AVR–GCC: kompilator C dla mikrokontrolerów AVR, część 10
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.

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. 23**.

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

Rys. 23. Podgląd zmiennych lokalnych na stosie

```
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. 23). Przekonamy się, że wartości chwilowe *a* i *b* zmieniają się w zależności od tego, która funkcja aktualnie z nich korzysta.

Może 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
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:*

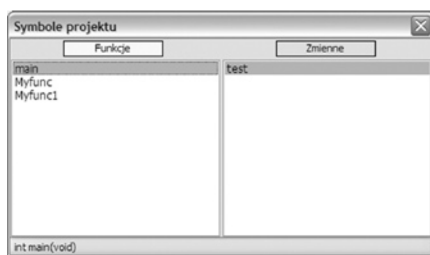
```
// deklaracja zmiennej lokalnej dla mo-
dułu funkcje
static char k=2;

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. 24**), 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. 25**), 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*



Rys. 24. Tablica symboli pokazuje tylko symbole globalne

```
Memory
Data          8/16  abc.   Address: 0x60
000060 02 0A 00 01 FF FF FF FF  ....
000068 FF FF FF FF FF FF FF FF  ........
000070 FF FF FF FF FF FF FF FF  ........
000078 FF FF FF FF FF FF FF FF  ........
000080 FF FF FF FF FF FF FF FF  ........
```

Rys. 25. Przydział pamięci dla zmiennych lokalnych

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:

```
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. 23 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. 26**.

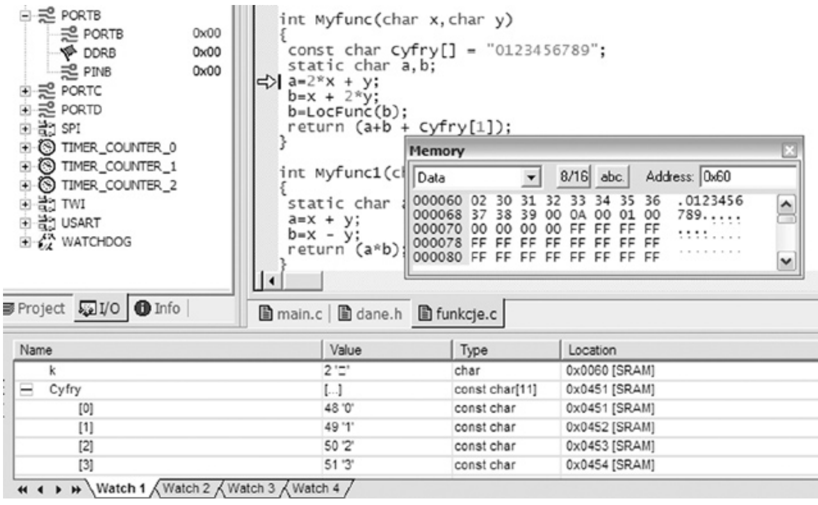
Okazuje się, że string *Cyfry[]* jest już w trakcie ogólnej inicjalizacji również przepisywany na stałe do obszaru *data* SRAM (podobnie jak wszystkie "zwykłe" zmienne inicjalizowane), gdzie spokojnie czeka na wywołanie funkcji. Wtedy dopiero spod adresu w sekcji *data* jest przepisywany do ramki stosu.

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).

Wykorzystanie plików nagłówkowych

Do używanych w projekcie plików nagłówkowych **.h* odwołujemy się w dwojaki sposób: *#include <file.h>* albo *#include „file.h"*. Różnica leży tylko w sposobie wyszukiwania przez kompilator na dysku pliku o podanej nazwie. Przy odwołaniu <> w pierwszej kolejności sprawdzane są własne, systemowe zasoby plików *avr–gcc* (a więc foldery *avr\include* i *lib\gcc\avr\wersja\include*), do których nie musimy podawać ścieżki. Dołączamy więc takim zapisem wszystkie potrzebne w projekcie pliki *avr–libc*. Konieczne jest jednak zaznaczenie wejścia do ewentualnych subfolderów a więc np. *#include <avr/io.h>*. Użycie uniksowego *slasha* / zamiast windowsowego *backslasha* \ nie jest błędem: kompilator interpretuje go poprawnie natomiast znacznie ułatwione jest przeniesienie projektu do środowiska uniksowego (jak Linuks).

Forma „" uruchamia wyszukiwanie pliku od bieżącego folderu projektu oraz dodatkowych podanych kompilatorowi lokalizacji. W ten sposób powołujemy się więc na własne pomocnicze pliki nagłówkowe projektu (jak *dane.h* w naszym przykładzie). Dodatkowe ścieżki przeszukiwania wprowadzamy przy pomocy opcji *–Iścieżka*. AvrSide wspiera jedną pomocniczą lokalizację, którą wpisujemy w oknie edycyjnym dialogu *Konfiguracja projektu > Ścieżki*. Zazwyczaj będzie to folder [*\AvrSide\Myinc*] przewidziany na ogólne własne pliki nagłówkowe z ulubionymi typami,



Rys. 26. Zmienne lokalne funkcji w wersji inicjalizowanej

definicjami, pomocniczymi makrami używanymi w wielu projektach. Zauważmy jednak, że narzucenie konkretnej pełnej ścieżki dostępu może sprawiać kłopoty przy przenoszeniu projektu na inną maszynę z inaczej zainstalowanym AvrSide – po prostu takiej ścieżki może nie być co spowoduje błędy. Dla takich przenośnych zastosowań przewidziałem dodatkowe subfoldery foldera projektu [*lib*] oraz [*inc*]. Posługują się one lokalizacją względną nie utrudniającą przenoszenia. Utworzenie tych subfolderów jest dokonywane samoczynnie przy zapisaniu projektu z ustawioną opcją *Konfiguracja projektu>Ścieżki>Używaj lokalnych podkatalogów lib/inc*. Automatycznie jest także przy wywołaniu kompilatora dodawana ścieżka do subfoldera [*inc*].

Z lokalizacją własnych plików nagłówkowych związana jest jeszcze jedna opcja AvrSide: *Konfiguracja projektu>AvrSide>Szukaj deklaracji w folderach inc*. Jest ona połączona z mechanizmem podpowiedzi deklaracji symbolu (**F1**). Przeszukanie plików w poszukiwaniu deklaracji odbywa się domyślnie wewnątrz foldera projektu. Zaznaczenie opcji powoduje również przeszukanie plików w dodatkowych lokalizacjach – jest to bardziej uniwersalne ale może spowalniać pracę AvrSide w przypadku nagromadzenia dużej liczby plików.

Następne ułatwienie w pracy z plikami nagłówkowymi dotyczy zależności. Mianem tym określamy w tym przypadku ustalenie, w których plikach źródłowych **.c* oraz **.s* jest używany dany plik nagłówkowy. Zmiana w takim pliku nagłówkowym pociąga za sobą oczywiście zmiany w tych źródłach co wymaga ich ponownego przekompilowania. Jednak bez samoczynnego wsparcia musielibyśmy albo zawsze pamiętać, których plików mogą dotyczyć zmiany albo zawsze po korekcie dowolnego nagłówka ponownie kompilować cały projekt (*Build*), co przy rozbudowanych programach może znacznie wydłużać całą operację. Obsługa zależności w AvrSide jest dosyć uproszczona: polega na przeszukaniu pierwszych kilkunastu wierszy kodu każdego źródła *.c* i *.s* i utworzeniu tabeli zależności na podstawie znalezionych dyrektyw dołączenia (*#include*) wszelkich otwartych w projekcie (obecnych na zakładkach edytora) plików nagłówkowych **.h*. Tabela ta jest później sprawdzana przy zapisywaniu zmian w pliku nagłówkowym – na tej podstawie są oznaczane jako zmienione (podkreślenie nazwy pliku na zakładce edytora) i włączane do polecenia *Make* odpowiednie moduły źródłowe.

Dla przyśpieszenia działania środowiska aktualizacja tabeli zależności nie odbywa się samoczynnie; po wprowadzeniu w projekcie korekt takich jak dodanie lub usunięcie pliku czy też dopisanie (usunięcie) dołączenia nagłówka w kodzie źródłowym, należy użyć polecenia menu *Projekt>Aktualizuj zależności*. Nie musimy jednak tego każdorazowo robić przy ładowaniu projektu – w tym przypadku utworzenie tabeli jest automatyczne. Powyższe polecenie jest także przydatne dla zresetowania tabeli w razie wystąpienia błędu, którego niestety nie udało mi się dotychczas zlokalizować, a który powoduje ciągłe oznaczenie plików źródłowych jako zmienione pomimo ich przekompilowania.

Należy też pamiętać, że powyższa obsługa jest jednopoziomowa, nie wspiera zagnieżdżonych dołączeń (w nagłówku *1.h* dołączamy *2.h* a z kolei *1.h* jest dołączony do *3.c* – nasza uproszczona obsługa wykryje zmianę w *1.h* ale w *2.h* już nie, konieczna jest komenda *Build*). Jak widać nie może się więc ona równać z rozbudowanymi narzędziami kontroli zależności używanymi w plikach *makefile*. Jednak dość dobrze zdaje egzamin w przeciętnej wielkości amatorskich projektach, dla których przede wszystkim AvrSide powstało.

Często zbiór funkcji obsługujących konkretne zadanie (np. obsługę urządzenia peryferyjnego) lokujemy w oddzielnym module źródłowym *xx.c* a dostęp do niego realizujemy poprzez skojarzony plik nagłówkowy o takiej samej nazwie *xx.h* . AvrSide oferuje dodatkowe skróty klawiszowe (*SHIFT + F6* albo *ALT + UP*) do szybkiego przełączania pomiędzy tak nazwaną parą plików.

Jerzy Szczesiul, EP
jerzy.szczesiul@ep.com.pl