AVR–GCC: kompilator C mikrokontrolerów AVR, część 7
Obsługa przerwań

Przechodzimy do omówienia obsługi przerwań za pomocą programów napisanych w AVR–GCC. Jak się okazuje, jest to bardzo skuteczne narzędzie do ich obsługi.

AVR–GCC pozwala na skuteczną kontrolę obsługi sprzętowych przerwań. Jest jednak kilka szczegółów często sprawiających na początku kłopoty – zobaczymy jak sobie z nimi poradzić. Na wstępie dla krótkiego przypomnienia spójrzmy na **rys. 18**, na którym skrótowo pokazano typowy przebieg operacji podczas przerwania.

Wystąpienie sprzętowego przerwania powoduje kolejno:
- zapisanie na stosie stanu licznika rozkazów PC (zauważmy, że stos musi być wcześniej prawidłowo zainicjalizowany – AVR–GCC robi to automatycznie, musimy jedynie uważać w przypadku ATmega 128 aby wyłączyć zaprogramowany fabrycznie *fuse bit* kompatybilności z ATmega 103; w trybie kompatybilności ustawiany przez kompilator na adres RAMEND początek stosu jest fizycznie niedostępny gdyż ATmega 103 ma mniejszy RAM),
- zablokowanie wszelkich następnych przerwań,
- skok programu do odpowiedniej dla przerwania pozycji w tablicy wektorów,
- na pozycji tej musi być wpisana instrukcja skoku do właściwej procedury obsługi przerwania,
- zakończenie obsługi instrukcją *reti* powoduje odblokowanie przerwań i przywrócenie ze stosu stanu licznika PC (czyli wznowienie wykonywania programu od miejsca, w którym wystąpiło przerwanie).

Część z tych operacji jest automatyczna ale napisanie odpowiedniej procedury obsługi i wprowadzenie jej adresu do tablicy wektorów spoczywa na programiście. Popatrzmy jakie wsparcie oferuje w tym zakresie AVR–GCC.

Korzystając z już omówionych elementów rozpocznijmy w subfolderze [Przyklad–04] nowy projekt *test04* zawierający na początek pojedynczy plik *main.c* z szablonem programu głównego. Po skompilowaniu zajrzyjmy jeszcze raz (było to już wstępnie omawiane) do wygenerowanego kodu assemblera (**CTRL+F7**). Na początku znajdziemy wektory przerwań, które na razie oczywiście nie wskazują na żadne konkretne procedury i ograniczają się do skoku pod wspólny adres *__bad_interrupt* obsługi błędnego przerwania:

```
test04.elf:   file format elf32-avr
Disassembly of section.text:
00000000 <__vectors>:
   0:  12 c0       rjmp  .+36     ; 0x26
   2:  2b c0       rjmp  .+86     ; 0x5a
   4:  2a c0       rjmp  .+84     ; 0x5a
   6:  29 c0       rjmp  .+82     ; 0x5a
   8:  28 c0       rjmp  .+80     ; 0x5a
   a:  27 c0       rjmp  .+78     ; 0x5a
   c:  26 c0       rjmp  .+76     ; 0x5a
   e:  25 c0       rjmp  .+74     ; 0x5a
  10:  24 c0       rjmp  .+72     ; 0x5a
  12:  23 c0       rjmp  .+70     ; 0x5a
  14:  22 c0       rjmp  .+68     ; 0x5a
  16:  21 c0       rjmp  .+66     ; 0x5a
  18:  20 c0       rjmp  .+64     ; 0x5a
  1a:  1f c0       rjmp  .+62     ; 0x5a
  1c:  1e c0       rjmp  .+60     ; 0x5a
  1e:  1d c0       rjmp  .+58     ; 0x5a
  20:  1c c0       rjmp  .+56     ; 0x5a
  22:  1b c0       rjmp  .+54     ; 0x5a
  24:  1a c0       rjmp  .+52     ; 0x5a

[....]

0000005a <__bad_interrupt>:
  5a:  d2 cf       rjmp  .-92     ; 0x0
```

AVR–GCC oferuje makra, które automatyzują proces tworzenia procedur obsługi przerwań (*handlerów*): SIGNAL (signame) oraz INTERRUPT (signame) (znajdziemy je w pliku nagłówkowym *signal.h*). *Signame* jest nazwą potrzebnego wektora. Generalnie nazwa ta może mieć uniwersalną postać _VECTOR (numer przerwania). Jednak jest to mało czytelne i dlatego plik nagłówkowy *ioxxx.h* dla danego typu kostki zawiera dużo łatwiejsze w użyciu nazwy opisowe, np. w *io8.h* (ATmega 8) znajdziemy następujące definicje:

```
#define SIG_INTERRUPT0     __VECTOR(1)
#define SIG_INTERRUPT1     __VECTOR(2)
#define SIG_OUTPUT_COMPARE2   __VECTOR(3)
#define SIG_OVERFLOW2      __VECTOR(4)
#define SIG_INPUT_CAPTURE1 __VECTOR(5)
#define SIG_OUTPUT_COMPARE1A  __VECTOR(6)
#define SIG_OUTPUT_COMPARE1B  __VECTOR(7)
#define SIG_OVERFLOW1      __VECTOR(8)
#define SIG_OVERFLOW0      __VECTOR(9)
#define SIG_SPI          __VECTOR(10)
#define SIG_UART_RECV      __VECTOR(11)
#define SIG_UART_DATA      __VECTOR(12)
#define SIG_UART_TRANS     __VECTOR(13)
#define SIG_ADC          __VECTOR(14)
#define SIG_EEPROM_READY      __VECTOR(15)
#define SIG_COMPARATOR     __VECTOR(16)
#define SIG_2WIRE_SERIAL      __VECTOR(17)
#define SIG_SPM_READY      __VECTOR(18)
```

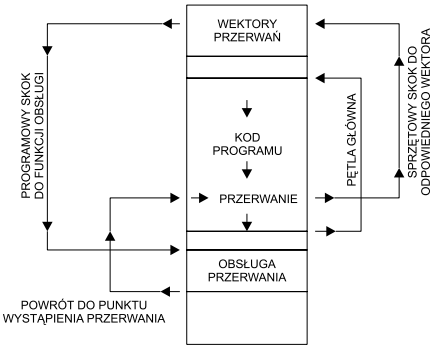
Wystarczy zdefiniować potrzebne makro, aby kompilator wygenerował podstawowy szablon kodu obsługi oraz umieścił w tablicy wektorów odpowiedni skok. Wypróbujmy to zaraz dopisując w naszym *main.c* obsługę np. dla pierwszego z brzegu przerwania zewnętrznego INT0, która na razie nie robi nic konkretnego (*SIGNAL (SIG_INTERRUPT0)* {} ;). Koniecznie musimy też dołączyć nagłówki *avr/signal.h* oraz *avr/io.h* (zwłaszcza brak *signal.h* może wprawić w zakłopotanie kilkoma mało czytelnymi w pierwszej chwili ostrzeżeniami). Wygenerowany kod wygląda następująco:

```
0000005c <__vector_1>:
SIGNAL (SIG_INTERRUPT0)
{
  5c:  1f 92       push  r1
  5e:  0f 92       push  r0
  60:  0f b6       in r0, 0x3f ; 63
  62:  0f 92       push  r0
  64:  11 24       eor   r1, r1

  6e:  0f 90       pop   r0
  70:  0f be       out   0x3f, r0 ; 63
  72:  0f 90       pop   r0
  74:  1f 90       pop   r1
  76:  18 95       reti
}
```

W prologu obsługi znajdujemy zapamiętanie na stosie wykorzystywanych przez kompilator rejestrów *r0* (rejestr tymczasowy *__tmp_reg__*) oraz *r1* (rejestr zerowy *__zero_reg__*). Następnie zachowany zostaje rejestr stanu SREG (0x3f) a rejestr *r1* zostaje wyzerowany (AVR–GCC wymaga aby był on równy zeru przy każdym wywołaniu funkcji a nie wiadomo jaka jest jego wartość w momencie wystąpienia przerwania).

Zakończenie *handlera* odtwarza poprzedni stan rejestrów oraz za-



Rys. 18. Przebieg obsługi przerwania

Rys. 19. Okienko autokompletacji przerwania w AvrSide

myka obsługę instrukcją *reti*.

Adres *handlera* (w tym przypadku 0x5c) pojawia się samoczynnie w tablicy wektorów:

```
00000000 <__vectors>:
   0:  12 c0       rjmp  .+36      ; 0x26
   2:  2c c0       rjmp  .+88      ; 0x5c
```

Zauważmy, że kod zachowuje "naturalny" dla AVR przebieg obsługi – z wszystkimi pozostałymi przerwaniami zablokowanymi do momentu wykonania instrukcji *reti*. Czasem jednak chcemy aby na inne, krytyczne czasowo przerwania reakcja następowała natychmiast – wtedy musimy w naszej obsłudze samodzielnie je ponownie włączyć. Robi to samoczynnie drugie makro. Sprawdźmy, że wywołanie:

```
INTERRUPT (SIG_INTERRUPT0) {} ;
```

generuje taki sam kod ale rozpoczynający się włączającą przerwania instrukcją *sei*. Jednak trzeba ten sposób stosować w odpowiednią uwagą gdyż może spowodować kilka niespodzianek (do czego zaraz wrócimy).

Makra *SIGNAL* oraz *INTERRUPT* zadziałają nawet w przypadku wstawienia dowolnej nazwy nie odpowiadającej żadnemu z rzeczywistych wektorów przerwań. Kod zostanie wygenerowany i umieszczony w programie ale oczywiście kompilator nie będzie mógł mu przypisać żadnej pozycji w tablicy wektorów. W niektórych przypadkach (o czym za chwilę) zrobimy tak celowo. Niestety zazwyczaj ta cecha jest raczej źródłem zaskakujących błędów wynikających np. z drobnej pomyłki literowej w nazwie wektora albo ze skopiowania *handlera* z programu dla innej kostki. Kompilator nie zgłasza w takim przypadku żadnych wątpliwości a przerwanie pozostaje nie obsługiwane. (W niektórych wersjach AVR–GCC są zdaje się łatki powodujące poinformowanie o występującej rozbieżności ale generalnie lepiej na to nie liczyć).

AvrSide wyposażono w dynamiczną podpowiedź właściwych nazw (**CTRL+L**) co pozwala wyeliminować taki błąd (**rys. 19**) jednakże w razie jakichś kłopotów z działaniem programu zajrzyjmy zawsze do tablicy wektorów i upewnijmy się czy zawiera ona właściwy skok do istniejącego kodu obsługi przerwania (np. taką niespodziankę miałem po skopiowaniu kodu obsługi TWI z projektu ATmega8 do ATmega88: cały interfejs działa i jest opisany identycznie z wyjątkiem właśnie zmienionej – z *SIG_2WIRE_SERIAL* na *SIG_TWI* – nazwy wektora).

Pojawia się od razu pytanie kiedy stosować *SIGNAL* a kiedy *INTERRUPT*. Zawsze będzie to oczywiście zależeć głównie od potrzeb konkretnego programu, jednak można sformułować kilka podstawowych reguł:

A. W niektórych przypadkach *INTERRUPT* nie możemy używać w ogóle. Przypomnijmy sobie, że przerwanie w AVR może być wywołane zdarzeniem (ustawiającym odpowiednią flagę we właściwym rejestrze) – np. przepełnieniem licznika; albo warunkiem – przerwanie jest aktywne cały czas dopóki zachodzi określona sytuacja – np. w rejestrze odbiornika USART znajduje się nie odczytany znak. Dodatkowa komplikacja to fakt, że chociaż zazwyczaj rozpoczęcie obsługi przerwania zdarzeniowego powoduje w chwili skoku do wektora przerwania samoczynne (sprzętowe) zgaszenie flagi to jednak są od tego wyjątki. np. przerwanie magistrali TWI (i2c). W obu ostatnich przypadkach usunięcie przyczyny przerwania (eliminacja warunku albo zgaszenie flagi) musi byc wykonane programowo wewnątrz funkcji obsługi. Na przykład dla wspomnianego odbiornika USART będzie to odczyt rejestru UDR. W tych właśnie przypadkach generowane przez *INTERRUPT* odblokowanie przerwań na samym początku *handlera* spowoduje natychmiastowe wywołanie tego samego przerwania – program jeszcze nie dotarł i nigdy nie będzie mógł dotrzeć do fragmentu kodu wyłączającego warunek wyzwalający (spójrzmy jeszcze raz na rys. 18 – zaraz po wejściu do funkcji obsługi i wykonaniu *sei* nastąpi ponowny skok do wektora). Taka aplikacja nie ma szans na poprawne działanie.

Pomyłka ta pojawia się na tyle często, że autorzy *avr–libc* zaczęli nawet rozważać ewentualną zmianę wprowadzającego w błąd nazewnictwa – ale to na razie tylko wstępne propozycje.

B. Jak łatwo się domyśleć, *INTERRUPT* ma służyć do zastąpienia nieobecnej w AVR kontroli priorytetu przerwań. Pozwala na obsłużenie krytycznego czasowo przerwania niezależnie od faktu czy program wykonuje pętlę główną czy też już obsługuje zgłoszone wcześniej przerwanie o mniejszym dla nas znaczeniu. Sprawa jest prosta jeśli mamy do czynienia z dwoma przerwaniami: dla podrzędnego używamy makra *INTERRUPT* co pozwala na praktycznie natychmiastową obsługę drugiego – ważniejszego. Gorzej jeśli przerwań jest kilka – globalne odblokowanie umożliwia wykonywanie wszystkich pozostałych co nie zawsze jest pożądane. W takim przypadku selektywne podwyższenie priorytetu tylko jednego wybranego przerwania wymaga każdorazowo szczegółowego przełączania konfiguracji zezwoleń na poszczególne przerwania w kodzie *handlerów*.

C. Z powyższych ograniczeń wynika, że na ogół domyślnym sposobem obsługi będzie *SIGNAL*, natomiast *INTERRUPT* użyjemy w specyficznych przypadkach, dokładnie rozważając potrzeby, korzyści i możliwości wystąpienia niepożądanych efektów.

D. Ponieważ makro *SIGNAL* blokuje wszystkie inne przerwania (zgod-

List. 3. Plik z programem obsługi timerów

```
// obsługa timerów

#include „projdat.h"
#include <avr/io.h>
#include <avr/signal.h>

volatile uchar T2_counter;

/* Licznik T2 posłuży nam jako podstawowy timer systemowy, na bazie którego będziemy
  realizować cykliczne akcje, odliczać timeouty oraz uruchomimy prototyp zegara.
  Wykorzystamy tryb pracy CTC - wygodny ze względu na samoczynne zerowanie licznika.
*/

void InitT2(void)
// atmega 8 pracuje z wewnętrznym oscylatorem 8 MHz - pojedynczy cykl ma długość 0.125 us
// (1 / 8000000)
// Jeśli ustawimy preskaler = 64 pojedynczy tick licznika ma 64 * 0.125 = 8 us
// Dla uzyskania przerwania licznika co 1 ms ustawimy jego wartość przeładowania na 124
// (8 * (124+1) = 1000 us = 1 ms)
{
 OCR2 = 124;              // wartość przeładowania w trybie CTC
 TCCR2 = _BV(WGM21) | _BV(CS22);   // tryb CTC bez zewnętrznego wyjścia, preskaler 64
 TIMSK |= _BV(OCIE2);         // włączenie przerwania CTC
}

SIGNAL (SIG_OUTPUT_COMPARE2)
{
 if (++T2_counter == 100);
  {
   T2_counter = 0;
   MST00_FLAG = true;
  }
}
```

nie ze sprzętowym działaniem mikrokontrolera) zazwyczaj powinniśmy zadbać aby funkcje obsługi były jak najkrótsze: nie umieszczać w nich skomplikowanych przeliczeń lub konwersji, obsługi zewnętrznych urządzeń itp. a już w żadnym przypadku nie wstawiać do nich programowych pętli opóźniających. Takie poczynania mogą doprowadzić do utraty jakichś innych przerwań, których mikrokontroler nie zdąży obsłużyć.

W wielu typowych zastosowaniach mikrokontrolera (jak różne transmisje, pomiary, akwizycja danych itp.) dobrze sprawdza się następująca recepta:

– stosujemy wyłącznie makra SIGNAL,
– funkcje obsługi skracamy do niezbędnego minimum i przekazujemy z nich, za pośrednictwem zmiennych logicznych lub flag bitowych, do pętli głównej informację o konieczności realizacji czynności związanych z wystąpieniem przerwania,
– pętla główna sprawdza stan takich flag a w momencie ich ustawienia wykonuje potrzebne działania, niekiedy mocno pracochłonne, bez blokowania dostępu do przerwań.

W ten sposób żadne z przerwań nie przejmuje sterowania na zbyt długi czas a wszystkie zadania są wykonywane mniej więcej równomiernie (zauważmy, że jest to bardzo uproszczony model znanego z dużych systemów programowania zdarzeniowego). Zróbmy sobie od razu tego typu przykład wykorzystujący wiele wcześniej omawianych technik. Do projektu dodajmy plik *timers.c* o zawartości pokazanej na **list. 3** oraz plik nagłówkowy *projdat.h* gromadzący globalne zmienne, deklaracje funkcji i różne definicje (**list. 4**). Do pliku *main.c* dodamy kod pokazany na **list. 5**.

List. 4. Listing pliku nagłówkowego *projdat.h*

```
// plik nagłówkowy globalnych danych
projektu
#ifndef _PROJ_DAT_H_
#define _PROJ_DAT_H_
// #include:
#include „mynames.h”

// #define:

// definicje typów typedef

// dane globalne

volatile Flags SysFlags;

#define MS100_FLAG SysFlags.Bits.Flag1


#ifdef _MAIN_MOD_
// definicje danych - tylko w module
main()
// char x;

#else
// deklaracje danych jako importowanych
- w każdym innym module
// extern char x;

#endif

// deklaracje funkcji
// extern char Myfunc(int,char);

extern void InitT2(void);

#endif
```

List. 5. Główny moduł przykładowego projektu

```
// główny moduł projektu
#define _MAIN_MOD_ 1
// pliki dołączone (include):
#include „projdat.h”
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/eeprom.h>
#include <avr/signal.h>
#include <string.h>

#define MS100_DELAY 5

// dane:
static char Ms100_counter;
static volatile uchar LedState = 1;

// funkcje:

INTERRUPT (SIG_INTERRUPT0)
{
}

//====================
// funkcja main()
int main(void)
{
 // inicjalizacja
 OSCCAL=eeprom_read_byte((uchar*)E-
2END);   // zapis kalibracji w ostat-
niej komórce eeprom
 DDRB=0xff;

 InitT2();
 sei();

 // pętla główna
 while (1)
 {
  if (MS100_FLAG)
    {
     MS100_FLAG = false;
     if (++Ms100_counter == MS100_DELAY)
      {
       Ms100_counter = 0;
       // nasza okresowa akcja (przełą-
czenie wyjścia) uruchamiana
       // zegarem systemowym co 100ms *
MS100_DELAY (0,5s)
       PORTB=LedState;
       if(LedState==128) LedState=1;
else LedState = LedState<<1;
      }
    }
 }
}
```

List. 6. Plik nagłówkowy z deklaracjiami ulubionycyh typów i definicji

```
// ulubione oznaczenia

#ifndef _MY_NAMES_H
#define _MY_NAMES_H

#include <stdbool.h>

#define uint unsigned int
#define uchar unsigned char
#define ulong unsigned long

#define forever while(1)
#define EEPROM __attribute__ ((sec-
tion(„.eeprom”)))
#define NOINIT __attribute__ ((sec-
tion(„.noinit”)))
#define NAKED __attribute__ ((naked))

typedef struct
{
 uchar Flag1:1;
 uchar Flag2:1;
 uchar Flag3:1;
 uchar Flag4:1;
 uchar Flag5:1;
 uchar Flag6:1;
 uchar Flag7:1;
 uchar Flag8:1;
} FlagBits;

typedef union
{
 FlagBits Bits;
 uchar   Byte;
} Flags;

#endif
```

Plik nagłówkowy *projdat.h* korzysta również z ogólnego, wspólnego nagłówka *mynames.h* zawierającego ulubione typy i definicje (umieściłem go w folderze *\AvrSide\Myinc* podając odpowiednią ścieżkę w opcjach projektu) – **list. 6**.

Jak widać działanie programu sprowadza się do kilku podstawowych operacji:

– inicjalizacja ustawia potrzebne nam linie I/O (PORTB jako wyjściowy), konfiguruje *timer* T2 (niestety nie dopisałem jeszcze w AvrSide kreatora automatycznej konfiguracji *timerów*, jest ona wykonana ręcznie ale została dosyć szczegółowo opisana w komentarzu) i na koniec uruchamia system przerwań (dodatkowo znajdujemy tu ustawienie kalibracji *OSCCAL* dla pracy z wewnętrznym oscylatorem 8 MHz, potrzebna wartość jest przechowana w ostatnim bajcie eeprom; taką metodą posługuje się wbudowany w AvrSide programator usb ale każdy użytkownik prawdopodobnie zastosuje jakiś własny sposób pasujący do posiadanego sprzętu);
– 1 ms przerwanie *timera* odmierza (przy pomocy dodatkowego lokalnego programowego licznika *T2_counter*) okresy 100 ms i ustawia flagę bitową *MS100_FLAG* zdefiniowaną globalnie w *projdat.h* (zwróćmy jeszcze raz uwagę na niezbędne klasyfikatory *volatile* dla zmiennych współużytkowanych przez program główny oraz *handler* przerwania);
– pętla główna śledzi nieustannie stan flagi *MS100_FLAG*, po jej ustawieniu przystępuje do wykonania przypisanych fladze procedur:
– kasuje flagę (co jest konieczne aby wykonanie było tylko jednokrotne);
– przy pomocy lokalnego programowego licznika *Ms100_counter* odmierza interwał czasowy określony stałą *MS100_DELAY* (w przykładzie jest to 0,5 s);
– co 0,5 s przesuwa okrężnie pojedynczy ustawiony bit w lokalnej zmiennej stanu portu *LedState* oraz przepisuje ją na wyjście portu B.

Jerzy Szczesiul, EP
jerzy.szczesiul@ep.com.pl

UWAGA!
Środowisko IDE dla AVR-GCC opracowane przez autora artykułu można pobrać ze strony http://avrside.ep.com.pl.