Problemy C - Ustawianie i zerowanie bitów

Autor: Dondu

Ustawianie i zerowanie bitów w rejestrach czy portach mikrokontrolerów, często powoduje frustrację wśród początkujących programistów C i nie tylko. Problemem z reguły są:

1. braki w wiedzy na temat arytmetyki logicznej i składni języka C
2. nie uwzględnianie początkowych wartości rejestrów i portów
3. zapominanie o tym, że ustawiło się lub wyzerowało jakiś bit rejestru

Skorzystaj z: Kurs języka C z przykładami i kompilatorem (online)

1. Braki w wiedzy na temat arytmetyki logicznej i składni języka C

1.1 Operacje na bitach

Nie opisuję tutaj wszelkich zasad i przypadków języka C, a jedynie te, w których początkujący popełniają najwięcej błędów. Jednak na początku przypomnę, że:

- mnożenie (**AND**) reprezentuje znak: **&**
- dodawanie (**OR**) reprezentuje znak: **|**
- dodawanie modulo 2 (**XOR**) znak: **^**
- negacja (**NOT**) znak: **~**

Dodatkowo można używać działań na bitach w połączeniu z operatorem przypisywania: **=**
czyli odpowiednio **&=**, **|=**, **^=**, **~=** .

Szczegóły znajdziesz tutaj: **Kurs C - Operatory bitowe**

Także tylko dla przypomnienia zasady wykonywania działań, które określił **George Boole - Algebra Boole'a** i wyglądają tak:

a	b	a&b
0	0	0
0	1	0
1	0	0
1	1	1

a	b	a|b
0	0	0
0	1	1
1	0	1
1	1	1

a	b	a^b
0	0	0
0	1	1
1	0	1
1	1	0

a	~a
0	1
1	0

Więcej na ten temat opisane w przystępny sposób, możesz przeczytać tutaj: Algebra Boole'a

1.2 _BV(x) czy (1<<x) ?

Aktualnie preferowane jest operowanie bitami nie za pomocą makra **_BV()**, ale z wykorzystaniem rozkazu przesuwania bitów: **<<**
Dlaczego? Ponieważ:

- kod jest czytelny dla pomagających Ci osób
- ułatwia przenoszenie kodu bez konieczności przenoszenia definicja makr

view plainprint?Zwiększ szerokość kodu

```
1.  // nie pisz tak (choć to poprawne)
2.   PORTA |= _BV(PA3);
3.   ADCSRA |= _BV(ADEN) | _BV(ADSC);
4.
5.
6.   // pisz tak
7.   PORTA |= (1<<PA3);
8.   ADCSRA |= (1<<ADEN) | (1<<ADSC);
```

1.3 BŁĄD: Używanie ! zamiast ~

Początkujący często mylą znaczenie ! oraz ~. Stosowanie ! (negacji używanej w warunkach) do negowania bitów jest nieprawidłowe i nie powoduje negowania bitów. Do tego służy znak: ~

view plainprint?Zwiększ szerokość kodu

```
1.  // źle
2.   PORTA = !(1<<PA3);
3.   PORTA = !0x04;
4.
5.   // dobrze
6.   PORTA = ~(1<<PA3);
7.   PORTA = ~0x04;
```

1.4 BŁĄD: Używanie podwójnych && lub || oraz == zamiast pojedynczych i na odwrót

Częste pomyłki zdarzają się także z powodu stosowania podwójnych **&&** i **||** zamiast pojedynczych i na odwrót. **Pamiętaj, że podwójne służą do operacji logicznych, a pojedyncze bitowych.**

view plainprint?Zwiększ szerokość kodu

```
1.  // Sprawdź czy na pinie PA0 jest wysoki stan (jedynka logiczn
    a)
2.
3.   // źle
4.   if(PINA && (1<<PA0)){
5.      //tak jest jedynka
6.   }
7.
8.   // dobrze
9.   if(PINA & (1<<PA0)){
10.     //tak jest jedynka
11.  }
```

Przy sprawdzaniu równości najczęstszym błędem jest stosowanie pojedynczego znaku równości:

view plainprint?Zwiększ szerokość kodu

```
1.  // Sprawdź czy zmienna jest równa 5
2.
3.   // źle
4.   if(zmienna = 5){
5.      //tak
6.   }
7.
8.   // dobrze
9.   if(zmienna == 5){
10.     //tak
11.  }
```

1.5 BŁĄD: Jednoczesne ustawianie i zerowanie bitów za pomocą |= oraz (0<<x)

Pomyłki także dotyczą przypadków gdy w jednej linijce kodu nowicjusz chce jednocześnie wyzerować i ustawiać bity używając operatora **|=**

view plainprint?Zwiększ szerokość kodu

```
1.  // źle !!!
2.   PORTA |= (1<<PA3) | (0<<PA2);    //
    ustaw bit PA3 i jednocześnie zeruj PA2 (źle!)
3.
4.   // dobrze
5.   PORTA |= (1<<PA3);    //ustaw bit PA3
6.   PORTA &= ~(1<<PA2);    //zeruj bit PA2
```

1.6 BŁĄD: Brak wiedzy o pułapkach AVR-ów

Problem dotyczy przypadku, gdy chcemy wyzerować bit, który jest bitem nietypowym, zerowanym poprzez wpisanie jedynki logicznej, a nie zera (!). Szczegóły opisałem tutaj: AVR: Czyhające pułapki

2. BŁĄD: Nie uwzględnianie wartościach początkowych rejestrów i portów

W większości przypadków mikrokontroler, któremu włączono zasilanie lub wykonano sprzętowy reset, przyjmuje w rejestrach wartości 0 na większości bitów rejestrów i portów:

TWBR

Bit	7	6	5	4	3	2	1	0
	TWBR7	TWBR6	TWBR5	TWBR4	TWBR3	TWBR2	TWBR1	TWBR0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Jednakże są wyjątki!!!

SPDR

Bit	7	6	5	4	3	2	1	0
	MSB							LSB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	X	X	X	X	X	X	X	X
	X – Undefined							

UCSRC

Bit	7	6	5	4	3	2	1	0
	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	1	0	0	0	0	1	1	0

Dlatego bardzo istotnym jest sprawdzanie datasheet pod tym kątem i odpowiednie pisanie kodu, by nie powstawały sytuacje, w których spodziewałeś się, że nie musisz kasować danego bitu, a okazało się, że on jest ustawiony domyślnie.

ppawel12
Problem już rozwiązałem. Mój błąd to rutyna :-)
... przypomniałem sobie, że w nocie katalogowej jest umieszczone czasem R/W(1/1) lub R/W(0/0). Po zmianie rejestru ... TMR0 zaczął zliczać impulsy :-)

Podobny przypadek:

figa_miga
No tak, IRCF 2:0 są już ustawione po resecie...ale pomroczność. :-)

3. BŁĄD: Zapominanie o tym, że ustawiło się lub wyzerowało jakiś bit rejestru

To podobny problem jak opisany w pkt. 2. W czasie działania programu czasami trzeba przestawiać bity w wybranym rejestrze. Początkujący zapominają, że wcześniej w tym rejestrze ustawiali jakieś bity, ustawiają lub zerują tylko niektóre, co w konsekwencji prowadzi do nieprawidłowego działania programu.

TCCR0

Bit	7	6	5	4	3	2	1	0
	–	–	–	–	–	CS02	CS01	CS00
Read/Write	R	R	R	R	R	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

- **Bit 2:0 – CS02:0: Clock Select**

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/(No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

Rys. Atmega8 - konfigurowanie preskalera Timer0 - rejestr TCCR0

Pokażę to na przykładzie ustawiania preskalera Timer0 (Atmega8)

Przykład: Błędny kod

```
view plainprint?Zwiększ szerokość kodu
1.  TCCR0 |= (1<<CS00);    // ustaw brak preskalera
2.
3.   // program coś realizujący
4.   // ....
5.
6.   TCCR0 |= (1<<CS01);    // ustaw preskaler na 8
7.
8.   // dalsza część prograu
9.   // ....
```

Błąd polega na tym, że po włączeniu zasilania czy resecie linia 01 prawidłowo ustawi brak preskalera, ale linia 06 zadziała źle ponieważ do wcześniej ustawionego bitu CS00 doda bit CS01, co w konsekwencji ustawi preskaler na 64 zamiast na spodziewane 8 (patrz tabelka powyżej).

Przykład: Poprawny kod

```
view plainprint?Zwiększ szerokość kodu
1.  // ustaw brak preskalera
2.   TCCR0 |= (1<<CS00);    // ustaw bit CS00
3.   TCCR0 &= ~((1<<CS02) |
     (1<<CS01));    // zeruj bity CS02 i CS01
4.
5.   // program coś realizujący
6.   //....
7.
8.   // ustaw  preskaler na 8
9.   TCCR0 |= (1<<CS01);    // ustaw bit CS01
10.  TCCR0 &= ~((1<<CS02) |
     (1<<CS00));    // zeruj bity CS02 i CS00
11.
12.  // dalsza część prograu
13.  // ....
```

Teraz kod zadziała poprawnie ponieważ zawsze ustawiane są odpowiednio wszystkie 3 bity preskalera.

Rada TYLKO dla początkujących:

- zawsze konfiguruj wszystkie niezbędne rejestry i bity do ustawienia wybranego przez Ciebie timera, licznika i innych wewnętrznych peryferii

4. BŁĄD: Wielokrotne ustawianie całego rejestru

Aby kod był bardziej czytelny, programiści często dzielą ustawianie bitów w jednym rejestrze, na kilka linii kodu. Jest to akceptowalny sposób, ale jest w nim bardzo łatwo popełnić tzw. "czeski błąd", który trudno wykryć, jak na przykład w tym temacie:

zumek
70 postów i nikt nie zwrócił na to uwagi ?

Czego nie zauważyliśmy doradzając autorowi tematu?
Autor chciał ustawić w rejestrze TCCR0, bity WGM01 oraz CS02, a pozostałe wyzerować. Niestety zrobił tak:

view plainprint?Zwiększ szerokość kodu

```
1.  TCCR0 = (1<<WGM01);       //
    ustawia bit WGM01 i zeruje pozostałe
2.  TCCR0 = (1<<CS02);        //ustawia bit CS02 i zeruje pozostałe
```

Błąd polega na tym, że w dwóch kolejnych liniach kodu, następują sprzeczne ustawienia bitów tego samego rejestru. Najpierw ustawiany jest bit WGM01, a pozostałe bity są zerowane. W drugiej linii wszystkie bity są zerowane (w tym także WGM01), a ustawiany tylko bit CS02.

W rezultacie tylko bit CS02 został ustawiony poprawnie.

Co zrobić by było prawidłowo?
Należy użyć operatora **|=** w drugim i każdej następnej operacji na tym samym rejestrze, czyli tak:

view plainprint?Zwiększ szerokość kodu

```
1.  TCCR0 = (1<<WGM01);       //
    ustawia bit WGM01 i zeruje pozostałe
2.  TCCR0 |= (1<<CS02);        //dodaje bit CS02
```

W rezultacie w rejestrze TCCR0 ustawione są tylko bity WGM01 i CS02, a tak właśnie chciał autor tematu.

nsmarcin
Hehe już wiem Teraz jest już ok, wielkie dzięki na zwrócenie uwagi na tak banalny błąd, a patrzyłem na to tysiąc razy.

Ja także patrzyłem i nie widziałem - wstyd! :-)

Można także od razu ustawiać wybrane bity.
W ten sposób unikniesz "czeskich błędów" nieznacznie pogarszając czytelność kodu (kwestia przyzwyczajenia).

view plainprint?Zwiększ szerokość kodu

```
1.  TCCR0 = (1<<WGM01) | (1<<CS02);     //
    ustawia bity WGM01 i CS02 i zeruje pozostałe
```

Sam się parę razy złapałem na tym błędzie, dlatego przeważnie stosuję definiowanie rejestru w jednej linii.
Ale są wyjątki opisane w punkcie 1.5 na tej stronie!