AVR–GCC: kompilator C mikrokontrolerów AVR, część 8

Obsługa przerwań

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

Po uruchomieniu sesji AvrStudio dla naszego projektu (**rys. 20**) możemy ustawić *breakpoint* i obejrzeć pracę programu. Na tym możliwości wizualne AvrStudio się kończą. Jeśli zechcemy zaprezentować symulację w bardziej naturalny sposób musimy sięgnąć po dodatkowe wyposażenie. Jest nim bezpłatne rozszerzenie AvrStudio o nazwie *Hapsim* (*Helmi's AVR Periphery Simulator*) autorstwa Helmuta Wallnera (*http://www.helmix.at/hapsim.htm#hapsimdownload*). Pozwala ono – podczas pracy symulatora AvrStudio – na wizualizację pracy peryferiów mikrokontrolera (np. diod led, wyświetlacza graficznego, terminala tekstowego).

Środowisko AvrSide zostało przystosowane do współpracy z *Hapsim*. Każdorazowo podczas zapisu nowego projektu w subfolderze projektu jest samoczynnie tworzony plik konfiguracyjny sesji *Hapsim*: nazwa_projektu.xml (na bazie ustawień zawartych w szablonie *\AvrSide\bin\haptemplate.xml*). Natomiast pobrane ze strony pliki wykonawcze *Hapsim* należy umieścić w folderze *\AvrSide\Hapsim*. Teraz z poziomu menu AvrSide możemy uruchomić symulator od razu z konfiguracją przypisaną projektowi (warunkiem jest obecność pracującego AvrStudio, jeśli nie jest on spełniony komenda menu, zamiast uruchamiać Hapsima z występującym w takim przypadku ostrzeżeniem, rozpoczyna zapobiegawczo od startu AvrStudio).

W naszym przykładzie uruchomimy *Hapsima*, zmienimy jego konfigurację z szablonowej (wyświetlacz LCD) na zespół 8 diod led (przypisanych do portu B, bez inwersji, kolory dowolne) – jak na **rys. 21**. Konfigurację tę wpiszemy poleceniem *Save* do pliku *\Projects\Kurs\Przyklad–04\test04.xml* potwierdzając jego nadpisanie.

Rys. 20. Kontrola pracy testowego programu w symulatorze AvrStudio

Teraz po uruchomieniu (F5) sesji AvrStudio (i ewentualnym przywołaniu *Hapsima* na ekran) ujrzymy „rzeczywistą" pracę portu Atmegi z podłączonymi diodami – **rys. 22**.

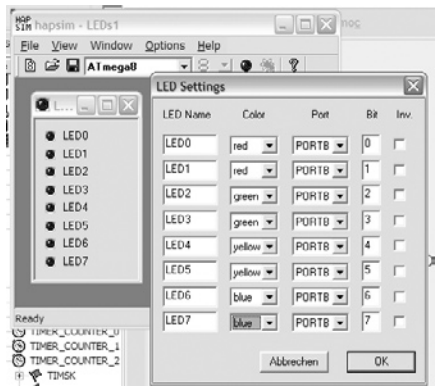
Hapsim potrafi czasem znacznie spowolnić pracę AvrStudio. Wtedy możemy wykorzystać kompilację warunkową, np. do innego ustawienia *timerów* dla prób z *Hapsim* i dla wersji docelowej. AvrSide wspiera taką operację definiując w wywołaniu kompilatora symbol HAPSIM po zaznaczeniu odpowiedniego *checkboxa* w oknie opcji projektu (zakładka *Kompilator*).

Oczywiście opisywany przykład ma znikomą wartość praktyczną i służy głównie zaprezentowaniu narzędzia. Symulacja terminala lub wyświetlacza lcd może być sporo bardziej przydatna (chociaż nie zawsze będzie równoznaczna z prawidłowym działaniem rzeczywistego docelowego układu). Po tej przykładowej prezentacji zastosowania ogólnych zasad wróćmy jeszcze raz do mniej typowych problemów związanych z przerwaniami.

Tworząc automatycznie szablon *handlera* makrem *SIGNAL* lub *INTERRUPT* avr–gcc chroni używane przez siebie rejestry (przepisując je na stos w prologu i odtwarzając w epilogu). Dopiszmy do wcześniejszego testowego przerwania *SIG_INTERRUPT0* jakąś najprostszą operację, np.

```
INTERRUPT (SIG_INTERRUP-
T0)
{
 PORTB=0x20;
}
```

i sprawdźmy, że chroniony jest tylko użyty w kodzie rejestr r24:



Rys. 21. Konfiguracja sesji Hapsima dla naszego przykładu

```
INTERRUPT (SIG_INTERRUPT0)
{
 62:  78 94       sei
 64:  1f 92       push  r1
 66:  0f 92       push  r0
 68:  0f b6       in r0, 0x3f ; 63
 6a:  0f 92       push  r0
 6c:  11 24       eor   r1, r1
 6e:  8f 93       push  r24
 PORTB=0x20;
 70:  80 e2       ldi   r24, 0x20   ; 32
 72:  88 bb       out   0x18, r24   ; 24
 74:  8f 91       pop   r24
 76:  0f 90       pop   r0
 78:  0f be       out   0x3f, r0 ; 63
 7a:  0f 90       pop   r0
 7c:  1f 90       pop   r1
 7e:  18 95       reti
```

A teraz zadeklarujmy (*extern void Little(void);* w *projdat.h*) i zdefiniujmy w *main.c*:

```
void Little(void)
{
 PORTB=0x20;
}
```

niewielką funkcję, która robi dokładnie to samo używając tego samego rejestru:

```
void Little(void)
{
 PORTB=0x20;
 5c:  80 e2       ldi   r24, 0x20   ; 32
 5e:  88 bb       out   0x18, r24   ; 24
 60:  08 95       ret
```

Spróbujmy wykonać nową funkcję wewnątrz obsługi przerwania:

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

i zobaczmy jak zmienił się kod wynikowy – spotyka nas niemiła niespodzianka gdyż uległ on znacznemu (i w naszym przypadku zbędnemu) wydłużeniu:

```
INTERRUPT (SIG_INTERRUPT0)
{
```

```
 62:  78 94      sei
 64:  1f 92      push  r1
 66:  0f 92      push  r0
 68:  0f b6      in r0, 0x3f ; 63
 6a:  0f 92      push  r0
 6c:  11 24      eor   r1, r1
 6e:  2f 93      push  r18
 70:  3f 93      push  r19
 72:  4f 93      push  r20
 74:  5f 93      push  r21
 76:  6f 93      push  r22
 78:  7f 93      push  r23
 7a:  8f 93      push  r24
 7c:  9f 93      push  r25
 7e:  af 93      push  r26
 80:  bf 93      push  r27
 82:  ef 93      push  r30
 84:  ff 93      push  r31
Little();
 86:  ea df      rcall .-44     ; 0x5c
 88:  ff 91      pop   r31
 8a:  ef 91      pop   r30
 8c:  bf 91      pop   r27
 8e:  af 91      pop   r26
 90:  9f 91      pop   r25
 92:  8f 91      pop   r24
 94:  7f 91      pop   r23
 96:  6f 91      pop   r22
 98:  5f 91      pop   r21
 9a:  4f 91      pop   r20
 9c:  3f 91      pop   r19
 9e:  2f 91      pop   r18
 a0:  0f 90      pop   r0
 a2:  0f be      out   0x3f, r0 ; 63
 a4:  0f 90      pop   r0
 a6:  1f 90      pop   r1
 a8:  18 95      reti
```

Analiza użycia rejestrów nie sięga w głąb zagnieżdżonych funkcji – kompilator chroni na wszelki wypadek wszystkie, które mogą być potencjalnie zagrożone. Chociaż ten efekt na ogół nie powoduje jakichś problemów z działaniem programu to jednak może stwarzać kłopot w przypadkach krytycznych czasowo.

W takiej sytuacji możemy w razie potrzeby przejąć całkowitą kontrolę nad kodem *handlera*. Zamiast stosować omawiane powyżej makra zadeklarujmy po prostu funkcję o nazwie zgodnej z nazwą potrzebnego wektora i nadajmy jej atrybut *naked*. W ten sposób kompilator przypisze wektorowi skok do procedury całkowicie pozbawionej prologu i epilogu (nawet instrukcji powrotu *ret* – zgodnie ze znaczeniem atrybutu: ”goła”, “ogołocona”). Potrzebny prolog i epilog – ograniczone tylko do naszych rzeczywistych potrzeb – wpiszemy samodzielnie jako wstawkę asemblerową *inline* (możemy oczywiście jako podpowiedź wykorzystać automatyczny kod tworzony przez makra). Może to wyglądać np. tak:

```
void SIG_INTERRUPT0(void) NAKED; //
NAKED to własna skrócona definicja z my-
names.h

void SIG_INTERRUPT0(void)
{
 asm volatile („sei” „\n\t”
        „push r1” „\n\t”
        „push r0” „\n\t”
        „in r0,0x3f” „\n\t”
        „push r0” „\n\t”
        „eor r1,r1” „\n\t”
        „push r24”);
 Little();
 asm volatile („pop r24” „\n\t”
        „pop r0” „\n\t”
        „push r0” „\n\t”
        „out 0x3f,r0” „\n\t”
        „pop r0” „\n\t”
        „pop r1” „\n\t”
        „reti”);
}
```

Takie rozwiązanie oczywiście wymaga zwiększonej uwagi – np. wprowadzenie do *Little* zmian powodujących użycie następnych rejestrów wymagać będzie również równoległej ręcznej korekty powyższego prologu i epilogu.

Łatwo zauważyć, że w podobny sposób możemy też zminimalizować i maksymalnie przyśpieszyć obsługę przerwania, która nie potrzebuje nawet niewielkiej podstawowej ochrony – jak nasz pierwotny przykładowy *handler* ustawiający tylko port B , gdzie *r0* i *r1* nie są w ogóle użyte, zaś instrukcje *ldi* oraz *out* nie zmieniają flag w rejestrze stanu. Wystarczy nam wtedy z powodzeniem zapis (oczywiście *sei* także według potrzeb):

```
void SIG_INTERRUPT0(void)
{
 asm volatile („sei” „\n\t”
        „push r24” „\n\t”
        „ldi r24,0x20” „\n\t”
        „out 0x18,r24” „\n\t”
        „pop r24” „\n\t”
        „reti”);
}
```

Innym przypadkiem ręcznej ingerencji programisty mogą być procedury obsługi przerwań napisane całkowicie w asemblerze chociaż niekoniecznie najkrótsze. Dłuższa porcja kodu jest dosyć uciążliwa do wpisywania w formie wstawki *inline* (to oczywiście subiektywna opinia) – dużo wygodniej jest wtedy przenieść cały kod do asemblerowego modułu **.s*. Dla przykładu dodajmy do projektu stronę z plikiem asemblerowym *ints.s* o treści:

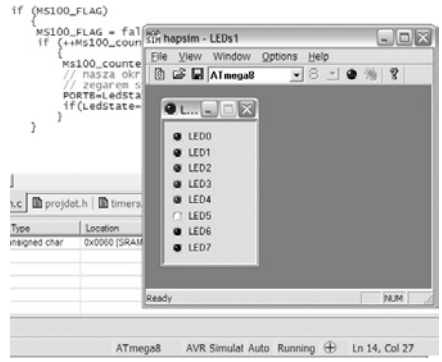
```
#define __SFR_OFFSET 0
#include <avr/io.h>

.global SIG_INTERRUPT1
   .section .text,”ax”,@progbits

SIG_INTERRUPT1:
  push r24
  ldi r24,0x40
  out PORTB,r24
  pop r24
  reti
```

Po skompilowaniu znajdziemy w kodzie obsługę przerwania pod nazwą *__vector_2* wraz z odpowiednim adresem skoku w tablicy wektorów. Jednak kilka spraw wymaga dodatkowego wyjaśnienia:

- Nazwa procedury jest zgodna z nazwą wektora z pliku nagłówkowego danego układu, który dołączamy dyrektywą *#include <avr/io.h>* tak jak dla pliku C.
- Aby uwzględnić tę nazwę plik musi być poddany obróbce w preprocesorze. Avr–gcc uruchamia samoczynnie preprocesor po napotkaniu rozszerzenia **.S* (natomiast pomija go przy rozszerzeniu **.s* małe). Jednak – ponieważ Windows generalnie nie rozróżnia wielkości liter – dla uniknięcia nieporozumień



Rys. 22. Ekran symulatora Hapsim

przyjąłem w AvrSide wyłącznie rozszerzenia małe.*s*, natomiast akcja preprocesora jest jawnie wymuszana odpowiednią opcją linii komendy kompilatora (*–x assembler–with–cpp* , możemy ją odnaleźć w pliku logu).

- Dołączenie <avr/io.h> umożliwia również używanie wszelkich symbolicznych nazw rejestrów (jak PORTB). Jednak sposób zdefiniowania rejestrów I/O w avr–libc nie jest zgodny z wymaganym w asemblerze adresowaniem przestrzeni I/O (z przesunięciem 0x20). Definicja *__SFR_OFFSET 0* służy właśnie do skorygowania tej rozbieżności (możemy sprawdzić, że bez niej w wynikowym kodzie ładowany jest nie rejestr portu B – 0x18 ale rejestr 0x38). Taki sam efekt uzyskamy konwertując SFR na adres liczbowy przy pomocy *_SFR_IO_ADDR (PORTB)* , co jest częściej zalecane w wielu poradach ale przy dłuższym kodzie wymaga większej ilości pisania. Dla zapoznania się z tymi niuansami warto przejrzeć plik nagłówkowy *\avr\include\avr\sfr_defs.h* w folderze kompilatora.
- Opisy dodatkowych klasyfikatorów sekcji znajdziemy przeglądając dokumentację avr–libc oraz avr–as (*ax* oznacza alokowalny + wykonywalny; *@progbits* informuje, że sekcja zawiera dane); klasyfikatory te nie są niezbędne do skompilowania (wystarcza samo podanie nazwy sekcji.*section .text)*

Modułów i funkcji asemblerowych nie będziemy niestety mogli obejrzeć w postaci źródłowej w debugerze AvrStudio. Brak w nich wymaganej do tego odpowiednio sformatowanej informacji. Ręczne dopisywanie jest raczej mało realne gdyż zapis jest zbyt złożony (możemy go podpatrzeć w do-

wolnym tymczasowym pliku.s wygenerowanym z modułu **.c* poleceniem menu *Sprawdź poprawność kodu*), nie zanosi się również na opracowanie do tego automatycznego narzędzia. W takich przypadkach pozostaje jedynie debugowanie na poziomie kodu asemblera (przykład z przerwaniami *INT0* oraz *INT1* szybko sprawdzimy w pracy krokowej ręcznie przestawiając w AvrStudio potrzebne bity w rejestrze *GICR* oraz *PIND*).

Opisane powyżej użycie własnych *handlerów* asemblerowych lub *naked* może się też przydać gdy zechcemy przypisać kilku różnym wektorom przerwań wspólną procedurę obsługi. Można to oczywiście zrealizować tradycyjnie, wywołując z kilku automatycznych *handlerów* tę samą funkcję, ale przed chwilą zobaczyliśmy, że może się to niekorzystnie odbić na szybkości kodu. Użyjmy więc makra *SIGNAL* (lub *INTERRUPT*) z nazwą wektora inną niż istniejące w kostce – makro wygeneruje kod *handlera* ale nie przypisze mu żadnego adresu skoku w obszarze wektorów przerwań (jest to właśnie wspomniany wcześniej przypadek celowego wykorzystania zapisu, który przy zwykłym użyciu zazwyczaj powoduje błąd). W tym *hadlerze* wpisujemy kod wspólnej obsługi dla kilku różnych przerwań. Natomiast wybrane do tej obsługi przerwania opisujemy procedurami *naked* zawierającymi tylko i wyłącznie skok pod jego adres. Przykładowo dla dwóch przerwań nadajnika uart moglibyśmy zapisać:

```
void SIG_UART_DATA (void) NAKED;
void SIG_UART_TRANS (void) NAKED;

SIGNAL (SIG_COMMONINT)
{
 // wykonanie wspólnej obsługi dla
sig_uart_data
 // oraz sig_uart_trans
}

void SIG_UART_DATA(void)
{
 asm volatile(„rjmp SIG_COMMONINT");
}

void SIG_UART_TRANS(void)
{
 asm volatile(„rjmp SIG_COMMONINT");
}
```

To samo możemy wykonać w pliku asemblerowym – należy jedynie dołączyć do kodu asm dyrektywę *.extern SIG_COMMONINT*. W kodzie wynikowym sprawdzimy, że rzeczywiście otrzymaliśmy skoki w potrzebne miejsca: z tablicy wektorów do krótkich procedur pośrednich a następnie do wspólnego *handlera* zakończonego instrukcją powrotu *reti*.

Przy wstępnym omawianiu automatycznego kodu generowanego przez avr–gcc stwierdziliśmy, że przerwania nie obsługiwane w programie mają przypisany domyślny skok do etykiety *__bad_interrupt*. Takie określenie (złe, błędne przerwanie) oczywiście nie oznacza, że jakieś przerwania są „lepsze" a inne „gorsze". Chodzi natomiast o podkreślenie, że w prawidłowo skonstruowanym i oprogramowanym urządzeniu brak obsługi przerwania oznacza, iż w żadnych okolicznościach z cała pewnością ono nie wystąpi. Wyzwolenie takiego przerwania (np. poprzez omyłkowe odblokowanie, pozostawienie „pływającego", narażonego na zakłócenia pinu itp.) świadczy po prostu o błędzie wykonawczym. Dlatego domyślna akcja w takim przypadku nie jest zbyt rozbudowana – powoduje skok pod adres 0 (nawet bez powrotu z przerwania *reti*). Czasem w trakcie uruchamiania chcielibyśmy tę akcję rozszerzyć – np. o zliczanie nieprzewidzianych przerwań. Avr–libc daje taką możliwość: wystarczy zdefiniować obsługę wektora domyślnego *__vector_default* i tam wpisać potrzebny kod:

```
SIGNAL(__vector_default)
{
 // obsługa nieprzewidzianych przerwań
}
```

W kodzie wynikowym sprawdzimy, że teraz etykieta *__bad_interrupt* nie zawiera już skoku pod adres 0 ale skok do naszego nowego domyślnego *handlera*.

Z powyższych rozważań widać, że avr–gcc daje nam praktycznie pełną kontrolę nad mechanizmem przerwań mikrokontrolera. Jak często te możliwości wykorzystamy – będzie zależeć od naszych preferencji i przyzwyczajeń. Moim subiektywnym zdaniem warto stosować przerwania jak najczęściej, eliminując wszelkie *pollingi* (czyli cykliczne sprawdzanie flag w rejestrach) i pętle oczekujące na wykonanie operacji – chociaż są one bardzo popularne w rozmaitych przykładach i kursach programowania. Dodatkowy nakład pracy szybko zwróci się z nawiązką w postaci bardziej logicznej i przejrzystej struktury programu oraz jego szybszego i płynniejszego działania.

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.