Osobliwości kompilatora AVR-GCC i mikrokontrolerów AVR (3)

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.

Oczywiste jest, że im krótszy program, tym szybciej się wykonuje. Okazuje się jednak, że jeśli pominiemy pewne istotne mechanizmy zaimplementowane przez projektantów mikroprocesora, to niekoniecznie musi tak być, zwłaszcza w sytuacji, w której mikroprocesor sięga do danych zapisanych w pamięci zewnętrznej. W czasach świetności Commodore 64 (6502 taktowany przebiegiem o częstotliwości 1 MHz, o mocy obliczeniowej porównywalnej do rdzenia AVR przy 4 MHz), aby przesunąć ekran (właściwie połowę) o jeden znak w 10 ms (czas generowania przerwania pionowego oraz wyświetlania części ekranu, na której nie były przeprowadzane operacje przesuwania), trzeba było użyć sekwencji rozkazów „LDA xxxx – STA yyyy" i tak 500 razy, przy czym w każdym rozkazie „xxxx" i „yyyy" były inne. Standardowa pętla:

```
LDA (aa),X
STA (aa),X
INC X
BNE -7
```

była zbyt wolna. Wpisanie kilkuset sekwencji LDA – STA bez pomyłki było czasochłonne (asembler nie obsługiwał makr). Problem rozwiązałem, pisząc program, który w pamięci RAM komputera wygenerował kilkaset rozkazów LDA, STA i zakończył kodem RET. Kod był szybki, ale zajmował 3001 bajtów, a nie około 10, jak byłoby w wypadku wykonania pętli.

Przyspieszenie obsługi funkcji w przerwaniu

W AVR dynamiczne generowanie i modyfikacja kodu nie są możliwe. Można to zrobić w procesorach z pamięcią programu w postaci RAM, ale mimo wszystko nie polecam tej operacji, ponieważ współczesne procesory są wyposażone w pamięć cache, którą trzeba czyścić po każdej modyfikacji kodu, a to nie wpływa dobrze na wydajność.

Jak więc zwiększyć prędkość programu poza oczywistą optymalizacją? Nasuwają się co najmniej trzy metody:

1. Funkcje inline.
2. Rezerwowanie rejestrów (bardzo niebezpiecznie w obsłudze przerwania).
3. Wstawki asemblerowe.

Optymalizacja procedur w IRQ jest dość trudna, ale często można poprawić kod wynikowy wygenerowany przez kompilator. Za przykład może posłużyć przerwanie od UART. Nie można go zdefiniować z atrybutem NOBLOCK (INTERRUPT). Typowy program zezwalający na IRQ w przerwaniu:

```
Usart1 _ RxBuf[Usart1 _ PtrRxW++] = UDR1;  //wpisz
znak do bufora i zwiększ długość
sei();
Usart1 _ PtrRxW &= USART1 _ RXBUF-1;      //
Ograniczenie wielości bufora
```

po skompilowaniu będzie wyglądał następująco:

```
//1f 92       push  r1
//0f 92       push  r0
//0f b6       in       r0, 0x3f  ; 63
//0f 92       push  r0
//0b b6       in       r0, 0x3b  ; 59
//0f 92       push  r0
//24          eor   r1, r1
//93          push  r24
//9f 93       push  r25
//ef 93       push  r30
//ff 93       push  r31
//80 91 2f 03 lds   r24, 0x032F
//90 91 ce 00 lds   r25, 0x00CE
//tu mógłby być rozkaz sei
//eb e2       ldi   r30, 0x2B  ; 43
//f5 e0       ldi   r31, 0x05  ; 5
//e8 0f       add   r30, r24
//f1 1d       adc   r31, r1
//90 83       st       Z,  r25
//8f 5f       subi  r24, 0xFF  ; 255
//80 93 2f 03 sts   0x032F, r24
//78 94sei  tu jest rozkaz sei
//80 91 2f 03 lds   r24, 0x032F  // Ograniczenie
wielości bufora
//8f 70       andi  r24, 0x0F  ; 15
//80 93 2f 03 sts   0x032F, r24
//ff 91       pop   r31
//ef 91       pop   r30
//9f 91       pop   r25
//8f 91       pop   r24
//0f 90       pop   r0
//0b be       out   0x3b, r0  ; 59
//0f 90       pop   r0
//0f be       out   0x3f, r0  ; 63
//0f 90       pop   r0
//1f 90       pop   r1
//18 95       reti
```

Widać tam operacje na R1 i R0, które można pominąć, ale trzeba też zmodyfikować rozkaz „ADC r31,r1", ponieważ „r1" może mieć wartość różną od zera. Pon rozkaz występuje za „sts" (kolor niebieski), a mógłby po „lds" (kolor zielony). Zmienia kod źródłowy na:

```
byte x;
    x=UDR1;
    sei();
    Usart1 _ RxBuf[Usart1 _ PtrRxW++] = x;
```

```
    Usart1_PtrRxW &= USART1_RXBUF-1;  //
Ograniczenie wielości bufora
```

Sytuacja nieco się poprawi (pominięto operacje na stosie):

```
90 91 ce 00 lds  r25, 0x00CE  // wpisz znak
do bufora i zwiększ długość
78 94        sei
80 91 2f 03 lds  r24, 0x032F
eb e2        ldi  r30, 0x2B    ; 43
f5 e0        ldi  r31, 0x05    ; 5
e8 0f        add  r30, r24
f1 1d        adc  r31, r1
90 83        st   Z,   r25
8f 5f        subi r24, 0xFF    ; 255
80 93 2f 03 sts  0x032F, r24
80 91 2f 03 lds  r24, 0x032F  // Ograniczenie
wielości bufora
8f 70        andi r24, 0x0F    ; 15
80 93 2f 03 sts  0x032F, r24
```

Rozkaz `sei()` jest umieszczany wcześniej. Można jednak jeszcze lepiej (przerwanie z atrybutem NAKED):

```
  __asm__ volatile (
    „push  r25        \n\t”
    „in    r25, 0x3F  ;SREG na stos  \n\t”
    „push  r25        \n\t”
    „sei   \n\t”
    „lds   r25, 0xCE  ;UDR1  \n\t”
    „push  r24  \n\t”
    „push  r1   \n\t”
    „eor   r1,  r1  ;zerowanie R1  \n\t”
    „push  r30  \n\t”
    „push  r31  \n\t”
);

_ _asm_ _ volatile (  //
Usart1_RxBuf[Usart1_PtrRxW]
    „lds  r24, Usart1_PtrRxW  ;  \n\t”
    „ldi  r30, lo8(Usart1_RxBuf)     ;LO  \n\t”
    „ldi  r31, hi8(Usart1_RxBuf)     ;HI  \n\t”
    „add  r30, r24  ;LO + Usart1_PtrRxW  \n\t”
    „adc  r31, r1  \n\t”
    „st   Z,   r25 \n\t”
);

_ _asm_ _ volatile (  //  Usart1_PtrRxW++
    „lds   r24, Usart1_PtrRxW  \n\t”
    „subi  r24, 0xFF  ; 255  \n\t”
    „sts   Usart1_PtrRxW, r24  \n\t”
);

_ _asm_ _ volatile (// Usart1_PtrRxW &=
USART1_RXBUF-1;
// Ograniczenie wielości bufora (max 255 bajtów)
    „lds   r24, Usart1_PtrRxW  \n\t”
    „andi  r24, 0x0F  ; 15  \n\t”  //todo: zamiast
0x0f dać: USART1_RXBUF-1
    „sts  Usart1_PtrRxW, r24  \n\t”
);

_ _asm_ _ volatile (
    „pop   r31  \n\t”
    „pop   r30  \n\t”
    „pop   r1   \n\t”
    „pop   r24  \n\t”
    „pop   r25  \n\t”
    „out   0x3F,r25  ;SREG ze stosu  \n\t”
```

```
    „pop   r25  \n\t”
    „reti  \n\t”
);
```

Teraz `sei()` jest czwartym rozkazem. Wydawałoby się, że musi być piątym, ale w AVR po rozkazie `sei()` gwarantowane jest wykonanie co najmniej jednego rozkazu.

Opisany tu problem dotyczy także wejścia INT wyzwalanego poziomem. Gdy do takiego wejścia podłączę kilka układów (popularne RTC, PIO na IIC), a mamy krytyczne czasowo przerwania, to wystąpi ten sam kłopot. Rozwiązaniem jest wstawka asemblerowa (jak dla UART), w IRQ sprawdzić, który układ zgłasza przerwanie, skasować je, odblokować przerwania i realizować IRQ od urządzenia.

Inny przypadek optymalizacji: fragment kodu odczytu bajtu podczas emulacji slave 1-Wire w przerwaniach. „Opcja 1” to fragment programu znaleziony w ogólnie dostępnych źródłach, „opcja 2” powstała w wyniku szybkiej modyfikacji, natomiast „opcja 3” – po głębszych przemyśleniach.

```
// opcja 1
    if(WIRE1_RD) byte_rdE|=(1<<bit_indexE); //
wymaga zerowania „byte_rdE” przed odczytem
// opcja 2
    if(WIRE1_RD) byte_rdE|=(1<<bit_indexE);
    else byte_rdE &= ~(1<<bit_indexE);  // nie
wymaga zerowania „byte_rdE”
// opcja 3
        byte_rdE >>= 1; if(WIRE1_RD) byte_
rdE|=0x80; // OK - nie wymaga zerowania „byte_rdE”
```

Efekt kompilacji:

```
// opcja 1
  if( WIRE1_RD ) byte_rdE |= (1<<bit_indexE);
    8c6:  60 9b         sbis  0x0c, 0 ; 12
    8c8:  0e c0         rjmp  .+28          ; 0x8e6
<IrqWdg+0x24>
    8ca:  81 e0         ldi   r24, 0x01     ; 1
    8cc:  90 e0         ldi   r25, 0x00     ; 0
    8ce:  00 90 1d 03   lds   r0, 0x031D
    8d2:  02 c0         rjmp  .+4           ; 0x8d8
<IrqWdg+0x16>
    8d4:  88 0f         add   r24, r24
    8d6:  99 1f         adc   r25, r25
    8d8:  0a 94         dec   r0
    8da:  e2 f7         brpl  .-8           ; 0x8d4
<IrqWdg+0x12>
    8dc:  20 91 28 03   lds   r18, 0x0328
    8e0:  28 2b         or    r18, r24
    8e2:  20 93 28 03   sts   0x0328, r18
// opcja 2
    8e8:  30 91 1d 03   lds     r19, 0x031D
    8ec:  20 91 28 03   lds     r18, 0x0328
  if(WIRE1_RD) byte_rdE|=(1<<bit_indexE); else
byte_rdE &= ~(1<<bit_indexE);
    8f0:  60 9b         sbis  0x0c, 0 ; 12
    8f2:  09 c0         rjmp  .+18          ; 0x906
<IrqWdg+0x44>
    8f4:  81 e0         ldi   r24, 0x01   ; 1
    8f6:  90 e0         ldi   r25, 0x00   ; 0
    8f8:  02 c0         rjmp  .+4         ; 0x8fe
<IrqWdg+0x3c>
    8fa:  88 0f         add   r24, r24
    8fc:  99 1f         adc   r25, r25
    8fe:  3a 95         dec   r19
    900:  e2 f7         brpl  .-8         ; 0x8fa
<IrqWdg+0x38>
    902:  28 2b         or    r18, r24
    904:  09 c0         rjmp  .+18          ; 0x918
<IrqWdg+0x56>
    906:  81 e0         ldi   r24, 0x01   ; 1
    908:  90 e0         ldi   r25, 0x00   ; 0
    90a:  02 c0         rjmp  .+4         ; 0x910
<IrqWdg+0x4e>
    90c:  88 0f         add   r24, r24
    90e:  99 1f         adc   r25, r25
    910:  3a 95         dec   r19
    912:  e2 f7         brpl  .-8         ; 0x90c
<IrqWdg+0x4a>
    914:  80 95         com   r24
    916:  28 23         and   r18, r24
    918:  20 93 28 03   sts   0x0328, r18
// opcja 3
  byte_rdE >>= 1; if( WIRE1_RD ) byte_rdE |=
0x80;
    91e:  80 91 28 03   lds   r24, 0x0328
    922:  86 95         lsr   r24
    924:  80 93 28 03   sts   0x0328, r24
    928:  60 9b         sbis  0x0c, 0 ; 12
    92a:  03 c0         rjmp  .+6         ; 0x932
<IrqWdg+0x70>
    92c:  80 68         ori   r24, 0x80   ; 128
    92e:  80 93 28 03   sts   0x0328, r24
    932:  08 95         ret
```

Jak można zauważyć, opcja 1 (wymagająca wyzerowania bajtu przed odbiorem) – 14 rozkazów. Opcja 2 (nie wymaga zerowania) – 23 rozkazy. Opcja 3 oparta o przesuwanie (nie wymaga zerowania) – 8 rozkazów. Najkrótsza, najszybsza i nie wymaga zerowania bajtu przed odczytem, co jest istotne podczas pracy w trybie overdrive.

Skok do funkcji przez wektor

Ta funkcjonalność przydaje się, jeśli:

- W programie trzeba skoczyć do części programu zależnej od warunku.
- Warunek jest sprawdzany często, a same składniki wyrażenia logicznego zmieniają się rzadko.

Na przykład, w emulatorze 1-Wire będzie to typ emulowanego układu. Warunek jest ustalany raz na kilka minut czy godzin, a sprawdzany tysiące razy na sekundę. Typowa konstrukcja switch...case nie sprawdza się (można zmienić w opcjach kompilacji, ale dotyczy to całego kodu, a nie jego fragmentów). W wyniku kompilacji fragmentu programu:

```
switch(zmienna)
{
    case ( 0x28 ):
        funkcja1;
        break;
    case ( 0x2d ):
        funkcja;2;
        break;
    case ( 0x31 ):
        funkcja3;
        break;
}
```

Zostanie wygenerowany kod:

```
                        ;switch(EmulowanyRD)
    1c7c:  80 91 4d 04  lds   r24, 0x044D
    1c80:  88 32        cpi   r24, 0x28
    1c82:  09 f4        brne  .+2
    1c84:  3a c1        rjmp  .+628
    1c86:  8d 32        cpi   r24, 0x2D
    1c88:  09 f0        breq  .+2
    1c8a:  a5 cf        rjmp  .+203
```

```
    1c8c:   8d 32         cpi    r24, 0x31
    1c8e:   09 f0         breq   .+2
    1c8g:   a5 cf         rjmp   .-182
```

Podobne fragmenty programu zostaną utworzone dla kolejnych warunków. Rozsądniejsze jest zastąpienie `switch…case` (praktycznie ten sam kod wygeneruje `if…else`) skokiem przez wektor. Można to wykonać w następujący sposób:

```
void RejestracjaAdresuFunkcji()
{
    switch( EmulowanyRD )
    {
        case(EMU _ 18B20):
            adrCallEmu = &irq _ Subcommand _ DS18B20;
            break;
        case(EMU _ 2431):
            adrCallEmu = &irq _ Subcommand _ DS2431;
            break;
        case(EMU _ 2502):
            adrCallEmu = &irq _ Subcommand _ DS2502;
            break;
        default:
            adrCallEmu = &irq _ Subcommand _ unknown;
    }
}
```

Wywołanie ma postać

```
(*((void(*)(void))adrCallEmu))();  // CALL do funkcji
```

o adresie „adrCall”.

Zmienna **adrCall** musi być zadeklarowana jako **long** (w procesorach do 128 kB wystarczy **unsigned int**). Podczas jej zmiany należy zablokować przerwania. Wywołanie funkcji po skompilowaniu nie zmieni się bez względu na liczbę warunków i będzie miało postać:

```
    1c70:   e0 91 bd 07    lds   r30, 0x07BD
    1c74:   f0 91 be 07    lds   r31, 0x07BE
    1c78:   09 95          icall
```

Rezerwowanie rejestrów

Rezerwacja rejestrów dla funkcji wykonywanych w IRQ jest o tyle niebezpieczna, że w przypadku takiej rezerwacji, kompilator powinien, ale nie musi, pozostawić taki rejestr na konkretną zmienną. W konsekwencji w kodzie programu może być on zmodyfikowany, wtedy kompilator wykona następującą sekwencję rozkazów:

```
push <zarezerwowany rejestr>
...operacje z jego użyciem
pop <Rx>
```

Jeśli wtedy nastąpi przerwanie, to procedura przerwań, zakładając, że nie był modyfikowany, może „pójść w maliny”.

Zmienne bitowe

Jeśli chcemy operować na bitach (warunki prawda, fałsz), to przydałoby się tu adresowanie bitowe (jak w 80651). Wtedy program zamiast pobierać zmienną, modyfikować ją i zapisać z powrotem, wykona tylko jeden rozkaz. Problemu nie rozwiązuje deklaracja struktury:

```
struct
{
    byte F1 : 1;  // Flaga 1
    byte R2 : 1;  // Flaga 2
} FlagF;
```

Operacje i tak są wykonywane na bajcie, a nie na bicie:

```
FlagF.F1 = 1;
    97a:   80 91 5f 03   lds   r24, 0x035F
    97e:   81 60         ori   r24, 0x01   ; 1
    980:   80 93 5f 03   sts   0x035F, r24
```

Takie „pole bitowe” nie przyśpiesza wykonywania kodu programu, tylko zwiększa jego czytelność. Należy też pamiętać, aby flagi modyfikowane w IRQ nie były w tym samym bajcie, co flagi modyfikowane w programie głównym. Flagi bitowe nie muszą zajmować jednego bitu – mogą to być dwa, trzy lub więcej bitów:

```
  struct
  {
  byte F1 : 3; // Flaga 1
    byte R2 : 6;  // Flaga 2
} FlagF;
FlagF.F1 = 5;
FlagF.F2 = 0x30;

FlagF.F1 = 5;
  92:   80 91 72 00    lds    r24, 0x0072
  96:   88 7f          andi   r24, 0xF8   ; 248
  98:   85 60          ori    r24, 0x05   ; 5
  9a:   80 93 72 00    sts    0x0072, r24

FlagF.F2 = 0x30;
  92:    80 91 72 00      lds    r24, 0x0072
  96:   87 70          andi   r24, 0x07   ; 7
  98:   80 68          ori    r24, 0x80   ; 128
  9a:   80 93 72 00    sts    0x0072, r24
  9e:   80 91 73 00    lds    r24, 0x0073
  a2:   81 60          ori    r24, 0x01   ; 1
  a4:   80 93 73 00    sts    0x0073, r24
```

Przykład pokazuje, że jeżeli flagi wielobitowe znajdą się na granicy bajtu, operacja będzie przeprowadzona na dwóch bajtach – trzeba brać to pod uwagę przy deklaracji tego typu zmiennych.

Niektóre AVR mają rejestry GPIO operujące na bitach. Trzeba być ostrożnym podczas ich używania, ponieważ niektóre (np. GPIO1 i GPIO2 w ATmega1281/2561) nie są tak adresowane (adres większy niż 0x31), choć GPIO0 tak. A gdy procesor nie ma GPIO? Każdy AVR je ma, tylko sobie tego nie uświadamiamy. Funkcję GPIO mogą pełnić:

- nieużywane PIO (rejestry PORT i DDR),
- nieużywane rejestry PORTx portów wejściowych, jeśli kwestia pullup jest nieistotna,
- rejestry nieużywanych modułów funkcjonalnych (peryferyjnych),
- rzadko używany bit TXB8 w UCSRB,
- PORTx i DDRx linii Rx i Tx USARTa, ale nie mogą to być peryferie, które, aby pracowały, wymagają ustawienia bitu rejestrze DDR, np. PWM czy SPI, bo zakłóci to ich pracę.

Nie można stosować nieużywanych flag przerwań, bo choć można je zapisywać i odczytywać, to zapis powoduje automatyczne wyzerowanie.

Sławomir Skrzyński, EP