AVR–GCC: kompilator C dla mikrokontrolerów AVR, część 9

Zakres zmiennych, pliki nagłówkowe

Zgłębiając tajniki AVR–GCC przechodzimy teraz do omówienia sposobów deklarowania zmiennych oraz ich dopuszczalnych zakresach. Jak pokazuje praktyka, zrozumienie tych zagadnień ma duży wpływ na komfort pracy programisty i – w konsekwencji – na jakość przygotowanego oprogramowania.

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. Ich zwykłe 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 widocz-

na 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. 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 resecie 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       mov   r18, r24
 5e: 86 2f       mov   r24, r22
 char a,b;

 a=2*x + y;
 60: 92 2f       mov   r25, r18
 62: 99 0f       add   r25, r25
 64: 96 0f       add   r25, r22
 b=x + 2*y;
 66: 88 0f       add   r24, r24
 68: 82 0f       add   r24, r18
 return (a+b);
 6a: 29 2f       mov   r18, r25
 6c: 33 27       eor   r19, r19
 6e: 27 fd       sbrc  r18, 7
 70: 30 95       com   r19
 72: 99 27       eor   r25, r25
 74: 87 fd       sbrc  r24, 7
 76: 90 95       com   r25
 78: 82 0f       add   r24, r18
 7a: 93 1f       adc   r25, r19
 7c: 08 95       ret
}
```

Optymalizator wykonał wszystkie potrzebne działania w obszarze rejestrów w sposób na tyle „zwięzły”, że nawet 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 asemblera 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).

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.