1L, który jest automatycznie zwiększany o 1. Przepełnienie licznika, tzn. przejście 0xFFFF → 0x0000, powoduje ustawienie flagi przerwań TMR1IF.

Sygnał taktujący może pochodzić z zewnątrz (TMR1CS = 1) albo z tego samego zegara (TMR1CS = 0), który taktuje wykonywanie kolejnych instrukcji (1 instrukcja = 4 cykle). Sygnał taktujący inkrementację przechodzi przez preskaler o wybranym podziale 1:1, 1:2, 1:4 albo 1:8. Bity kontrolujące Timer1 zebrano w rejestrze T1CON:

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
–	–	T1CKPS1	T1CKPS0	T1OSCEN	nT1SYNC	TMR1CS	TMR1ON
bit 7							bit 0

Timer1 – c.d.

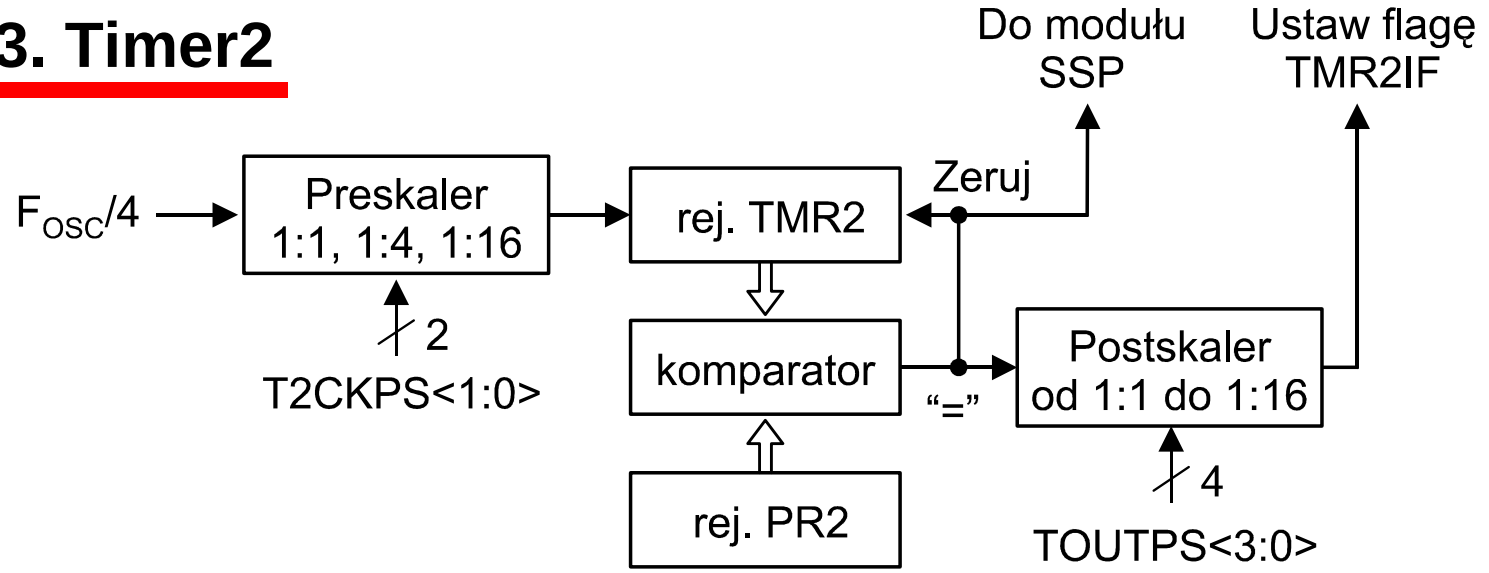
Przykład procedury wprowadzqającej opóźnienie o daną liczbę cykli zegara:

```
#include<htc.h>

void Delay_Tmr1(unsigned long short cykle) {
  // 1 cykl = 4 okresy zegara; najdłuższy odsługiwany czas: 0x80000 = 524288 cykli
  if (cykle > 0) {
    if (cykle > 0x80000) cykle = 0x80000;
    T1CON = 0;  // wyłącz zegar 0
    TMR1IF = 0; // zeruj flagę zgłoszenie przerwania zegara 0
    unsigned char PodzialPreskaler = 0;
    while(cykle > 0x10000) {
      cykle >>= 1;
      PodzialPreskaler++;
    }
    TMR1 = (unsigned int)(0x10000 - cykle);

    T1CONbits.T1CKPS = PodzialPreskaler;
    TMR1ON = 1;  // bit 0 z rej. T1CON - włącz zegar 1
    // zaczekaj na zgłoszenie przerwania zegara
    while (TMR1IF == 0);
  }
}
```

TMR1 jest typu unsigned int (2 bajty).

T1CKPS - pole 2-bitowe

8.10.3. Timer2



Timer2 posiada 8-bitowy licznik dostępny jako TMR2, który jest automatycznie zwiększany o 1. Licznik ten jest zerowany po osiągnięciu wartości z rejestru PR2 i generowany jest sygnał do postskalera o podziale 1:1, 1:2, 1:3, … 1:15, 1:16. Sygnał taktujący może pochodzić tylko z zegara taktującego wykonywanie kolejnych instrukcji (1 instrukcja = 4 cykle). Sygnał taktujący inkrementację TMR2 jest dzielony przez 1:1, 1:4 albo 1:16 w preskalerze.

Bity kontrolujące Timer2 zebrano w rejestrze T1CON :

U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
–	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0
bit 7							bit 0

8.10.4. Generowanie dźwięku z zestawie ZL4PIC

W zestawie ZL4PIC głośniczek piezoelektryczny można dołączyć przez zworę JP4 tylko do wyjścia RA2. Ponieważ żaden zegar nie może być wyprowadzony na to wyjście, należy przełączać stan RA2 w procedurze odsługi przerwania wybranego zegara:

```
#include<htc.h>

void interrupt IntHandler(void) {
  if (TMR?IE && TM?2IF) {
    // odmierzanie okresu przełączeń głośnika
    TMR?IF = 0;
    PORTA ^= 0b00000100;
  }
}// IntHandler

void main(void) {
   ADCON0 = 0;              // wyłącz przetwornik A/D
   ADCON1 = 0x06;           // ustaw linie RA0...RA5 jako cyfrowe
   TRISA = 0b11111011;      // ustaw linię RA2 jako wyjście, reszta wejścia
   PEIE   = 1;              // odblokuj przerwania układów peryferyjnych
   ei();                    // odblokuj wszystkie przerwania

  … // skonfiguruj Timer0, 1 albo 2 do generowania przerwań z odpowiednim okresem
  … // użyj innego zegara do odmierzenia czasu trwania dźwięku
  … // wyłącz przerwania i zegary
}
```

Z zależności od numeru zegara:
TMR0IE && TMR0IF
albo
TMR1IE && TMR2IF
albo
TMR2IE && TMR2IF

8.11. Literatura

[1] „Michrochip HI-TECH C® for PIC10/12/16 User's Guide", data sheet DS51865B, ©2010 Microchip Technology Inc.
[2] Instrukcja do ćwiczenia E59 w lab. techniki cyfrowej „Programowanie mikrokontrolerów PIC16 i urządzeń peryferyjnych w języku C".
[3] B.W. Kernighan, D.M. Ritchie, „Język ANSI C. Programowanie", Wydanie II, Helion, Gliwice 2010.
[4] T. Jabłoński, „Mikrokontrolery PIC16F8x w praktyce", Wydawnictwo BTC, Warszawa 2002.
[5] T. Jabłoński, K. Pławsiuk, „Programowanie mikrokontrolerów PIC w języku C", Wydawnictwo BTC, Warszawa 2005.
[6] S. Pietraszek, Mikroprocesory jednoukładowe PIC, Helion, Gliwice 2002,
[7] PICmicro™ Mid-Range MCU Family Reference Manual, data sheet DS33023A.

Pozycje [1] i [2] w plikach PDF dostępne na stronie:
https://fizyka.p.lodz.pl/pl/dla-studentow/tc/

Pozycja [1] jest dostępna także jako część instalacji pakietu HI TECH Software. Instalator tworzy w Menu Windows grupę „HI-TECH Software" zawierającą pozycję [1] pod nazwą „User manual".