Osobliwości kompilatora AVR-GCC i mikrokontrolerów AVR (2)

Kompilator AVR GCC jest chętnie stosowany do kompilowania programów dla mikrokontrolerów AVR. Jak każdy kompilator ma swoje wady i zalety. Specyfika kompilatorów może być istotna, gdy program ma działać szybko lub zajmować mało miejsca w pamięci. Bałagan w definicjach rejestrów czy ich funkcjonalność zmieniana przez producenta w niektórych typach procesorów nie ułatwia pisania programów.

Zacznę od banalnej sprawy, ale może być to istotne przy kompilowaniu programów dla ATTiny. Praktycznie zawsze funkcja *main* jest pętlą nieskończoną i dlatego warto ją zadeklarować w postaci *void main(void) __attribute__((noreturn));*. Dzięki atrybutowi *noreturn* oszczędzamy pamięć Flash na rozkazy *push* i *pop*. W moim programie były to 32 bajty. Jeśli mikrokontroler ma 1 kB pamięci, to 32 B stanowią ponad 3% pojemności pamięci. Nie jest to wiele, ale może nas uchronić przed wymianą „małego” mikrokontrolera na droższy, „większy” model. Pewnym mankamentem zadeklarowania *main* jako *void* jest ostrzeżenie, że *main* nie zwraca *int*. Jeśli natomiast zadeklarujemy ją jako *int* (zajętość Flash nie ulegnie zmianie), to ostrzeżenie o braku *return*. Gdy damy *return*, to ostrzeżenie, że funkcja zwraca wartość, a jest zadeklarowana jako noreturn.

Odliczenie do zera

Liczyć od 0 do jakiejś wartości czy od wartości do 0? W assemblerze sprawa jest oczywista – odliczać do zera, a w C? Sprawdźmy, od 0 do 8:

```
for(x=0; x<8; x++)
{
    PORTE ^= 1;
      8c6: 90 e0        ldi   r25,  0x00; 0
      8c8: 21 e0        ldi   r18,  0x01; 1
      8ca: 8e b1        in    r24,  0x0e; 14
      8cc: 82 27        eor   r24,  r18
      8ce: 8e b9        out   0x0e, r24 ; 14
      8d0: 9f 5f        subi  r25,  0xFF; 255
      8d2: 98 30        cpi   r25,  0x08; 8
      8d4: d1 f7        brne  .-12      ; 0x8ca
<IrqWdg+0x8>
```

A teraz od 8 do 0:

```
for(x=8; x!=0; x--)
{
    PORTE ^= 1;
      8c6: 98 e0        ldi   r25,  0x08; 8
      8c8: 21 e0        ldi   r18,  0x01; 1
      8ca: 8e b1        in    r24,  0x0e; 14
      8cc: 82 27        eor   r24,  r18
      8ce: 8e b9        out   0x0e, r24; 14
      8d0: 91 50        subi  r25,  0x01; 1
      8d2: d9 f7        brne  .-10      ; 0x8ca
<IrqWdg+0x8>
```

Różnica niewielka – 8 rozkazów zamiast 7 („nadmiarowy” rozkaz oznaczono na czerwono) to niewiele, ale pętla w wersji „2” ma o jeden rozkaz mniej, przez co wykona się szybciej, co może mieć duże znaczenie, gdy jest wykonywana w krytycznych czasowo przerwaniach. Na zielono zaznaczono „dziwny” rozkaz odejmij 255 (0xFF), a miało być zwiększanie wartości. Kompilator często stosuje takie sztuczki, bo w danej sytuacji kod jest krótszy, szybszy, a wynik ten sam.

Pętle *for, while, do-while*

Która pętla jest szybsza, *for*, *while*, *do-while*? Odpowiedź poniżej:

```
for( x=0; x<8; x++)
{
      8c4: 90 e0        ldi   r25, 0x00
      8c6: 21 e0        ldi   r18, 0x01
      8c8: 8e b1        in    r24, 0x0e
      8ca: 82 27        eor   r24, r18
      8cc: 8e b9        out   0x0e, r24
      8ce: 9f 5f        subi  r25, 0xFF
      8d0: 98 30        cpi   r25, 0x08
      8d2: d1 f7        brne    .-12      ; 0x8c8
<IrqWdg+0x6>

x = 0;
while( x++<8 )
{
      8c4: 90 e0        ldi   r25, 0x00
      8c6: 21 e0        ldi   r18, 0x01
      8c8: 03 c0        rjmp .+6        ; 0x8d0
<IrqWdg+0xe>
      8ca: 8e b1        in    r24, 0x0e
      8cc: 82 27        eor   r24, r18
      8ce: 8e b9        out   0x0e,r24
      8d0: 9f 5f        subi  r25, 0xFF
      8d2: 99 30        cpi   r25, 0x09
      8d4: d1 f7        brne .-12       ; 0x8ca
<IrqWdg+0x8>

x=0;
do
{
      8c4: 90 e0        ldi   r25, 0x00
      8c6: 21 e0        ldi   r18, 0x01
      8c8: 8e b1        in    r24, 0x0e
      8ca: 82 27        eor   r24, r18
      8cc: 8e b9        out   0x0e, r24
      8ce: 9f 5f        subi  r25, 0xFF
      8d0: 99 30        cpi   r25, 0x09
      8d2: d1 f7        brne .-12       ; 0x8c8
<IrqWdg+0x6>
```

W tym przypadku pętle ***for*** i ***do while*** wygenerowały kod o tej samej długości. Pętla ***do while*** jest przydatna, gdy chcemy, aby wykonała się co najmniej raz, natomiast ***while*** może nie wykonać się nigdy (warunek jest sprawdzany przed realizacją pętli). Różnica pomiędzy ***while*** a ***do while*** polega na wykonaniu na początku skoku (kolor czerwony) do procedury sprawdzania warunku. Operacje na porcie znajdują się na żółtym tle.

Pętla nieskończona

W programach dla mikrokontrolerów bardzo często używamy pętli nieskończonych, aby CPU kręcił się w pętli również po zakończeniu realizacji zadania, ponieważ zwykle nie mamy do dyspozycji systemu operacyjnego, który zadba o stabilność systemu po zakończeniu pracy aplikacji.

Zwykle są stosowane trzy rodzaje pętli nieskończonych – *for(;;)*, *while (1)* oraz *do…while(0)*. To, która pętla jest używana, zwykle zależy po prostu od stylu programowania. A która pętla nieskończona jest wykonywana szybciej? Przyjrzyjmy się ich rozwinięciom w asemblerze.

```
for( ;; )
{
    PORTE ^= 1;
     8c4: 91 e0        ldi   r25, 0x01 ; 1
     8c6: 8e b1        in    r24, 0x0e ; 14
     8c8: 89 27        eor   r24, r25
     8ca: 8e b9        out   0x0e, r24 ; 14
     8cc: fc cf        rjmp  .-8         ; 0x8c6
<IrqWdg+0x4>
while( true )
{
    PORTE ^= 1;
     8c4: 91 e0        ldi   r25, 0x01 ; 1
     8c6: 8e b1        in    r24, 0x0e ; 14
     8c8: 89 27        eor   r24, r25
     8ca: 8e b9        out   0x0e,r24 ; 14
     8cc: fc cf        rjmp  .-8         ; 0x8c6
<IrqWdg+0x4>
do
{
     PORTA ^= 1;
     8c4: 91 e0        ldi   r25, 0x01 ; 1
     8c6: 82 b1        in    r24, 0x02 ; 2
     8c8: 89 27        eor   r24, r25
     8ca: 82 b9        out   0x02, r24 ; 2
     8cc: fc cf        rjmp  .-8       ; 0x8c6
<IrqWdg+0x4>
```

Jak łatwo zauważyć, we wszystkich trzech sytuacjach kod wygenerowany przez kompilator wygląda tak samo.

Przesunięcia

Jaki program w języku asemblera wygeneruje kompilator GCC na podstawie następującej sekwencji poleceń?

```
Static nsigned char w1, w2, w3, w4;
long w=0x11223344;
w1 = w;
w2 = w >> 8;
w3 = w >> 16;
w4 = w >> 24;
```

Czy zostaną użyte rozkazy przesunięć, które wszak znajdują się na liście poleceń realizowanych przez CPU mikrokontrolera AVR? Przekonajmy się.

```
w1 = w;
     8c6: 84 e4          ldi   r24, 0x44 ; 68
     8c8: 80 93 1f 03    sts   0x031F, r24
w2 = w >> 8;
     8cc: 83 e3          ldi   r24, 0x33 ; 51
     8ce: 80 93 36 03    sts   0x0336, r24
w3 = w >> 16;
     8d2: 82 e2          ldi   r24, 0x22 ; 34
     8d4: 80 93 1d 03    sts   0x031D, r24
w4 = w >> 24;
     8d8: 81 e1          ldi   r24, 0x11 ; 17
     8da: 80 93 38 03    sts   0x0338, r24
```

Generowanie przebiegu prostokątnego

Generując przebieg prostokątny na wyjściu mikrokontrolera, możemy skorzystać między innymi z następujących sekwencji poleceń:

```
PORTx ^= _BV(Px);
PORTx ^= _BV(Px);
```

lub

```
PORTx |= _BV(Px);
PORTx &= ~_BV(Px);
```

Który z nich pozwoli na wygenerowanie przebiegu z większą częstotliwością? Przyjrzyjmy się wynikowi pracy kompilatora GCC w asemblerze.

```
while( true )
{
    PORTA ^= _BV(PA2);
    PORTA ^= _BV(PA2);
}
     8c6: 91 e0        ldi   r25, 0x01 ; 1
     8c8: 82 b1        in    r24, 0x02 ; 2
     8ca: 89 27        eor   r24, r25
     8cc: 82 b9        out   0x02, r24 ; 2
     8ce: fc cf        rjmp  .-8       ; 0x8c8
<poligon+0x2>
while( true )
{
    PORTA |= _BV(PA2);
    PORTA &= ~_BV(PA2);
}
     8c6: 12 9a        sbi   0x02, 2 ; 2
     8c8: 12 98        cbi   0x02, 2 ; 2
     8ca: fd cf        rjmp  .-6     ; 0x8c6
<poligon>
```

A tak z ciekawości, jak skompiluje się poniższy program?

```
PORTA = 0;
while( false )
{
    PORTG |= _BV(PA2);
    PORTG &= ~_BV(PA2);
}
PORTA = 255;
```

Oto wynik pracy kompilatora w języku asembler.

```
PORTA = 0;
     8c6: 12 b8        out   0x02, r1 ; 2
```

A oto kolejny przykład:

```
{
    //PORTA ^=1;
    PORTG |= _BV(PA2);
    PORTG &= ~_BV(PA2);
}
PORTA = 255;
     8c8: 8f ef        ldi   r24, 0xFF ; 255
     8ca: 82 b9        out   0x02, r24 ; 2
}
```

Czegoś jakby brakuje – fragmentu zaznaczonego na czerwono. Dlaczego? Ponieważ warunek nigdy nie będzie wykonany. Ponadto, może zdziwić wynik kompilacji *PORTA = 0;* do postaci *out 0x02, r1*, bez wcześniejszego załadowania R1 wartością „0”. Wynika to z faktu, że w R1 jest przechowywana wartość 0 (w sekcji *.init2* rejestr ten jest zerowany rozkazem *eor r1,r1)*. Jeśli funkcja zmienia stan tego rejestru (np. w wyniku wykonania rozkazu mul), musi go wyzerować najpóźniej przed wyjściem z funkcji. Z tego powodu (możliwość zmiany zawartości *r1* przez funkcję) w przerwaniu ten rejestr należy zachować na stosie i wyzerować, bo nie ma gwarancji, że będzie miał wartość 0. Przy wyjściu z przerwania *r1* należy przywrócić.

Sławomir Skrzyński, EP