Osobliwości kompilatora AVR-GCC i mikrokontrolerów AVR (1)

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.

Najwięcej materiałów do artykułu dostarczyło pisanie programu funkcjonującego z wykorzystaniem przerwań, emulującego układy 1-Wire pracujące w trybie *overdrive*. W tym trybie trzeba w ciągu maksymalnie 2 µs od opadającego zbocza sygnału odczytu wystawić transmitowany bit. Czas wykonania jednego rozkazu przy zegarze 16 MHz to około 0,0625 µs. Biorąc pod uwagę fakt, że gdy pisze się aplikację w GCC, przy wejściu w przerwanie operacje na stosie zajmują około 2 µs (na stos jest odkładana zawartość 20 rejestrów), wydaje się niemożliwe obsłużenie tego trybu. A jednak się udało, o czym dalej.

Zakres zmiennych

Wydawałoby się, że wynikiem mnożenia dwóch liczb *unsigned int* będzie *unsigned long*. Czy na pewno? Spróbujmy:

```
unsigned int a=10000, b=10;
unsigned long w;
```

Program działa poprawnie, jeśli wynikiem mnożenia jest liczba typu *unsigned int*. Gdy jest większy, nieoczekiwanie otrzymujemy dziwne rezultaty. Bliższe przyjrzenie się wynikom operacji w debuggerze ujawnia, że starsze bajty zmiennej są zerem. Jak rozwiązać problem? Trzeba wykonać jawną konwersję typu i operację zapisać tak:

```
w = (unsigned long)a * b;
```

Dlaczego tak niewielka zmiana spowodowała poprawne działanie programu? Otóż kompilator wykonuje następujące działania:

```
a * b -> w
```

A więc mnoży zmienną „*a*” przez zmienną „*b*”, zapisując wynik w zmiennej „*a*”, po przepisuje go do zmiennej „*w*”. Gdy zmienna „*a*” była zadeklarowany jako *unsigned int*, kod wynikowy programu w asemblerze wyglądał następująco:

```
„w = a * b;”
1c74:    20 91 02 01      lds     r18, 0x0102
1c78:    30 91 03 01      lds     r19, 0x0103
1c7c:    80 91 00 01      lds     r24, 0x0100
1c80:    90 91 01 01      lds     r25, 0x0101
1c84:    ac 01            movw    r20, r24
1c86:    24 9f            mul     r18, r20
1c88:    c0 01            movw    r24, r0
1c8a:    25 9f            mul     r18, r21
1c8c:    90 0d            add     r25, r0
1c8e:    34 9f            mul     r19, r20
1c90:    90 0d            add     r25, r0
1c92:    11 24            eor     r1, r1
1c94:    a0 e0            ldi     r26, 0x00        ; 0
1c96:    b0 e0            ldi     r27, 0x00        ; 0
1c98:    80 93 05 02      sts     0x0205, r24
1c9c:    90 93 06 02      sts     0x0206, r25
1ca0:    a0 93 07 02      sts     0x0207, r26
1ca4:    b0 93 08 02      sts     0x0208, r27
```

Wyraźnie widać, że najstarsze bajty są wyzerowane. Gdy zmienną rzutujemy na *unsigned int*, wynik pracy kompilatora wygląda następująco:

```
„w = (long)a * b;”
1c74:    60 91 02 01      lds     r22, 0x0102
1c78:    70 91 03 01      lds     r23, 0x0103
1c7c:    80 e0            ldi     r24, 0x00        ; 0
1c7e:    90 e0            ldi     r25, 0x00        ; 0
1c80:    20 91 00 01      lds     r18, 0x0100
1c84:    30 91 01 01      lds     r19, 0x0101
1c88:    40 e0            ldi     r20, 0x00        ; 0
1c8a:    50 e0            ldi     r21, 0x00        ; 0
1c8c:    0e 94 4e 20      call    0x409c    ; 0x409c
< _ _ mulsi3>
1c90:    60 93 05 02      sts     0x0205, r22
1c94:    70 93 06 02      sts     0x0206, r23
1c98:    80 93 07 02      sts     0x0207, r24
1c9c:    90 93 08 02      sts     0x0208, r25
```

Użyta procedury mnożenia ***__mulsi3*** jest dosyć długa, dlatego zainteresowanych zachęcam do obejrzenia wyniku kompilacji na własnym komputerze. Ważne, że procedura ta operuje na 32 bitach.

Przerwanie programowe

Mikrokontrolery AVR nie mają możliwości wywoływania przerwań z aplikacji użytkownika. Pomijam tu asemblerową instrukcję *BREAK*, której działanie nie jest szeroko opisane. Jeśli istnieje potrzeba wygenerowania takiego przerwania i mamy wolne wejście przerwań zewnętrznych, można poradzić sobie w następujący sposób:

Skonfigurować wejście INTx jako wywołujące przerwanie opadającym zboczem sygnału.

Ustawić pin jako **wyjście**.

Aby wywołać przerwanie, wykonać rozkaz *PORTx &= ~_BV(y);*.

W programie obsługi przerwania wykonać *PORTx |= _BV(y);* i obsłużyć przerwanie.

Rozwiązania tego używałem w celu emulowania impulsatora za pomocą UART obsługiwanego z terminalu na komputerze PC.

Przerwanie od WDG

W niektórych mikrokontrolerach AVR układ czasowy watchdog (WDG) może generować przerwanie. Jeśli w obsłudze tego przerwania nie ustawimy flagi WDIE (kasowana automatycznie przez przerwanie od WDG), to kolejne zadziałanie WDG spowoduje restart CPU. Aby funkcjonalność przerwań od WDG zadziałała, nie może być ustawiony bit WDTON w bitach konfiguracyjnych. Funkcjonalność IRQ od WDG włączamy, ustawiając WDIE poleceniem *WDTCSR |= (1<<WDIE);*. Oczywiście, należy ustawić globalne zezwolenie na przerwania instrukcją sei(), w przeciwnym wypadku po dwukrotnym przepełnieniu timera nastąpi reset mikrokontrolera.

Listing 1. Przykładowa procedura obsługi przerwania od WDG

```
/* „czas” - define czau zadziałania WDG, dostępne wartości:
WDTO_30MS, WDTO_60MS, WDTO_120MS, WDTO_250MS, WDTO_500MS, WDTO_1S, WDTO_1S
*funkcja - adres funkcji wywołanej przed wyjściem z przerwania od WDG
W zmiennej globalnej „adrCallWdg” znajduje sie adres z którego nastąpiło przerwanie WDG. Jeśli „funkcja” = 0 to skok nie będzie wykonany
void InitWdgI( byte czas, long *funkcja )
{
byte cSREG;
    cSREG = SREG;
    cli();
    InitWdg( czas );
    #if defined(__AVR_ATmega640__) || defined(__AVR_ATmega1280__) || defined(__AVR_ATmega1281__) || defined(__AVR_ATmega2560__) || defined(__
AVR_ATmega2561__)
        WDTCSR |= (1<<WDIE);  // Aby ustawić WDIE nie może być ustawiony bit WDTON w fuses
        adrUserCallWdg = funkcja;
        adrUserCallWdg <<= 1;
    #else
        #error „Tan procesior nie generuje IRQ od WDG. Uzyj funkcji ‚InitWdg()’.”
    #endif
    SREG = cSREG;
}

// „czas” - define czau zadziałania WDG, dostępne wartości:
void InitWdg( byte czas )
{
byte cSREG;
    cSREG = SREG;
    cli();
    wdt_enable( czas );  // Ustawienie WDG (IRQ muszą byc wyłączone)
    SREG = cSREG;
}

// WDIE jest automatycznie kasowany, dlatego następne zadziałanie WDG wywoła reset
#if defined(__AVR_ATmega164__) || defined(__AVR_ATmega640__) || defined(__AVR_ATmega1280__) || defined(__AVR_ATmega1281__) || defined(__AVR_AT-
mega2560__) || defined(__AVR_ATmega2561__)
#ifdef WDGI_NAKED
    ISR(WDT_vect, ISR_NAKED )
#else
    SIGNAL( WDT_vect )
#endif
{
    byte *ptr, ofs=0, adr[3];
    word stack, cmd;
    #if  defined(__AVR_ATmega2560__) || defined(__AVR_ATmega2561__)
        long vect;  // Ponad 128kB adres jest 3-bajtowy
#else
  word vect;
#endif
#ifndef WDGI_NAKED
    nop();  // Pierwszy rozkaz procedury (kod maszynowy rozkazu = $0000)
#endif
    vect = WDT_vect;  // Adres wektora przerwania od WGD
    vect <<= 1;  // Adres w słowach więc mnożymy przez 2
#ifndef WDGI_NAKED
//xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    for( byte x=0; x<32; x++ )          // Liczymy liczbe rozkazów push
    {
        #if defined(__AVR_ATmega1280__) || defined(__AVR_ATmega1281__) || defined(__AVR_ATmega2560__) || defined(__AVR_ATmega2561__)
            cmd = pgm_read_word_far(vect);// Ponad 64KB „far”
        #else
            cmd = pgm_read_word( vect );
        #endif
        vect +=2;
        if ( !cmd ) break;  // Jeśli kod rozkazu „nop”
        if ( ((cmd&0xFF00)==0x9200) || // Jeśli rozkaz push R0..15
              ((cmd&0xFF00)==0x9300) ) // Jeśli rozkaz push R16..31
        {
            ofs++;
        }
    }
#endif
    stack = SP + ofs;
    stack++;  // Stos wskazuje pierwszy wolny bajt więc zwiększamy o 1
    ptr = stack;
    adr[0] = *ptr++;
    adr[1] = *ptr++;
    adr[2] = *ptr++;
    #if  defined(__AVR_ATmega2560__) || defined(__AVR_ATmega2561__)
        adrCallWdg = ((adr[0] << 8) <<8);         // Adres powrotu dla cpu ponad 128kB
        adrCallWdg |= adr[1] << 8;
        adrCallWdg |= adr[2];
    #else
        adrCallWdg = adr[0] << 8;  // Adres powrotu dla cpu do 128kB
        adrCallWdg |= adr[1];
    #endif
    adrCallWdg *= 2;  //PC wskazuje nr słowa więc adres jest 2 razy większy
//xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    if ( adrUserCallWdg )
    {
        long adrCall = adrUserCallWdg >> 1;
        (*((void(*)(void))adrCall))();  // CALL do funkcji o adresie „adrUserCallWdg”
    }
#ifdef WDGI_NAKED
    while( true );
#endif
}
#endif

//Deklaracje:
#define  DEF_RST_SOFT  0x534f4654
#define  DEF_RST_UNKOWN  (DEF_RST_SOFT ^ 0xaa55a55a)
#define WDGI_NAKED  //Po obsłudze IRQ wykonanie resetu (krótszy kod).
long IdSoftReset NOINIT;
long adrUserCallWdg=0, adrCallWdg=0;
extern void InitWdgI( byte czas, long *funkcja );  // Init IRQ od WDG
extern void InitWdg( byte czas );  // Init WDG
#define SoftReset()  IdSoftReset=DEF_RST_SOFT; IdSoftReset=DEF_RST_UNKOWN; while(true);
```

Listing 2. Procedura konwersji zmiennej long na łańcuch znaków

```
void sPrintfLongDec( char *str, long dec )
{
void sPrintfLongDec( char *str, long dec )
{
    ulong w;
    byte z = TRUE;
    if ( dec & 0x80000000 )  // Jeśli liczba ujemna
    {
        dec = ~dec + 1;
        *str++ = ,-';
    }
// 2147483647 (0x7FFFFFFF)
    w = dec / 1000000000;
    dec %= 1000000000;
    if ( w || !z )
    {
        *str++ = ( w + ,0' );
        z = FALSE;
    }
    w = dec / 100000000;
    dec %= 100000000;
    if ( w || !z ) {
        *str++ = ( w + ,0' );
        z = FALSE;
    }
    w = dec / 10000000;
    dec %= 10000000;
    if ( w || !z ) {
        *str++ = ( w + ,0' );
        z = FALSE;
    }
    w = dec / 1000000;
    dec %= 1000000;
    if ( w || !z ) {
        *str++ = ( w + ,0' );
        z = FALSE;
    }
    w = dec / 100000;
    dec %= 100000;
    if ( w || !z ) {
        *str++ = ( w + ,0' );
        z = FALSE;
    }
    w = dec / 10000;
    dec %= 10000;
    if ( w || !z ) {
        *str++ = ( w + ,0' );
        z = FALSE;
    }
    w = dec / 1000;
    dec %= 1000; f ( w || !z ) {
        *str++ = ( w + ,0' );
        z = FALSE;
    }
    w = dec / 100;
    dec %= 100;
    if ( w || !z ) {
        *str++ = ( w + ,0' );
        z = FALSE;
    }
    w = dec / 10;
    dec %= 10;
    if ( w || !z ) {
        *str++ = ( w + ,0' );
        z = FALSE;
    }
    *str++ = ( dec + ,0' ); *str = 0;
}

void sPrintfWordDec( char *str, word dec )
{
    word w;
    byte z = TRUE;
    w = dec / 10000; dec %= 10000; if ( w )                    { *str++ = ( w + ,0' );
z = FALSE; }
    w = dec /  1000; dec %=  1000; if ( w || !z ) { *str++ = ( w + ,0' ); z = FALSE; }
    w = dec /   100; dec %=   100; if ( w || !z ) { *str++ = ( w + ,0' ); z = FALSE; }
    w = dec /    10; dec %=    10; if ( w || !z ) { *str++ = ( w + ,0' ); z = FALSE; }
    *str++ = ( dec + ,0' ); *str = 0;
}

void PrintLongHex( long hex )
{
    PrintWordHex( ((hex>>8)>>8) );
    PrintWordHex( hex );
}

void PrintWordHex( word hex )
{
    PrintByteHex( hex >> 8 );
    PrintByteHex( hex );
}

void PrintByteHex( byte hex )
{
    static byte hexmem;
    hexmem=hex;
    PrintNibleHex( hex >> 4 );
    PrintNibleHex( hex );
}

void PrintNibleHex(  byte hex )
{
    hex &= 0x0F;
    static byte nibmem;
    nibmem=hex;
    if ( hex <= 9 ) Usart1_Transmit( hex + ,0' );
    else Usart1_Transmit( hex-10 + ,A' );
}
```

W przerwaniu od WDG możemy poznać adres, z którego program skoczył do obsługi IRQ. Każde CPU, skacząc do procedury obsługi IRQ, odkłada na stos adres powrotu, a niektóre także rejestr stanu. Dodatkowo jest zapamiętywany stan flagi „I". Jeśli przerwanie jest zadeklarowane jako INTERRUPT lub ISR z atrybutem NOBLOCK, pierwszym rozkazem w obsłudze przerwań jest *sei()*. Umożliwia to obsługę kolejnego przerwania podczas obsługi aktualnego. Używając INTERRUPT, trzeba pamiętać, że nie wszystkie przerwania po wejściu do procedury obsługi automatycznie zerują bit powodujący ich zgłoszenie, dlatego użycie INTERRUPT dla przerwań od UART czy wejścia INT wyzwalanego poziomem spowoduje przepełnienie stosu. Jeśli zależy nam na obsłudze innych przerwań podczas obsługi przerwania od np. UART, to można to zrobić na przykład tak:

```
SIGNAL ( INT_UART_vect )
{
    unsigned char a, b, d;
    a = UCSRA;
    b = UCSRB;
    d = UDR;
    sei();
//tu obsługa IRQ
}
```

Instrukcja *sei()* pojawia się po odczycie rejestru UDR. Odczyt UDR kasuje flagę RXC (UDRE), dlatego kolejne przerwanie od UART nie będzie wywołane (chyba że pojawi się kolejny znak). Rejestry UDRa, UDRb, UDRc muszą być zapamiętane przed odczytem UDR, ponieważ tak jak rejestr UDR, flagi RXB8 (błędów) są zaopatrzone w FIFO (w AVR – 2 bajty).

Trochę odbiegliśmy od głównego tematu. Jak więc poznać adres, z którego nastąpił skok do obsługi przerwania? Należy odjąć od wskaźnika stosu (SP) 2 lub 3 bajty. Tu należy wiedzieć, że adres powrotu dla mikrokontrolerów AVR z pamięcią Flash większą niż 128 kB jest 3-bajtowy. Niestety, zanim zostanie wykonany kod obsługi przerwania, kompilator odłoży na stos rejestry używane w przerwaniu (chyba że użyjemy flago NAKED). To, ile rozkazów *push* zostanie użytych, zależy od kodu procedury przerwania. Nie ma tu uniwersalnej metody – należy obejrzeć wynik kompilacji w asemblerze i wpisać odpowiednią wartość. Na szczęście, jeśli nie będziemy modyfikować naszej procedury, liczba rozkazów *push* nie zmieni się. Jako pierwszej instrukcji obsługi przerwania można by oczywiście użyć rozkazu *nop()*, w obsłudze przerwania odliczyć liczbę rozkazów *push* i zmodyfikować wskaźnik stosu. Można też użyć atrybutu *ISR_NAKED*. Wtedy rejestry nie będą odkładane na stos. Z procedury przerwania nie można wyjść, ponieważ program główny „pójdzie w maliny" z powodu zmiany stanu rejestrów. Ponadto, samemu trzeba by procedurę zakończyć rozkazem *reti()*. Przykładową procedurę obsługi przerwania od czasomierza WDG pokazano na **listingu 1**.

Uruchomienie WDG przebiega w następujący sposób:

```
InitWdgI(WDTO_500MS, &IrqWdg); // Inicjowanie przerwania od WDG
sei();  // sei() konieczne, aby działały przerwania; w przeciwnym
        //wypadku, po 2-krotnym
przepełnieniu timera WDG, nastąpi
```

```
        //restart mikrokontrolera
```

Deklaracja *#define WDGI_NAKED* zmniejszy rozmiar kodu wynikowego procedury i wywoła reset po obsłudze przerwania od WDG.

W funkcji *InitWdgI()* adres funkcji użytkownika nie jest konieczny – można wpisać zero. Nie ma to jednak większego sensu, bo przeważnie chcemy poznać adres, na którym zadziałało przerwanie. Funkcja użytkownika może wyglądać następująco:

```
void IrqWdg()
{
    PrintString_P( (char*)PSTR(CRLF"********"CRLF) );
    if ( IdSoftReset == DEF_RST_SOFT )
        PrintString_P( (char*)PSTR(„Soft Reset") );
        else PrintString_P( (char*)PSTR(„Wdg Error")
);
    PrintLongHex( adrCallWdg );
    PrintString_P( (char*)PSTR(CRLF"********"CRLF) );
}
```

Funkcja odróżnia przerwanie wywołane przez WDG od resetu programowego, w tem celu należy zadeklarować *IdSoftReset* (najlepiej jako long). W procedurze *main()* nadać wartość zmiennej *IdSoftReset*:

```
IdSoftReset =  DEF_RST_SOFT ^ 0xFFFFFF;
```

Aby wywołać reset programowy, należy wykonać:

```
IdSoftReset =  DEF_RST_SOFT; while( true ) ;
```

Jeśli nie korzystamy z funkcji *IrqWdg()*, adres, z którego nastąpiło zadziałanie WDG, będzie znajdował się w zmiennej *adrCallWdg*. Adres będzie ważny, jeśli flaga WDFR w MCUSR będzie ustawiana.

Adres powrotu z funkcji

Podczas debugowania przydatna jest znajomość adresu powrotu z funkcji. W asemblerze jest to proste – wystarczy sprawdzić adres PC na stosie. W języku C, zanim zostanie wykonany pierwszy rozkaz funkcji na stosie mogą być odkładane rejestry. Aby nie sprawdzać po każdej kompilacji liczby odłożonych danych, można posłużyć się fragmentem kodu umieszczonym pomiędzy „gwiazdkami" na list. 1. Ten fragment można zawrzeć w funkcji, ale należy pamiętać, że adres powrotu zwiększy się o adres powrotu i ewentualnie odkładane rejestry. Odkładania adresu powrotu można uniknąć, deklarując funkcję jako *inline*.

Wywołanie funkcji nie musi powodować odłożenia adresu powrotu na stosie. Nie zostanie on zapamiętany, jeśli:

Funkcję zadeklarowano jako inline.

Funkcji użyto tylko raz (!).

Jest to ostatnie (niekoniecznie) użycie funkcji w kodzie programu.

Skupmy się na drugim przypadku, dlaczego tak może się stać? Przy włączonej optymalizacji, jeśli funkcja jest użyta raz, kompilator zamiast skompilować kod C:

```
main()
{
// polecenia main
    funkcja()
// rozkazy main
}

void funkcja()
{
//rozkazy funkcji
}
```

do postaci symbolicznej:

```
main:
…rozkazy main…
call funkcja
…rozkazy main…

funkcja:
…rozkazy funkcji…
ret
```

wygeneruje kod asemblerowy w postaci:

```
main:
...rozkazy main
...rozkazy funkcji
...rozkazy main
```

Natomiast w trzecim wypadku, program w języku C w postaci:

```
main()
{
...rozkazy main
    funkcja()
    funkcja()
   funkcja()
}

void funkcja()
{
...rozkazy funkcji
}
```

skompiluje do:

Listing 3. Przykładowy sposób określenia zużycia pamięci

```
// Sekcja „.initX"  (ZERO zainicjalizowane)
unsigned char DnoStosu NOINIT;  // Kontrola stosu
void ClrIntRam(void) __attribute__ ((naked)) __attribute__ ((section („.init3")));
void ClrIntRam(void)
{
    unsigned char *AdrRam;
    // Zapisujemy od ostatniej zajetej komórki ram do wierzchołka stosu -32 bajty rezerwy
    for (AdrRam=&DnoStosu; AdrRam < (unsigned char*)RAMEND-32; AdrRam++)        // Wpisanie do IntRam $FF
    {
        *AdrRam = 0xFF;
    }
    DnoStosu  = ,@';  // Kontrola stosu
}

//Przykład sprawdzania zajętości pamięci. Najlepiej wywoływać cyklicznie, na przykład co 100ms w pętli głównej programu:
void TestStosu()
{
    unsigned char *AdrRam;
    unsigned int cnt=0;
    // Sprawdzamy od ostatniej zajetej komórki ram do wierzchołka stosu -32 bajty rezerwy
    for (AdrRam=(&DnoStosu)+1; AdrRam < (unsigned char*)RAMEND-32; AdrRam++) // Wpisanie do IntRam $FF
    {
        cnt++;
        if (*AdrRam != 0xFF)
        {
            FreeRam = cnt;  // Wolny obszar RAM'u
            if (FreeRam < 32 )
            { // Jeśli za mały obszar to generuj błąd
                PrintError( ERR_STOS );
                LedErrorOn();
                sprintf_P(str, PSTR(„ Free $%04x RAM"CRLF), FreeRam); PrintString(str);
            }
            return;
        }
    }
}
```