AVR–GCC: kompilator C mikrokontrolerów AVR, część 5

W tej części kursu skupiamy się na omówieniu zakresu zmiennych, budowie i funkcjach plików nagłówkowych, przybliżając w ten sposób kolejne tajniki kapryśnego – jak głosi nośna opinia – kompilatora.

Zakres zmiennych

W zależności od miejsca oraz sposobu zadeklarowania zmiennych mogą mieć one w naszym projekcie różny zasięg – tzn. możemy z nich korzystać w jednym pliku źródłowym (module), w wielu plikach albo tylko wewnątrz kodu funkcji. Mówimy w takim przypadku o zmiennych globalnych oraz lokalnych. Podział ten nie ma wpływu na typ zmiennej ale jest istotny w trakcie pisania programu, inny jest też sposób obsługiwania zmiennych lokalnych przez kompilator.

Do tej pory ograniczaliśmy się do zmiennych globalnych (zasięg globalny jest domyślny) deklarowanych i używanych w pojedynczym pliku (module) źródłowym projektu. Utwórzmy teraz następny przykładowy projekt zawierający kilka modułów: *main.c*, *funkcje.c* oraz *dane.h* – zapiszmy go w subfolderze *\Projects\Kurs\Przyklad–03* jako *Test03*. Dodawanie plików do projektu jest w AvrSide bardzo proste: wykonujemy komendę menu *Projekt>Dodaj pustą stronę* (dostępna także w menu kontekstowym projektu wywoływanym skrótem **CTRL+.**) i zapisujemy nową zakładkę *NoName* jako odpowiedni typ pliku (c, s, h) z wybraną nazwą (typ pliku źródłowego wybieramy z listy – rozszerzenie będzie dodane automatycznie więc nie musimy go dopisywać). Jednak najpierw musimy wpisać do modułu jakiś kod (może to byc na wstępie sam komentarz) gdyż AvrSide blokuje zapis pliku pustego. W pliku *main.c* wstawimy jak zwykle szablon modułu głównego natomiast w pliku *dane.h* – szablon "nagłówek danych projektu" (*headdat*).

Szablon danych został przygotowany tak aby bez wielokrotnego przepisywania deklaracji można było używać w całym projekcie wspólnych globalnych zmiennych, funkcji oraz definicji:

```
// plik nagłówkowy globalnych danych
projektu
#ifndef _PROJ_DAT_H_
#define _PROJ_DAT_H_
// #include:

// #define:

// definicje typów typedef

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

int test = 10;

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

extern int test;

#endif

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

extern int Myfunc(char x,char y);

#endif
```

Wstawiamy tutaj wspólne dla wszystkich modułów projektu pliki nagłówkowe (np. *#include <avr/io.h>*), definicje konfiguracji i podłączeń sprzętowych (np. *#define LED PB2*), własne definicje typów (np. *typedef unsigned char uchar*). Po dołączeniu naszego nagłówka do dowolnego modułu (*#include „dane.h"*) mamy od razu w module dostęp do wszystkich tych ustawień.

Trochę więcej komplikacji jest z globalnymi zmiennymi. Zwykłe ich zadeklarowanie spowoduje wprawdzie, że będą widoczne w projekcie i nie zostanie zgłoszony błąd na etapie kompilacji poszczególnych modułów ale nie da sobie z tym rady konsolidator sygnalizując błąd wielokrotnej definicji. Możemy to od razu sprawdzić dopisując *int test=10;* w obu naszych plikach źródłowych c (*main* i *funkcje*): kompilacja (**CTRL+F9**) przebiegnie sprawnie ale projektu nie da się zakończyć (**F9** – błąd linkera – *"multiple definition of test"*).

Z pomocą przychodzi kompilacja warunkowa: w pliku głównym ze zdefiniowanym makrem _MAIN_MOD_ preprocesor wstawi pełną definicję *int test=10;*natomiast w pozostałych plikach tylko informację dla kompilatora, że zmienna *test* już gdzieś w projekcie istnieje (*extern*) i można z niej bezpiecznie korzystać.

Nowsze wersje *avr–gcc* pozwalają na pominięcie tego sposobu w przypadku zmiennych automatycznie zerowanych (sekcja *bss*) – taka zmienna (np. *int test;*) jest samoczynnie bez dodatkowych zabiegów traktowana jako pojedyncza pomimo wielokrotnego zdefiniowania i zostaje jej przydzielony jeden wspólny obszar w SRAM.

W przypadku funkcji można bez błędu użyć we wszystkich modułach deklaracji *extern* – w ten sposób funkcja (którą dokładnie zdefiniujemy tylko w jednym dowolnie wybranym module) będzie widoczna i możliwa do użycia w całym projekcie. Zróbmy to zaraz definiując w pliku *funkcje.c* funkcję zadeklarowaną w *dane.h* jako *extern int Myfunc (char x, char y);* (funkcja o dwóch argumentach typu *char*, zwracająca rezultat typu *int*) (nie zapomnijmy oczywiście o dołączeniu do obu źródeł nagłowka z danymi: *#include „dane.h"*):

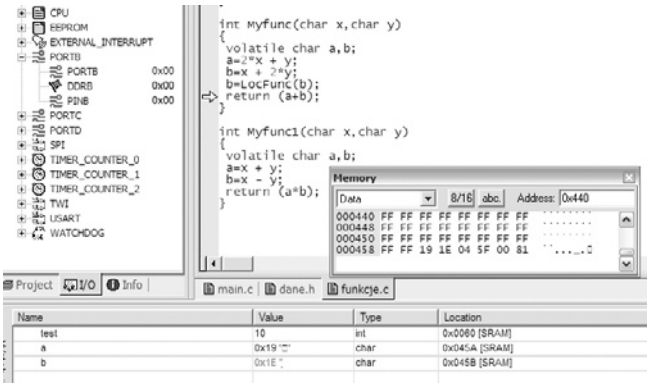
```
int Myfunc(char x,char y)
{
 char a,b;

 a=2*x + y;
 b=x + 2*y;
 return (a+b);
}
```

Teraz w pliku głównym *main.c* możemy już bez problemu posłużyć się tą funkcją:

```
test = Myfunc(10,5);
```

W funkcji celowo wprowadziłem zmienne lokalne *a*, *b* (chociaż nie są dla wykonania obliczeń konieczne) aby przedstawić sposób ich obsługi przez kompilator. Takie zmienne – definiowane wewnątrz ciała funkcji (zwane też zmiennymi automatycznymi) są dostępne i możliwe do wykorzystywania tylko i wyłącznie w obrębie tego ciała funcji. Próba odwołania do nich spoza funkcji powoduje błąd.



Rys. 14. Podgląd zmiennych lokalnych na stosie

Zmienne te istnieją tylko w czasie wykonywania funkcji – po wywołaniu funcji, w prologu, są tworzone albo na stosie albo (jeśli optymalizator stwierdzi, że ma chwilowo do dyspozycji odpowiednią liczbę rejestrów) w obszarze rejestrów roboczych. Po zakończeniu działania funkcji po prostu przestają istnieć – pamięć dla nich przydzielona zostaje przeznaczona na inne bieżące cele.

Zobaczmy, jak przedstawi nam to w działaniu AvrStudio. Po omawianym już wstępnym skonfigurowaniu sesji AvrStudio wstawmy do okienka podglądu zmiennych wszystkie użyte zmienne: *test*, *a*, *b*.

Test po zerowaniu przyjmuje wartość 10, natomiast *a* i *b* są określone jako *„not in scope”* (poza zakresem),czyli wszystko zgodnie z oczekiwaniami. Przejdźmy teraz krokami (**F11**) do wnętrza funkcji, spotka nas niestety niespodzianka: zmienne *a* i *b* nadal nie są obsługiwane (*„location not valid”* – AvrStudio ma kłopot z ich umiejscowieniem w pamięci). Przyczyną jest wspomniane powyżej skuteczne działanie optymalizatora. W kodzie asemblera znajdujemy:

```
int Myfunc(char x,char y)
{
 5c:  28 2f     movr18, r24
 5e:  86 2f     movr24, r22
 char a,b;

 a=2*x + y;
 60:  92 2f     movr25, r18
 62:  99 0f     addr25, r25
 64:  96 0f     addr25, r22
 b=x + 2*y;
 66:  88 0f     addr24, r24
 68:  82 0f     addr24, r18
 return (a+b);
 6a:  29 2f     movr18, r25
 6c:  33 27     eorr19, r19
 6e:  27 fd     sbrc  r18, 7
 70:  30 95     comr19
 72:  99 27     eorr25, r25
 74:  87 fd     sbrc  r24, 7
 76:  90 95     comr25
 78:  82 0f     addr24, r18
 7a:  93 1f     adcr25, r19
 7c:  08 95     ret
}
```

Optymalizator wykonał wszystkie potrzebne działania w obszarze rejestrów w sposób na tyle zwięzły, że nie zaszła potrzeba wyraźnego wyodrębniania zmiennych lokalnych. Jest to bardzo pozytywny rezultat jednak dla potrzeb naszego testu wyłączmy na chwilę optymalizację (odpowiada to opcji *–O0* kompilatora). Teraz widzimy (pamiętajmy o użyciu komendy *Build* a nie *Make* po zmianie opcji), że zmienne *a* oraz *b* są z chwilą wejścia programu do funcji tradycyjnie tworzone tymczasowo na stosie (w moim przykładzie pod adresami *0x045A* i *0x045B*) i niszczone po zakończeniu funkcji. Jednak od razu zauważymy też znaczący przyrost objętości kodu. Możemy przy okazji porównać generowane kody assemblera i obejrzeć ile pożytecznej pracy wykonuje optymalizator. Nic dziwnego, że często symulacja w AvrStudio „nie zgadza się” z naszym zapisem źródłowym: nie wykorzystywane zmienne mogą byc usunięte, niektóre linie kodu są eliminowane itd. Ingerencja optymalizatora może być na tyle duża, że ten sam program ze zmienionym poziomem optymalizacji czasem zaczyna zachowywać się nieco inaczej. Dlatego chwilowe przełączanie poziomów optymalizacji tylko po to aby lepiej obejrzeć wynik w symulatorze (tak jak to przed chwilą zrobiliśmy w celach edukacyjnych) jest generalnie kiepskim pomysłem (nie ma niestety możliwości selektywnego ustawiania różnych poziomów optymalizacji dla poszczególnych fragmentów kodu).

W praktyce zamiast rezygnować z zalet optymalizacji lepiej jest kontrolować istotne dla nas zmienne przy pomocy używanego już słowa kluczowego *volatile*. Informuje ono kompilator, żeby tak opisanej zmiennej nie poddawać jakimkolwiek działaniom optymalizującym i upraszczającym i wykonywać na niej wszystkie operacje przewidziane w kodzie (chociaż z punktu widzenia optymalizatora mogą one wyglądać na zbędne). Główne zastosowanie tego mechanizmu to zabezpieczanie zmiennych używanych w przerwaniach (to wynika bezpośrednio z nazwy: *volatile* – czyli ulotny, nietrwały – oznacza, że wartość zmiennej może być w każdej chwili uaktualniona przez czynnik zewnętrzny – przerwanie – i nie można w związku z tym pominąć żadnej związanej z nią operacji w głównej pętli programu), jednak często jest pomocny także w różnych innych sytuacjach. Sprawdźmy zaraz, że zmiana deklaracji na *volatile char a,b;* (przy ponownym włączeniu maksymalnej optymalizacji) daje ten sam efekt: zmienne wędrują z obszaru rejestrów na stos. Jest to pokazane na **rys. 14**.

Zobaczmy jeszcze, że takie same nazwy zmiennych mogą być z powodzeniem użyte w innej funkcji – w tym celu definiujemy sobie dodatkowo:

```
int Myfunc1(char x,char y)
{
 volatile char a,b;

 a=x + y;
 b=x - y;
 return (a*b);
}
```

i oglądamy jak traktowane są zmienne *a* oraz *b* przy wywołaniach kolejno *Myfunc* oraz *Myfunc1* (dobrze jest w tym celu dodatkowo włączyć w AvrStudio okienko podglądu pamięci danych jak na rys. 14). Przekonamy się, że wartości chwilowe *a* i *b* zmieniają się w zależności od tego, która funkcja aktualnie z nich korzysta.

Może nas w pierwszej chwili zdziwić fakt, że w momencie wejścia do funkcji *Myfunc1* *a* oraz *b* zachowały wartości przypisane wewnątrz poprzedniej funcji (*Myfunc*) – przecież miały stracić ważność. Przyczyną jest prostota naszego przykładu. Kompilator nie niszczy zmiennych lokalnych (np. przez wyzerowanie) ale po prostu przestaje się nimi „przejmować”. Gdyby pomiędzy wywołaniami *Myfunc* i *Myfunc1* pojawiły się jakieś operacje wykorzystujące stos – *a* i *b* zostałyby nadpisane. Ponieważ jednak nic takiego nie zachodzi wartości wstawione pod adresy 0x45a i 0x45b pozostały nie zmienione.

Możliwość użycia takich samych nazw zmiennych lub funkcji jest też czasem korzystna w odniesieniu do poszczególnych modułów kodu źródłowego. W C uzyskujemy to poprzez ograniczenie zakresu ważności zmiennej (funkcji) do pojedynczego modułu – sprawia to słowo kluczowe *static* .

Zadeklarujmy sobie takie lokalne symbole: w module *main.c* dopiszemy na przykład:

```
// deklaracja zmiennej lokalnej dla
```

Rys. 15. Tablica symboli pokazuje tylko symbole globalne

```
modułu main
static char k=1;
// funkcje:
static char LocFunc(char Value);
// deklaracja funkcji lokalnej dla mo-
dułu main
// oraz definicja tej funkcji
char LocFunc(char Value)
{
 return Value + 2;
}
```

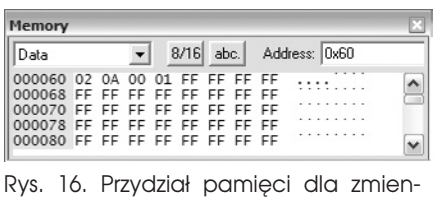
a w module *funkcje.c*:

```
// deklaracja zmiennej lokalnej dla mo-
dułu funkcje
static char k=2;
static char LocFunc(char Value);
// deklaracja funkcji lokalnej dla mo-
dułu funkcje
// oraz definicja tej funkcji
char LocFunc(char Value)
{
 return Value + 10 +k;
}
```

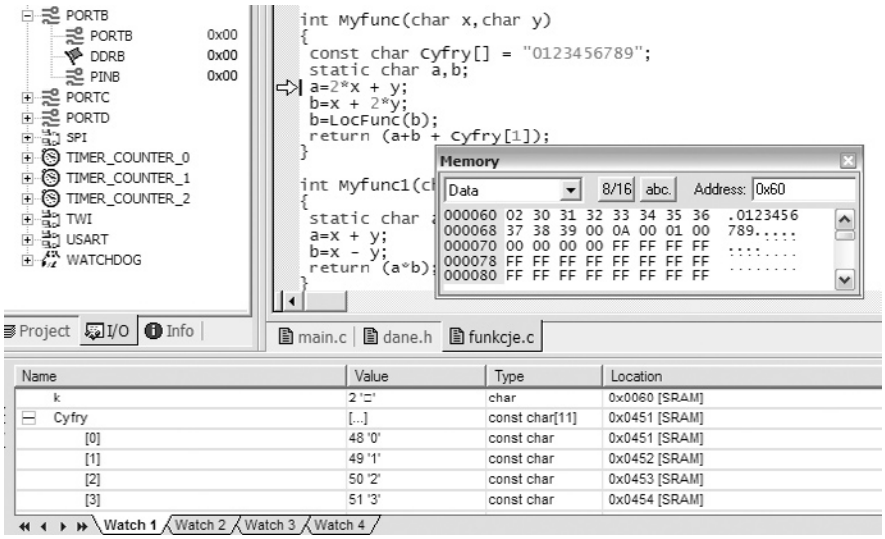
Przy kompilacji stwierdzamy, że w tym przypadku nie występuje błąd wielokrotnej definicji. Wiąże się z tym również ukrycie powyższych lokalnych nazw w oknie podglądu symboli konsolidatora (**rys. 15**), wyszczególnione są tylko symbole globalne (okno podglądu symboli wywołujemy klawiszem F8).

Oczywiście pomimo tego ukrycia zmienne *k* są fizycznie ulokowane w pamięci SRAM (pod adresami 0x60 oraz 0x63 na **rys. 16**), znajdziemy je też przeglądając plik symboli *Test03.smb*. Użycie poszczególnych adresów zależy od modułu, z którego się do naszej zmiennej *k* odwołujemy (kod modułu *main.c* korzysta z adresu 0x63, natomiast moduł *funkcje.c* używa 0x60). Jeśli zechcemy to prześledzić w AvrStudio zauważymy, że po wstawieniu do okienka podglądu zmiennej *k* będzie ona opisana wartością i adresem zależnym od modułu, do którego wchodzimy pracą krokową.

Podobnie jest z funkcjami – każdy moduł odwołuje się do swojej własnej lokalnej definicji *LocFunc*. Język C daje nam jeszcze jedną możliwość łączącą właściwości powyższych przypadków. Jeśli mianowicie użyjemy kwalifikatora *static* do zmiennej lokalnej deklarowanej wewnątrz ciała funkcji (automatycznej) uzyskamy następujacy efekt: zakres używania zmiennej pozostanie nadal ograniczony do ciała funkcji ale zarazem zmiennej zostaje przydzielona na stałe przestrzeń w obszarze danych SRAM. Po wyjściu z funkcji zmienna taka nie jest zatem - jak poprzednio - narażona na zniszczenie (nadpisanie) ale przechowuje ostatnio przypisaną wartość – aż do ponownego wywołania używającej ją funkcji. Wypróbujmy to zaraz przepisując nieco nasze poprzednie definicje:



Rys. 16. Przydział pamięci dla zmiennych lokalnych



Rys. 17. Zmienne lokalne funkcji w wersji inicjalizowanej

```
int Myfunc(char x,char y)
{
static char a,b;
 a=2*x + y;
 b=x + 2*y;
 return(a+b);
}

int Myfunc1(char x,char y)
{
 static char a,b;
 a=x + y;
 b=x - y;
 return(a*b);
}
```

Prowadząc krokowy debugging jak na rys. 14 zobaczymy teraz jak zmieniła się lokalizacja zmiennych *a* i *b*: mają one przydzielony obszar w sekcji *bss*. Opis *a* oraz *b* w okienku podglądu zmienia się w trakcie wchodzenia i opuszczania kolejnych funkcji. Zauważmy, że biorąc pod uwagę przydział pamięci zmienne te nie różnią się obecnie od zwykłych lokalnych czy nawet globalnych. Natomiast znacznie poprawia się czytelność kodu oraz jest redukowana możliwość błędów wynikających z powtórzenia nazw.

Zobaczmy jeszcze jak zachowają się zmienne automatyczne inicjalizowane. Jako przykład niech posłuży łańcuch (string) z cyframi (kwalifikator *const* informuje kompilator, że jest to szablon tylko do odczytu):

```
int Myfunc(char x,char y)
{
const char Cyfry[] ="0123456789";
static char a,b;

 a=2*x + y;
 b=x + 2*y;
 return(a+b+ Cyfry[1]);
}
```

Wydawałoby się, że w trakcie tworzenia ramki stosu dla funkcji podczas jej wywołania powinna być powtórzona procedura taka sama jak dla zmiennych inicjalizowanych *data* (przepisanie wartości z końca obszaru kodu bezpośrednio na stos). Niestety w tym przypadku avr-gcc nie postępuje optymalnie. Sprawdźmy to w AvrStudio – **rys. 17.**

Okazuje się, że *string Cyfry[]* jest już w trakcie ogólnej inicjalizacji również przepisywany na stałe do obszaru *data* SRAM (podobnie jak wszystkie "zwykłe" zmienne inicjalizowane) gdzie spokojnie czeka na wywołanie funkcji. Wtedy dopiero spod adresu w sekcji *data* jest przepisywany do ramki stosu.

Zamiast spodziewanych korzyści mamy więc w efekcie wydłużenie kodu wykonywalnego i żadnej oszczędności RAM w porównaniu z przypadkiem użycia tego stringa jako zwykłej zmiennej globalnej (ewentualnie lokalnej ale dla całego modułu). Widać więc, że takiej konstrukcji należy raczej unikać (chyba, że czytelność kodu postawimy na absolutnie priorytetowym miejscu).

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.