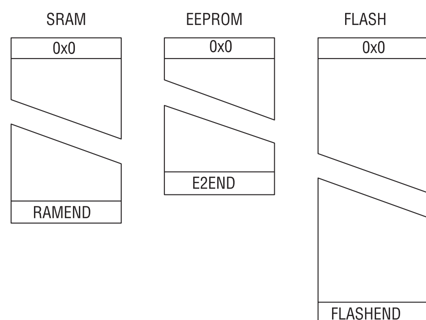
AVR–GCC: kompilator C dla mikrokontrolerów AVR, część 13
Obsługa obszarów pamięci mikrokontrolerów AVR, część 1

Jedną z wielkich zalet mikrokontrolerów AVR (podobnie zresztą jak układów wielu innych rodzin) jest integracja w jednym układzie wszystkich potrzebnych rodzajów pamięci (SRAM, EEPROM i Flash), co pozwala na znaczne uproszczenie budowanych urządzeń. Ich obsługa nie zawsze jest jednoznaczna, co sprawia duże trudności programistom, zwłaszcza tym, którzy są przyzwyczajeni do korzystania z pełni możliwości języka C.

Pamięci te są umieszczone w oddzielnych obszarach adresowych i wyposażone w oddzielne sprzętowe mechanizmy dostępu (architektura typu *Harvard*) obsługiwane różnymi instrukcjami maszynowymi – **rys. 30**. W obszarze *RAM* dodatkowo wydzielone są sekcje rejestrów roboczych i rejestrów wejść/wyjść wyposażone w odrębny sposób obsługi (m.in. występuje adresowanie bitowe) – **rys. 31**.

Wszystko to jest „chlebem powszednim" dla konstruktorów dobrze zaznajomionych z budową mikrokontrolerów i programowaniem w assemblerze. Jednak z punktu widzenia języka C sprawy się komplikują. Język ten pochodzi ze świata dużych maszyn o całkowicie odmiennej organizacji pamięci (jeden obszar o wspólnym sposobie adresowania i dostępu – czyli architektura *von Neumanna*) i nie posiada żadnych standardowych mechanizmów wspierających powyższe zróżnicowanie.

Kompilatory C tworzone dla konkretnych rodzin mikrokontrolerów są od razu wyposażane w specyficzne rozszerzenia obsługi różnych pamięci. Na przykład w *SDCC* dla rodziny '51 klasyfikator *data* identyfikuje podstawową pamięć danych, *idata* oznacza obszar rozszerzonej pamięci wewnętrznej adresowanej pośrednio, *xdata* – zewnętrzną pamięć RAM, zaś *code* oznacza umieszczenie stałej (wyłącznie do odczytu) ulokowanej w pamięci programu. Przypisanie zmiennej lub stałej określonego klasyfikatora pozwala kompilatorowi w momencie jej użycia na generację kodu odpowiedniego dla zadeklarowanego obszaru pamięci (np. zastosowanie instrukcji *MOVX* w przypadku zewnętrznej pamięci '51). Na ogół jest modyfikowana też budowa wskaźników tak, aby mieściły one dodatkową informację o typie pamięci dla aktualnie wskazywanego obiektu.



Rys. 30. Rodzaje pamięci w AVR. Każda z nich jest liniowym obszarem adresowanym od zera. Stałe *RAMEND, E2END* i *FLASHEND* są zdefiniowane w plikach nagłówkowych poszczególnych typów mikrokontrolerów

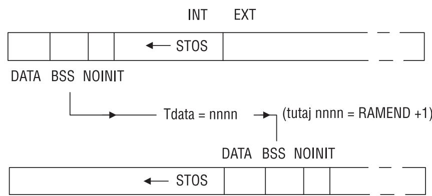
Rejestry robocze	Przestrzeń adresowa danych
R0	$0000
R1	$0001
R2	$0002
R29	$001D
R30	$001E
R31	$001F
Rejestry WE/WY	
$00	$0020
$01	$0021
$02	$0022
$3D	$005D
$3E	$005E
$3F	$005F
	Wewnętrzna pamięć SRAM
	$0060
	$0061
	$045E
	$045F

Rys. 31. Struktura wewnętrznej pamięci RAM mikrokontrolerów AVR

Niestety AVR–GCC nie jest narzędziem od podstaw dedykowanym rodzinie AVR, ale jedynie jednym z wielu portów uniwersalnego zestawu kompilatorów GNU GCC. GCC natomiast – tak samo jak nadmieniono powyżej o języku C – jest przystosowany do ciągłego modelu pamięci. Skłonienie go do współpracy z tak nietypową dla niego platformą sprzętową wymagało wielu wysiłków i kompromisów. Nie udało się niestety do tej pory uzyskać pełnej wygody użytkowania spotykanej w komercyjnych narzędziach – AVR–GCC wymaga dodatkowo poznania kilku specyficznych dla niego technik. Nie jest to mocno uciążliwe, ale potrafi zaskoczyć programistów przyzwyczajonych do PC lub innych kompilatorów C dla AVR.

Obsługa SFR oraz obszaru I/O

Ten temat był już wielokrotnie omawiany w poprzednich odcinkach. Obecnie AVR–GCC oferuje pełne wsparcie dla takich operacji. Nie są konieczne dawniejsze dodatkowe makra typu *inp* czy *outp* – stosujemy zwyczajne instrukcje przypisania. Przy tym optymalizator potrafi samodzielnie określić możliwości zaadresowania danego rejestru i wybrać najprostsze i najkrótsze instrukcje (*in*, *out*, *cbi*, *sbi*) – pokazywaliśmy to na różnych przykładach. Jedynym mankamentem tego postępu



Rys. 32. Przemieszczanie sekcji danych do zewnętrznej pamięci

jest częściowy brak kompatybilności ze starszymi projektami – jednak dla rozpoczynających naukę nie będzie to żadną przeszkodą.

Obsługa pamięci danych

Sposób zarządzania pamięcią danych także był już dość dokładnie przedstawiony (podział na sekcje, lokalizacja stosu itp.). Kompilator posługuje się tą pamięcią jako domyślnym obszarem – nie musimy stosować żadnych dodatkowych klasyfikatorów (oprócz wynikających z zastosowanej sekcji – jak *NOINIT*, czy wynikających ze struktury programu – jak *static* czy *volatile*). Także zawartość wskaźników domyślnie wskazuje na adresy w pamięci danych (z wyjątkiem wskaźników na funkcje odnoszących się do adresów w obszarze kodu). Niektóre układy z rodziny ATmega są wyposażone w sprzętowy interfejs zewnętrznej pamięci danych. Dostęp do zewnętrznej pamięci jest zorganizowany bardzo prosto: odwołanie się programu do adresu RAM wykraczającego poza pojemność wbudowaną w kostkę powoduje samoczynne wygenerowanie odpowiedniej sekwencji sygnałów na magistrali *ext–ram*. Nie są więc potrzebne (w przeciwieństwie do np. '51) żadne oddzielne instrukcje maszynowe. Wynika stąd od razu, że AVR–GCC może taką pamięć obsługiwać na zwykłych zasadach, bez żadnych dodatkowych adaptacji. Jeśli jednak przypomnimy omawiany wcześniej schemat domyślnego podziału RAM na sekcje (*.data*, *.bss*, *.noinit*, stos) od razu stwierdzimy, że obszar zewnętrzny jest ulokowany poza stosem i nie będzie mógł być bezkolizyjnie wykorzystany przez linker. Mamy do dyspozycji kilka sposobów uniknięcia tej przeszkody. Jeden z najczęściej stosowanych to poinstruowanie linkera o nowym, pasującym do przygotowanego sprzętu rozmieszczeniu sekcji. Służą do tego odpowiednie opcje linii komend AVR–GCC:

– Wl,––section–start=.data=adres_startowy
– Wl,––section–start=.bss=adres_startowy
– Wl,—section–start=.noinit=adres_startowy

albo (dla sekcji *.data* i *.bss*) wersje skrócone:

– Wl,–Tdata=adres_startowy
– Wl,–Tbss=adres_startowy

W AvrSide wprowadzamy je w polu edycyjnym dodatkowych opcji w zakładce *Linker* okna konfiguracji projektu – ale bez prefiksu –Wl, – AvrSide dodaje go automatycznie (czyli np. wpisujemy po prostu: *–Tdata=adres*). Jeśli używamy *makefile* dodajemy potrzebną opcję do LDFLAGS. Należy tylko pamiętać, że przesunięcie sekcji przemieszcza samoczynnie wszystkie znajdujące się za nią z zachowaniem domyślnej kolejności (**rys. 32**). Korzyść z takiego przemieszczenia jest podwójna: otrzymujemy do dyspozycji dużą przestrzeń na dane a zarazem nie musimy się kłopotać o wielkość stosu i potencjalne nadpisanie przez stos części danych. Możliwa jest także oczywiście operacja odwrotna: przeniesienie do zewnętrznej pamięci stosu (służy do tego dyrektywa kompilatora *–minit–stack=nnnn* albo zamiennie bezpośrednie zadeklarowanie początku stosu dyrektywą linkera *–defsym, __stack=nnnn*). Ale w praktyce nie ma to większego sensu, gdyż dostęp do stosu poprzez zewnętrzną magistralę będzie znacznie wolniejszy, co pogorszy efektywność całego programu.

Przy takim przemieszczeniu sekcji danych musimy samodzielnie zadbać o odpowiednio wczesne uruchomienie magistrali *ext––ram* – musi być ona aktywna już w momencie inicjalizacji danych *.data* oraz zerowania obszaru *.bss*. Zwykłe wpisanie konfiguracji magistrali na początku naszego programu nie wystarczy, gdyż (jak widzieliśmy w podglądzie assemblera) kod inicjalizacji (*__do_copy_data* oraz *__do_clear_bss*) wykonywany jest przed wejściem do funkcji *main*. Musimy zmodyfikować jedną z sekcji startowych (podział samoczynnie tworzonych sekwencji rozpoczynających i kończących program na sekcje startowe *init0...init9* i zamykające *fini0...fini9* jest dokładnie opisany w dokumentacji avr–libc) – na ogół wykorzystujemy przeznaczoną dla użytkownika sekcję *init1*. Umieszczenie funkcji w określonej sekcji realizuje atrybut *section(nazwa)*. Dodatkowo zastosujemy znany juz atrybut *naked*, który pozbawi funkcję samoczynnie tworzonego prologu i epilogu. W efekcie uzyskujemy bezpośrednie wstawienie kodu w potrzebnym miejscu, np. tak (przykład dla Atmega 128):

```
void EnableExtRam(void) __attribu-
te__ ((naked)) __attribute__ ((section
(„.init1”)));

void EnableExtRam(void)
{
  XMCRA = _BV(SRW00);
  XMCRB = _BV(XMBK);
  MCUCR = _BV(SRE);
}
```

Sprawdźmy, że rzeczywiście odpowiedni kod pojawił się zaraz za wektorami przerwań. Zwróćmy też uwagę, że w tym przypadku wystarcza sama deklaracja i definicja funkcji – nigdzie w programie jej jawnie nie wywołujemy – byłoby to wręcz błędem, gdyż spowoduje skok z powrotem do sekcji *.init1* praktycznie resetując program. Jeśli zrobimy kilka eksperymentów stwierdzimy także, że w sekcji *.init1* można spowodować błąd jawnym przypisaniem zera (np. *XMCRA=0*). Kompilator użyje rejestru zerowego *r1* (*sts 0x006D, r1)*, który tutaj może mieć jeszcze wartość przypadkową gdyż jest inicjalizowany dopiero w ustawiającej stos sekcji. *init2*. W razie konieczności przenieśmy więc nasz kod do następnej startowej sekcji użytkownika *.init3*.

Można też zamiennie dołączyć do projektu mały moduł assemblerowy, który realizuje tylko to zadanie:

```
// uruchomienie ext-ram:
#include <avr/io.h>

.global MemInit
 .section.init1,"ax",@progbits

MemInit:
  ldi r24,0x80
  out _SFR_IO_ADDR(MCUCR),r24
  ldi r24,_BV(SRW00)
  sts XMCRA,r24
  ldi r24,_BV(XMBK)
  sts XMCRB,r24
```

ale tu już musimy sami pamiętać o rozmieszczeniu SFR w przestrzeniach *IO* oraz *extended–sfr* i stosować odpowiednie instrukcje (*out* albo *sts*).

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.