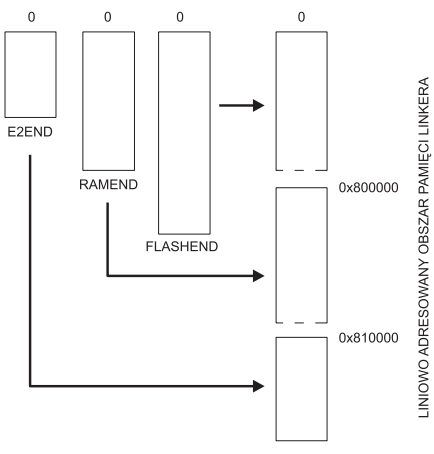
AVR–GCC: kompilator C dla mikrokontrolerów AVR, część 14

Obsługa obszarów pamięci mikrokontrolerów AVR, część 2

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.

Podczas konfigurowania pamięci danych ponownie pojawia się wspomniany wcześniej „dziwny" offset 0x800000 dodawany do adresów RAM. Zawsze należy go uwzględniać przy wpisywaniu nowej lokalizacji sekcji danych (czyli przenosząc *.data* do pamięci zewnętrznej jak w powyższym opisie, np. dla ATmegi 128 wpiszemy *–Tdata=0x801100,* a nie *–Tdata=0x1100* jak intuicyjnie wynikałoby z rozmiaru wewnętrznej pamięci kostki). W taki właśnie sposób – poprzez rzutowanie wydzielonych obszarów pamięci AVR na własny wspólny liniowy obszar pamięci – linker AVR–GCC radzi sobie z architekturą typu *Harvard* (**rys. 33**). Duża wartość offsetów zapewnia, że nawet dla największych dostępnych pamięci nie wystąpi nałożenie się obszarów. Z tak zapisanej (w pliku *elf*) zawartości pamięci narzędzie *avr–objcopy* ekstrahuje następnie potrzebne fragmenty do wynikowych plików *hex*. Stąd też wynika dość rozbudowana postać wywołania dla pliku zawartości eeprom, np:



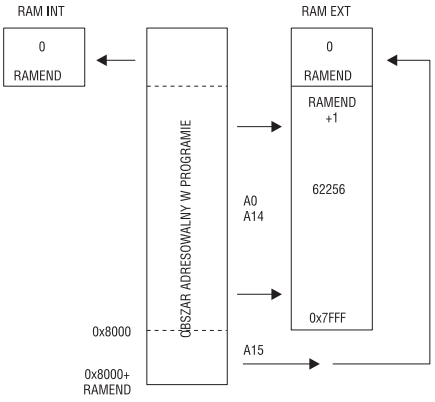
Rys. 33. Rzutowanie pamięci AVR na liniowy obszar pamięci linkera AVR–GCC

```
avr–objcopy.exe –j.eeprom –set–
–section–flags=.eeprom="alloc,load" —change–section–lma.eeprom=0
–O ihex t0_test.elf t0_test.eeh
```

które musi z powrotem przesunąć początkowy adres do pozycji 0.

Przy dokładaniu w układzie zewnętrznej pamięci *SRAM* należy mieć na uwadze, że (ze względu na wspominany już mechanizm sprzętowej obsługi magistrali) komórek od zera do *RAMEND* nie da się zaadresować bezpośrednio (te adresy dotyczą zasobów wewnętrznych mikrokontrolera). Jest to bardzo proste do ominięcia w razie stosowania kostki 62256 (32 kB). Adresy powyżej 32 kB (od 0x8000) mają ustawioną linię A15, której ta kostka nie używa – fizycznie więc adresowany jest obszar od zera. W rezultacie mamy do dyspozycji przestrzeń adresową od zera do *0x8000 + RAMEND* (**rys. 34**), natomiast linia A15 może pozostać wyłączona (odpowiedni pin jest wykorzystywany jako zwykłe we/wy).

Przy zewnętrznej pamięci o maksymalnej pojemności 64 kB sprawa nie jest już taka prosta. Zgodnie z opisami dokumentacyjnymi Atmela można to osiągnąć ustawiając adres >= 0x8000 (jak powyżej) i przełączając linię A15 programowo. Dolny obszar (*0 – RAMEND*) zewnętrznej



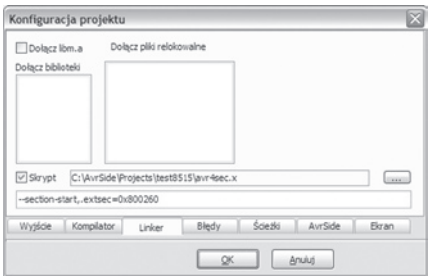
Rys. 34. Obszar adresowania 0x8000...(0x8000+RAMEND) fizycznie odpowiada obszarowi 0...RAMEND dodatkowego układu 62256

kostki pozostaje jednak niedostępny dla linkera i można go obsłużyć tylko bezpośrednio poprzez wskaźniki.

Po odpowiednim skonfigurowaniu nie musimy już w programie pamiętać o rozmieszczeniu poszczególnych obszarów – linker sam zadba o właściwe adresowanie używanych zmiennych. Inaczej wygląda sprawa gdy używamy magistrali do komunikacji z zewnętrznym urządzeniem, którego rejestry znajdują się pod konkretnymi – zależnymi od połączeń i systemu dekodowania – adresami. Mamy w tym celu do dyspozycji dwie różne metody. Pierwsza to zastosowanie standardowych operacji na wskaźnikach. 16–bitowy adres jest traktowany jako wskaźnik na 8–bitową komórkę pamięci (*char* * lub *unsigned char* *) – a dostęp do tej komórki jest realizowany jako odwołanie do obiektu wskazywanego. Stosujemy typowy zapis C:

```
*((volatile unsigned char *) adres_komorki)
```

któremu dyrektywą *#define* możemy dla wygody nadać czytelną nazwę zgodną z przeznaczeniem komórki. Sprawdźmy, że to rzeczywiście działa, np. tak (ATmega 8515, RAMEND=0x260):



Rys. 35. Wprowadzenie zmodyfikowanego skryptu linkera oraz adresu startowego dodatkowej sekcji RAM

```
#define EXT_MEM_CELL(X) *((volatile
unsigned char*)X)

EXT_MEM_CELL(0x400) = 0xaa;
 136: 8a ea      ldi   r24, 0xAA  ; 170
 138: 80 93 00 04    sts   0x0400, r24
```

Klasyfikator *volatile* jest w przypadku obsługi zewnętrznych urządzeń szczególnie istotny – często mamy do czynienia z sekwencyjnym zapisem lub odczytem danych lub nastaw konfiguracyjnych – bez *volatile* optymalizator może nam wiele z tych operacji całkiem pominąć czego skutkiem będzie błędne (lub brak) działanie układu. Taki bezpośredni dostęp przez wskaźniki został zastosowany w testowym układzie ATmega 8515 + 62256 dla sprawdzenia poprawności podłączenia i działania pamięci:

```
bool CheckRam(void)
{
 bool chkcell;
 chkcell=true;

 uint i,k;
 k=(uint)RAMEND + 0x8000;

 for (i=RAMEND+1;i<=k;i++)
 {
  *((uchar*)i)=0xa5;
 }

 for (i=RAMEND+1;i<=k;i++)
   {
   if (*((uchar*)i) != 0xa5)
     {
     chkcell = false;
     break;
     }
   }

 for (i=RAMEND+1;i<=k;i++)
 {
  *((uchar*)i)=0x5a;
 }

 for (i=RAMEND+1;i<=k;i++)
   {
   if (*((uchar*)i) != 0x5a)
     {
     chkcell = false;
     break;
     }
   }
 return chkcell;
}
```

Potencjalne zagrożenie stwarza przy takim bezpośrednim dostępie fakt, że linker nic nie „wie" o niezależnym wykorzystaniu przez nas niektórych adresów pamięci i w związku z tym może je przydzielić zmiennym. Jeśli zachodzi potrzeba musimy więc sami odpowiednio poprzesuwać sekcje pamięci, aby zapobiec takiej kolizji. Mankament ten jest w znacznej mierze wyeliminowany jeśli nakażemy linkerowi utworzenie dodatkowej sekcji w pamięci RAM i ulokowanie w niej zmiennych wyposażonych w odpowiedni atrybut.

Sprawdźmy szybko na małym przykładzie jak to działa: ulokujmy na początku zewnętrznej pamięci ATmega 8515 (od adresu *0x260*) sekcję *.extsec* i skierujmy tam obsługę czterech rejestrów. W tym celu do dyrektyw linkera dodamy wpis *–section–start,.extsec=0x800260* i zadeklarujemy odpowiednie zmienne z atrybutem przynależności do sekcji *.extsec*. Należy mieć jednak na uwadze, że chociaż zazwyczaj linker nadaje adresy w kolejności zgodnej z uszeregowaniem definicji w kodzie, to generalnie wcale nie jest to zagwarantowane. Dlatego zamiast 4 niezależnych zmiennych *char* użyjemy „opakowującej" je struktury:

```
volatile struct
   {
    char rejestr1;
    char rejestr2;
    char rejestr3;
    char rejestr4;
   } ExtStruct __attribute__((sec-
tion(„.extsec")));
```

Po skompilowaniu i wczytaniu do AvrStudio sprawdzamy, że *ExtStruct* jest rzeczywiście ulokowana pod adresem *0x260,* a odwołania do pól, np. *ExtStruct.rejestr2 = 0xaa ;* powodują zmianę we właściwym miejscu pamięci (w praktyce nie spotkała mnie ze strony AVR–GCC niespodzianka w postaci zamiany kolejności pól struktury w pamięci, jednak dla ostrożności można zamiast struktury zastosować po prostu tablicę – tu kolejność elementów jest już całkowicie jednoznaczna).

Pomimo przekazania kontroli linkerowi musimy jednak wstępnie zadbać, aby nie nastąpiła kolizja nowej sekcji z sekcjami tworzonymi automatycznie. Na ogół bierzemy też wtedy pod uwagę oferowany przez AVR sprzętowy rozdział zewnętrznej przestrzeni adresowej na dolną i górną z możliwością ustawienia różnych czasów dostępu – co pozwala na podłączenie jednocześnie szybszych oraz wolniejszych układów peryferyjnych.

Drugim – zamiennym – sposobem przekazania linkerowi instrukcji o dodatkowej sekcji jest modyfikacja skryptu. Na przykład dla atmega 8515 (architektura 4) skopiujemy sobie odpowiedni skrypt *\folder_kompilatora\avr\lib\ldscripts\avr4.x* jako *avr4sec.x* do subfoldera projektu i dopiszemy w nim:

– informację o nowym obszarze pamięci i jego adresie startowym:

```
MEMORY
{
 text   (rx) : ORIGIN = 0, LENGTH =
8K
 data   (rw!x): ORIGIN = 0x800060,
LENGTH = 0xffa0
 eeprom (rw!x): ORIGIN = 0x810000,
LENGTH = 64K
 page_2 (rw!x): ORIGIN = 0x800300,
LENGTH = 4K
}
```

– informację o nowej sekcji (w dziale *SECTIONS*):

```
.eeprom:
 {
  *(.eeprom*)
   __eeprom_end =. ;
 } > eeprom

.page2:
 { *(.page2) } > page_2
```

Następnie w wywołaniu linkera opcją *–Tścieżka_skryptu* wskazujemy zmodyfikowany skrypt. AvrSide oferuje w tym celu wsparcie – korzystanie z oddzielnego skryptu i jego pełną nazwę ustawiamy na zakładce *Linker* dialogu konfiguracji projektu (**rys. 35**). Po skompilowaniu projektu sprawdźmy, że odwołanie do zmiennej należącej do nowej sekcji, np:

```
volatile int Page2 __attribute__((sec-
tion(„.page2")));
.....................
Page2=0x55;

trafia pod zadeklarowany przez nas
w skrypcie adres:
 13c: 85 e5      ldi   r24, 0x55  ; 85
 13e: 90 e0      ldi   r25, 0x00  ; 0
 140: 90 93 01 03    sts   0x0301, r25
 144: 80 93 00 03    sts   0x0300, r24
```

Jednak po bliższym przyjrzeniu się rezultatom naszych poczynań stwierdzimy, że niezbędne będą pewne poprawki. Otóż zawartość tak utworzonych sekcji (wartości początkowe – także zerowe – zmiennych przypisanych do sekcji) jest dołączana do pliku wynikowego (*hex*) programu. Pamiętamy, że w przypadku domyślnej sekcji *.data* jest to zamierzone i pozwala na stosowanie zmiennych inicjalizowanych. Natomiast dla dodatkowych sekcji RAM nie jest już tak idealnie.

Po pierwsze: mechanizm samoczynnej inicjalizacji (i zerowania) nie będzie (bez modyfikacji znacznie głębszych niż nasza) działać dla sekcji dodatkowych (chociaż kompilator nie zgłosi żadnego błędu). Musimy więc samodzielnie inicjalizować każdą zmienną (także wartością zerową). Akurat w przypadku komunikacji z urządzeniem zewnętrznym nie jest to żadną wadą, a staje się wręcz zaletą: lepiej unikać wszelkich samoczynnych zapisów do urządzenia gdyż może to przynieść niespodziewane rezultaty. Taka sama korzyść wystąpi przy obsłudze pamięci nieulot-

nych (NVRAM, FRAM) gdyż wszelka inicjalizacja zniszczyłaby ich poprzednią zawartość.

Po drugie (gorsze): wpis do pliku *hex* zachowuje adresy z przesunięciem 0x800000, czyli poza zakresem wszelkiej pamięci *flash*. Może to prowadzić do zakłócenia działania nie przygotowanych na taką ewentualność programatorów i w konsekwencji do niemożności zaprogramowania kostki posiadanym sprzętem.

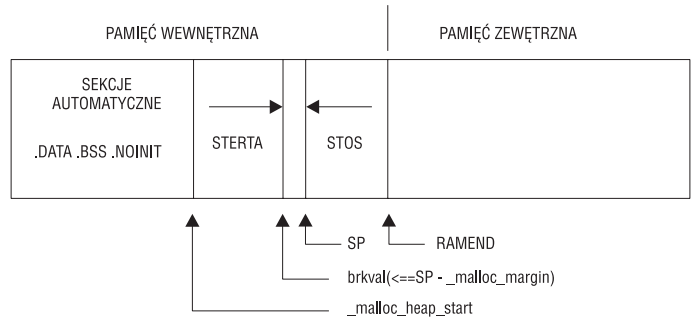
Z powyższego wynika natychmiast, że nasze dodatkowe sekcje muszą być koniecznie wyeliminowane z pliku wynikowego. Realizuje się to bardzo prosto poprzez modyfikację wywołania *avr–objcopy* konwertującego wybrane elementy pliku obiektowego *elf* do pliku *hex*. Sprawa staje się jednak utrudniona jeśli z poziomu używanego IDE nie mamy dostępu do zmiany potrzebnych opcji. Niestety także AvrSide nie jest obecnie wyposażone w żaden mechanizm zarządzania sekcjami pamięci i brak możliwości zmiany domyślnej postaci linii komendy dla *avr–objcopy*.

Na szczęście jest inny sposób rozwiązania problemu: poinstruowanie linkera, aby dodatkowych sekcji w ogóle nie umieszczał w pliku obiektowym *elf*. Jeśli stosujemy oddzielny skrypt wystarczy wyposażyć opis sekcji w atrybut NOLOAD (w podanym powyżej przykładzie będzie to *.page2 (NOLOAD):*). Wykonanie zadania z poziomu opcji wywołań linkera jest nieco bardziej skomplikowane:

– nazwę sekcji rozpoczynamy od frazy *.noinit*, w naszym przykładzie może to byc np. *.noinit_extsec*, jest to równoznaczne z ustawieniem atrybutu NOLOAD
– wywołanie linkera uzupełniamy opcją *–unique=”noinit_extsec”* nakazującą utworzenie całkiem oddzielnej sekcji, bez tego nasza *.noinit_extsec* zostanie domyślnie dołączona (bez zwracania uwagi na adres startowy) do podstawowej sekcji. *noinit* (linia dodatkowych opcji na rys. 35 będzie więc w końcu wyglądać tak: *–section–start,.noinit_extsec=0x800260, –unique=”.noinit_extsec”)*.

Teraz dodatkowe sekcje skonfigurowane są już całkowicie zgodnie z oczekiwaniami.

Wszystkie powyżej omówione metody dostępu do RAM zakładają przydzielenie na stałe adresów zmiennych już na etapie linkowania. Avr–gcc daje nam do dyspozycji dodatkowo możliwość dynamicznego wykorzystania pamięci.



Rys. 36. Lokalizacja sterty w wewnętrznej pamięci RAM

Funkcja *void* malloc (size_t __size)* (moduł *stdlib*) zwraca nam wskaźnik na przydzielony obszar pamięci o rozmiarze *__size* bajtów (ewentualnie *NULL* jeśli operacja się nie powiedzie). Obszar nie jest inicjalizowany (jego zawartość pozostaje przypadkowa).

Funkcja *void* realloc (void* ptr, size_t __size)* zmienia rozmiar przydzielonego pod adresem *ptr* obszaru na nową wielkość *__size* (w razie konieczności przeniesienia całego obszaru w inne wolne miejsce zwraca nowy wskaźnik).

Funkcja *void* calloc (size_t __nele, size_t __size)* przydziela obszar na *__nele* elementów o rozmiarze *__size* (czyli *__nele * __size* bajtów). Przy tym zawartość obszaru zostaje wyzerowana.

Funkcja *void free (void* ptr)* zwalnia przydzielony pod adresem *ptr* obszar i umożliwia jego ponowne wykorzystanie.

Domyślnie na obszar dynamicznej alokacji pamięci (czyli stertę, *heap*) przeznaczona jest przestrzeń pomiędzy omawianymi wcześniej automatycznymi sekcjami, a stosem (**rys. 36** – zaczerpnięty z dokumentacji *avr–libc*).

Podczas każdej nowej alokacji sprawdzany jest bieżący wskaźnik stosu. Po zmniejszeniu o margines bezpieczeństwa (*__malloc_margin*, domyślnie 32 bajty) stanowi ograniczenie dla wielkości alokowanego obszaru (koniec sterty jest opisany wewnętrzną zmienną *brkval*). Zabezpiecza to przed nałożeniem się sterty i stosu w trakcie dalszego działania programu. Oczywiście może się zdarzyć, że przy wielokrotnych zagnieżdżeniach lub dużej ilości zmiennych lokalnych w funkcjach stos rozszerzy się bardziej niż zakładaliśmy, jednak zmienna *__malloc_margin* jest udostępniona jako globalna i możemy ją skorygować według potrzeb.

W tak ciasnym środowisku stosowanie dynamicznej alokacji jest raczej problematyczne (co zresztą podkreślają sami autorzy avr–libc). Nie zwiększy nam ona w cudowny sposób brakującej pamięci RAM.

Jednak stertę – podobnie jak wszelkie inne sekcje – możemy przenieść do pamięci zewnętrznej. W tym celu odpowiednio definiujemy symbole *__heap_start* oraz *__heap_end* w opcjach wywołania linkera, albo zamiennie (co jest chyba trochę wygodniejsze) samodzielnie inicjalizujemy w programie zmienne *__malloc_heap_start* i *__malloc_heap_end*. Pamiętajmy przy tym, że sama zmiana początku sterty nie wystarcza, jej koniec także musi zostać ustawiony (domyślna wartość zero zmiennej *__malloc_heap_end* jest interpretowana jako położenie sterty poniżej stosu co doprowadzi do sprzeczności).

W przykładowym programie wygląda to np. tak:

– inicjalizacja sterty:

```
#define HEAP_START 0x4000
#define HEAP_END 0x8000+RAMEND

void InitHeap(void)
{
  __malloc_heap_start=(char*)HEAP_START;
  __malloc_heap_end=(char*)HEAP_END;
}
```

– zaalokowanie obszaru 10000 bajtów i jego wyzerowanie:

```
volatile char *Ptab100;
Ptab100 = calloc(10000,1);
```

– i zapisy do różnych miejsc alokacji:

```
*(Ptab100 + 99) = 20;
*(Ptab100 + 102) = 30;
```

Operacje te łatwo prześledzimy w oknie podglądu pamięci AvrStudio.

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.