AVR-GCC: kompilator C mikrokontrolerów AVR, część 3

Kontynuujemy cykl artykułów, których zadaniem jest przedstawienie podstaw oraz praktycznych zasad programowania mikrokontrolerów AVR w języku C z użyciem kompilatora avr-gcc. Oczywiście wybór kompilatora AVR-GCC może się jednym podobać, a innym nie. Postaramy się jednak uzasadnić, że nie jest to zły wybór.

Zmienne liczbowe i organizacja pamięci wewnętrznej ATmega

Pisząc oprogramowanie dla mikrokontrolera cały czas operujemy na rozmaitych wielkościach: odczytujemy, uśredniamy i filtrujemy wyniki przetwarzania ADC, zliczamy impulsy na wejściach licznikowych, odmierzamy czas, wyświetlamy napisy i liczby na różnego rodzaju wyświetlaczach, wyliczamy wypełnienie cyklu PWM itd. Wszystkie te wielkości zmieniające swoją wartość w trakcie działania programu noszą ogólną nazwę zmiennych. Klasyfikacja zmiennych jest bardzo różnorodna, na przykład:

– według pełnionej funkcji: zmienne liczbowe, znakowe, logiczne, tekstowe (łańcuchowe), wskaźniki;
– według złożoności: zmienne proste (np. pojedyncza liczba) i złożone (tablice, struktury, unie);
– według zakresu, znaku oraz typu liczby (dotyczy zmiennych liczbowych);
– według sposobu obsługi przez kompilator (inicjalizowane lub nie).

Tutaj zajmiemy się sposobami używania różnych zmiennych w avr–gcc. Bardziej sformalizowane i szczegółowe opisy i klasyfikacje znajdziemy w każdym uniwersalnym podręczniku języka C. Ponieważ każda zmienna jest dla mikrokontrolera po prostu pewną liczbą bajtów ulokowanych pod znanym adresem w pamięci danych, zobaczmy najpierw jak avr–gcc zarządza tą pamięcią (a konkretnie obszarem przeznaczonym dla użytkownika – powyżej rejestrów SFR). Na **rys. 8** (zaczerpniętym z podręcznika avr–libc) widzimy domyślnie stosowany schemat wykorzystania wewnętrznego SRAMU.

Stos (jak już stwierdziliśmy wcześniej) rozpoczyna się od końcowego adresu (określanego w avr–libc symbolem RAMEND) – jest wypełniany „w dół" czyli dekrementowany.

Bezpośrednio za obszarem SFR rozpoczyna się sekcja *.data*, w której konsolidator umieszcza wszystkie zmienne inicjalizowane (z przypisaną wstępnie niezerową wartością).

Dalej jest ulokowana sekcja *.bss*, zawierająca zmienne bez przypisanej wartości, które zgodnie ze standardem C zostają na początku programu wyzerowane;

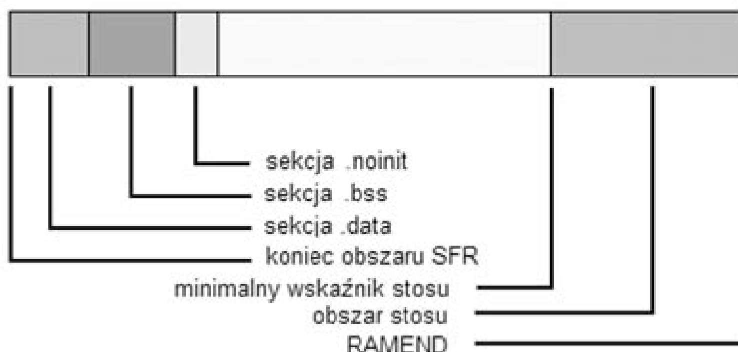
Następnie przewidziano dodatkową – specyficzną dla mikrokontrolerów – sekcję *.noinit*. Obejmuje ona zmienne, które chcemy pozostawić wyłącznie pod własną kontrolą – kompilator nie wykonuje na nich żadnych automatycznych operacji. Ma to na celu głównie zróżnicowanie sposobu inicjalizowania niektórych zmiennych w zależności od przyczyny resetu (np. możemy zechcieć aby licznik czasu pracy urządzenia był zerowany po włączeniu zasilania, ale zachowywał swoją wartość podczas resetu spowodowanego zadziałaniem watchdoga). Oczywiście wtedy musimy sami zadbać w kodzie o wpisanie odpowiednich wartości (także zer) – w przeciwnym razie pozostaną one całkowicie przypadkowe.

Pojawiło się tutaj pojęcie sekcji – jest to mechanizm wykorzystywany przez konsolidator do podziału dostępnych zasobów pamięci na poszczególne obszary i odpowiedniego przydzielenia do nich składników programu (oprócz wspomnianych powyżej sekcji danych mamy do czynienia z sekcją *.text* opisującą kod programu umieszczony w pamięci Flash oraz sekcją *.eeprom* przeznaczoną dla zawartości wewnętrznego EEPROMu kostki). Adresy startowe sekcji oraz ich maksymalne rozmiary dla danego mikrokontrolera znajdziemy we wspomnianych już wcześniej skryptach linkera.

Przypisanie zmiennej do konkretnej sekcji jest realizowane albo domyślnie przez konsolidator (sekcje *.data* oraz *.bss* są obsługiwane samoczynnie na podstawie deklaracji zmiennej) albo poprzez dodatkowy atrybut (dla *.noinit* lub *.eeprom*).

Zobaczmy teraz jak to działa w praktyce. Załóżmy sobie w AvrSide – zgodnie z poprzednimi opisami – nowy projekt Test02 w subfolderze [*Projects\Kurs\Przyklad–02*], z jednym plikiem źródłowym *main.c*. Zadeklarujmy kilka zmiennych typu *int* (mają one rozmiar 2 bajtów, o czym dokładniej za chwilę) oraz zdefiniujmy uproszczony zapis atrybutu sekcji *.noinit* (ta ostatnia operacja nic nie zmienia w działaniu kodu, służy wyłącznie wygodzie pisania):

```
// główny moduł projektu
#define _MAIN_MOD_ 1
#define NOINIT __attribute__ ((section
(„.noinit”)))
// pliki dołączone (include):
// dane:
int data1 = 2;  // zmienna inicjalizowa-
```



Rys. 8. Sekcje pamięci w wewnętrznym SRAM ATmega

```
na wartością
int bss1;   // zmienne zerowane
int bss2;
int noinit1 NOINIT;  // zmienna nie ini-
cjalizowana
// funkcje:
//==================
// funkcja main()
int main(void)
{
 // inicjalizacja
 noinit1 = 0x55;  // tutaj samodzielnie
inicjalizujemy zmienną NOINIT
 // pętla główna
 while (1)
  {
  }
}
```

Po kompilacji pasek statusu pokaże nam zużycie RAM równe 8 bajtów – jest to suma zmiennych we wszystkich sekcjach (4*2). Po bardziej szczegółowe informacje sięgnijmy do pliku rejestracyjnego *Text02.txt*. Znajdziemy tam m.in. tabelę dokładnej specyfikacji używanych sekcji (utworzoną w wyniku wywołania narzędzia avr–objdump z opcją –h):

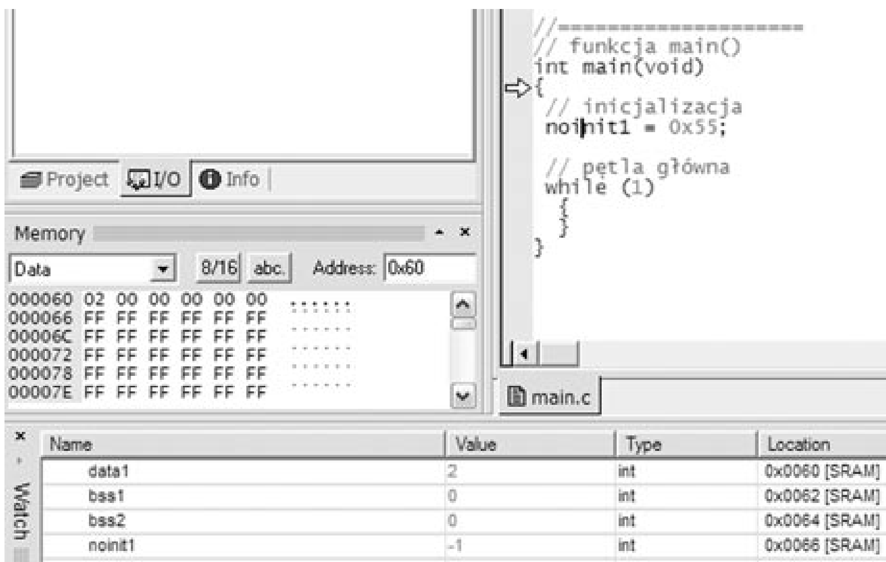
```
Sections:
Idx  Name       Size      VMA
  0  .text      00000072  00000000
  1  .data      00000002  00800060
  2  .bss       00000004  00800062
  3  .noinit    00000002  00800066
  4  .eeprom    00000000  00810000
```

Widzimy, że zmienne powędrowały do odpowiednich sekcji (jedna inicjalizowana – 2 bajty w *.data*, dwie zerowane – 4 bajty w *.bss*, jedna „samodzielna" – 2 bajty w *.noinit*; w *.eeprom* nie deklarowaliśmy nic). W kolumnie VMA znajdziemy też adres startowy każdej sekcji (początek *.data* to 0x60 – zaraz po obszarze SFR w Atmega 8 ; rolę przesunięcia 0x800000 wyjaśnimy później).

Spójrzmy jeszcze na sekcję kodu *.text*. Ma ona rozmiar 0x72=114 bajtów. Wynik kompilacji w AvrSide pokazał nam 116 bajtów. Te dodatkowe dwa bajty to właśnie początkowa wartość zmiennej *data1*. Nie weźmie się ona przecież „z powietrza" i musi być gdzieś przechowywana – avr–gcc dopisuje ją na końcu pliku wynikowego kodu, skąd przy starcie programu jest przepisywana (spójrzmy jeszcze raz na omówiony wcześniej kod automatycznej inicjalizacji) pod odpowiedni adres SRAM. Zajęty obszar zasobów Flash jest więc w rzeczywistości równy sumie sekcji *.text* i *.data* – taki też rezultat wyświetla AvrSide.

Zobaczmy teraz jak powyższe zmienne zachowają się w AvrStudio. Po uruchomieniu nowej sesji ustawmy sobie podgląd wszystkich zmiennych oraz uaktywnijmy okienko pamięci z obszarem danych (**rys. 9**). Widzimy, że *data1* przybrała odpowiednią początkową wartość (2), zmienne *bss1* i *bss2* zostały wyzerowane, a *noinit1* pozostała bez ingerencji (symulator AvrStudio jest nieco wyidealizowany i nadaje jej wartość 0xffff, w rzeczywistości komórki SRAM mogą po włączeniu zasilania zawierać całkiem przypadkowe wartości). Przejdźmy teraz pracą krokową (**F11**) do pętli *while*. Zmienna *noinit1* przybierze wartość zgodną z wpisanym przez nas kodem (0x55 czyli 85 dziesiętnie). Jeśli teraz zresetujemy program (**Shift + F5**), to zobaczymy, że wartość *noinit1* pozostanie nienaruszona.

Zwróćmy uwagę, że debugger C z AvrStudio całkowicie pomija automatyczną inicjalizację – widzimy od razu efekty jej działania (możemy ją prześledzić w oknie disasemblera, ale niestety bez prawidłowego podglądu zawartości pamięci).



Rys. 9. Zmienne inicjalizowane w AvrStudio

Omówimy teraz dokładniej używane przed chwilą zmienne liczbowe.

1 – Najprostszą wersją zmiennej liczbowej jest liczba całkowita bez znaku, (czyli podzbiór liczb naturalnych oraz zero). Avr–gcc obsługuje następujące typy liczby bez znaku, różniące się tylko wielkością:

unsigned char – zajmuje jeden bajt, może więc przyjąć wartość od zera do 0xff czyli 255;

unsigned int – 2 bajty, a więc 0 – 0xffff (65535);

unsigned long – 4 bajty – 0 – 0xffffffff (4294967295);

unsigned long long – 8 bajtów – do rzeczywiście wielkich wartości (raczej rzadko będzie nam potrzebny w świecie małych mikrokontrolerów, nie jest też obsługiwany przez AvrStudio).

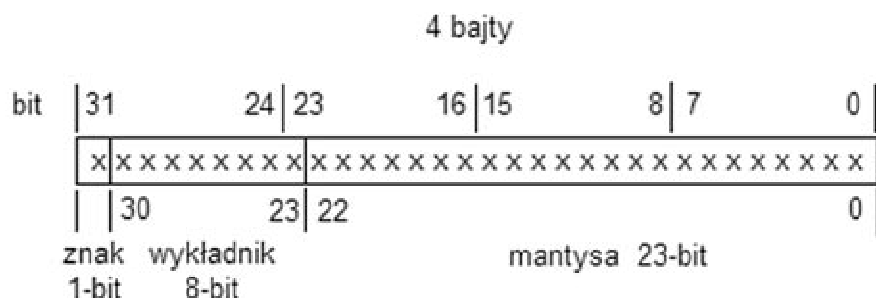
Te typy są interpretowane najbardziej bezpośrednio – wartość jest po prostu równa zawartości odpowiedniej liczby jednobajtowych komórek pamięci. Jednak nawet w tym prostym przypadku konieczne jest przyjęcie pewnej konwencji – określanej mianem „*endianess*" – czyli sposobu uporządkowania kolejnych bajtów liczby w pamięci. W różnych kompilatorach możemy napotkać dwie przeciwstawne metody:

big endian – bajty liczby są lokowane pod kolejnymi adresami pamięci od najbardziej do najmniej znaczącego (czyli np. liczba *unsigned long* 0x11223344 będzie zapamiętana w SRAM jako kolejno: 0x11, 0x22, 0x33, 0x44);

little endian – bajty liczby są lokowane od najmniej znaczącego (czyli 0x44, 0x33, 0x22, 0x11).

Jeśli spojrzymy na rys. 9 (oraz obejrzymy generowany kod asemblera) zauważymy od razu, że avr–gcc posługuje się modelem *little–endian*. Zazwyczaj ta informacja nie będzie nam specjalnie potrzebna, kompilator sam dba o odpowiedni porządek, jednak może być przydatna w momencie wykorzystywania zmiennej wielobajtowej z poziomu wstawki asemblerowej.

Aby sprawy nie wyglądały tak prosto należy dodać, że niektóre opcje kompilacji potrafią zmieniać domyślny rozmiar powyższych typów. Jeśli więc mamy w planach ich stosowanie (konkretnie chodzi o opcję *–mint8*, która zmniejsza rozmiary typów liczb, a tym samym pozwala na zredukowanie w razie konieczności objętości kodu), to dla zmiennych o



Rys. 10. Zapis pojedynczej precyzji liczby rzeczywistej

wymaganym znanym i stałym rozmiarze użyjmy raczej typów zdefiniowanych w pliku nagłówkowym *stdint.h* w subfolderze [*avr\include*] kompilatora. Jest to metoda bardzo zalecana przez autorów avr–libc jako zapewniająca całkowitą jednoznaczność określonego typu przy różnych warunkach kompilacji (np. *int8_t* ma zawsze 1 bajt, *int16_t* – 2 bajty itd.). Pozostaje oczywiście kwestia indywidualnych gustów i przyzwyczajeń, jednak C pozwala za pomocą operatora *typedef* określić zupełnie dowolne własne nazwy typów (np. często spotykane *s08*, *s16*, *u08*, *u16*). W prezentowanych przykładach również używam nazewnictwa tradycyjnego, które mi jakoś lepiej pasuje niż standard proponowany w avr–libc.

2 – Liczby całkowite ze znakiem są już nieco bardziej skomplikowane. Znak jest określony stanem najstarszego bitu – 0 oznacza plus, a 1 minus. Jednak wbrew oczekiwaniom pozostałe bity określają bezpośrednio wartość liczby tylko dla wartości dodatniej, wartości ujemne są zakodowane w tzw. dopełnieniu do dwóch (U2). Obejrzyjmy to zaraz w symulatorze wpisując do naszych zmiennych (np. *data1*) różne wartości i oglądając w okienku pamięci ich bajtową reprezentację. Na przykład wartość –1 zostanie zapisana jako 0xffff. Taki mało intuicyjny sposób kodowania wynika z prostoty zachowania wartości przy rzutowaniu typów oraz symetrycznej i niezależnej od liczby bajtów procedury odwracania znaku. Znak liczby w kodzie U2 zmieniamy negując wszystkie bity liczby i dodając do wyniku jeden. Możemy od razu sprawdzić jak to działa w praktyce poddając odpowiednim operacjom naszą zmienną – np. *data1*. Dopisujemy na początku *main()* sekwencję:

```
// inicjalizacja
data1 = ~data1;
data1 +=1;
data1 = ~data1;
data1 +=1;
```

Musimy też dodać do deklaracji *data1* słowo kluczowe *volatile* (*volatile int data1 = –1;*) gdyż w przeciwnym razie optymalizator wytnie zbędne z jego punktu widzenia pośrednie operacje na *data1* wstawiając od razu ostateczny wynik.

Oczywiście, ponieważ zajęty został najbardziej znaczący bit – zakres wartości bezwzględnej typu zostanie zmniejszony o mniej więcej połowę. Ze sposobu kodowania wynika pewna asymetria: np. dla typu *int* wartością maksymalną będzie 0b0111 1111 1111 1111 (czyli 0x7fff)=32767 zaś minimalną 0b1000 0000 0000 0000 (0x8000)=–32768.

Liczby ze znakiem możemy dla pełnej jasności deklarować ze słowem kluczowym *signed*, ale ponieważ jest to opcja domyślna zazwyczaj ją pomijamy.

Należy jeszcze dodać, że typ *int* jest domyślny dla kompilatora. Wszędzie gdzie z kodu nie wynika jednoznacznie, jakiego „rozmiaru" liczby użyć w operacji stosowany jest *int* (tzw. promocja do *int*). Czasem jest to pożyteczne, ale w pewnych przypadkach powoduje zgoła nieprzewidziane rezultaty (np. „obcinanie" wielkości liczb na pośrednich etapach bardziej złożonych przeliczeń). Będziemy do tej sprawy, jak również powiązanego z nią rzutowania typów wielokrotnie przy różnych okazjach powracać.

Jak widać z powyższego staranność i uwaga przy doborze odpowiedniego typu dla zmiennej może w wielu przypadkach wręcz decydować o poprawności działania programu. Przekroczenie zakresu, potraktowanie wartości dodatniej jako liczby ze znakiem (np. ten sam bajt 0xff zadeklarowany jako *unsigned char* jest traktowany przez kompilator jako wartość dodatnia +255, natomiast użyty jako *signed char* zostanie odczytany jako –1) itp. mogą powodować trudne do zlokalizowania (i w żaden sposób nie sygnalizowane na etapie kompilacji) błędy.

3 – Liczby rzeczywiste – możemy je obsługiwać w dwojaki sposób. Uproszczona forma to tzw. zapis stałoprzecinkowy (*fixed point*). Używamy tutaj z góry określonej i niezmiennej liczby cyfr po przecinku, niezależnie od wartości. Przykładem z codziennego świata liczb dziesiętnych mogą być ceny. Zauważmy, że stosowanie liczb stałoprzecinkowych w programie jest – przy odpowiednim doborze jednostek – równoznaczne z obliczeniami na liczbach całkowitych. Np. chcemy mierzyć napięcie w woltach z rozdzielczością 0,001 V. Wystarczy wtedy zaprojektować tor analogowo – cyfrowy tak, aby jednemu najmniej znaczącemu bitowi wyniku konwersji AC odpowiadał 1 mV sygnału wejściowego. Wszystkie wewnętrzne pomocnicze obliczenia (filtrowanie, alarmy itp.) wykonujemy wtedy na wartościach całkowitych wyrażonych w miliwoltach. Ostateczna prezentacja wyniku w woltach będzie polegać wyłącznie na wstawieniu kropki dziesiętnej w odpowiednim miejscu.

Zapis stałoprzecinkowy sprawdzi się dobrze jeśli wartość zmiennej pozostaje w ustalonym, znanym zakresie. Jednak wartości zbyt małe lub zbyt duże nie będą przedstawiane skutecznie. Np. dla dużych wartości mierzonych z dokładnością 1000 istotne jest rozróżnienie pomiędzy 1200000 a 1201000 – zapis 1200000,00 nie wnosi żadnej informacji i może prowadzić tylko do marnowania czasu programu na zbędne przeliczenia. Z kolei małe liczby będą całkiem nierozróżnialne: zarówno 0,001 jak i 0,004 zostaną zapisane jako 0,00. Od razu widać, że istotna jest nie przyjęta liczba cyfr po przecinku, ale grupa cyfr znaczących niosąca rzeczywistą informację o wartości.

W takich przypadkach stosujemy format zmiennoprzecinkowy (*floating point*). Jest on bardzo podobny do znanej notacji tzw. inżynierskiej: liczba jest zapisana jako mantysa (przedstawiająca grupę cyfr znaczących) oraz wykładnik potęgi (określający mnożnik decydujący o wielkości liczby). Zarówno wartości wielkie np. 2,78E6 (czyli $2{,}78*10^6=2780000$), jak i małe np. 3,55E–3 (czyli $3{,}55*10^{-3}=0{,}00355$) są przedstawione w jednakowy sposób bez utraty dokładności. Zapis samej mantysy jest również znormalizowany: wartość X umieszczona przed kropką dziesiętną zawiera się w zakresie $1=<X<10$ (a generalnie < podstawa używanej

potęgi) – nie piszemy np. 35,5E2 (3550), ale 3,55E3.

Reprezentacja binarna formatu zmiennoprzecinkowego jest określona specyfikacją IEEE 754 (*Institute of Electrical and Electronics Engineers* – zajmuje się m.in. międzynarodowymi standardami i normami). Stosowane są dwie odmiany zapisu: pojedynczej precyzji (*single precision*) zajmujący 4 bajty oraz znacznie dokładniejszy podwójnej precyzji (*double precision* – 8 bajtów). Avr–gcc obsługuje tylko pojedynczą precyzję (możemy sprawdzić, że niezależnie od użytej w programie deklaracji zmiennej: *float* czy *double* zajmie ona zawsze tylko 4 bajty). Znaczenie poszczególnych bitów w 4–bajtowym pakiecie przedstawia **rys. 10**.

Najstarszy bit (31) określa znak liczby: 0 odpowiada wartości dodatniej, 1 – wartości ujemnej.

Wykładnik potęgi o podstawie 2 jest zakodowany w 8–bitowym polu (30 – 23) jako wartość binarna tego pola pomniejszona o stałe przesunięcie 127 (czyli np. 0000 0010=2 oznacza wykładnik –125, 1000 0000=128 oznacza 1 itd.). Wartości 0000 0000 i 1111 1111 są zarezerwowane (o czym za chwilę) więc zakres wykładnika wynosi od –126 (0000 0001=1–127) do 127 (1111 1110=254–127).

Trochę więcej uwagi musimy poświęcić mantysie. Zgodnie z poprzednimi informacjami powinna być ona zapisana jako C.UUUU... gdzie część całkowita C zawiera się pomiędzy 1, a podstawą potęgi zaś U jest częścią ułamkową. W zapisie binarnym podstawą jest 2, a więc: 1<=C<2 co jak widać jest równoznaczne z warunkiem C=1. Dlatego w powyższej bitowej reprezentacji C zostało całkowicie pominięte (jest domyślnie traktowane jako 1) i wszystkie 23 bity mantysy są użyte do określenia części ułamkowej. Kolejne bity „ułamkowe" odpowiadają potęgom 2^{-n} gdzie n jest pozycją bitu za kropką dziesiętną. Pierwszy bit za kropką, (czyli pierwszy w opisywanej mantysie, co odpowiada bitowi nr 22 w całej 4–bajtowej paczce) ma, więc wartość (wagę) 2^{-1} (½), następny 2^{-2} (¼) itd. Dla określenia całej wartości należy zsumować wagi wszystkich bitów mantysy równych jeden oraz domyślny bit z wagą 2^0=1.

Sprawdźmy jak to działa w praktyce. Zadeklarujmy w naszym projekcie zmienną *float* i na początku funkcji *main* przypiszmy jej wartość np. –300,0. Po skompilowaniu znajdziemy kod (**F7** – relokowalny):

```
tst=-300.0;
   8: 80 e0        ldi   r24, 0x00   ; 0
   a: 90 e0        ldi   r25, 0x00   ; 0
   c: a6 e9        ldi   r26, 0x96   ;
150
   e: b3 ec        ldi   r27, 0xC3   ;
195
  10: 80 93 00 00   sts   0x0000, r24
  14: 90 93 00 00   sts   0x0000, r25
  18: a0 93 00 00   sts   0x0000, r26
  1c: b0 93 00 00   sts   0x0000, r27
```

Do rejestrów została załadowana wartość 0xC3960000, co odpowiada zapisowi binarnemu:

1100 0011 1001 0110 0000 0000 0000 0000

Zgodnie ze specyfikacją:

znak=1 czyli liczba ujemna,

wykładnik=135 (1000 0111)–127=8; zatem mnożnik wyniesie 2^8=256,

mantysa=1+1/8+1/32+1/64($2^0+2^{-3}+2^{-5}+2^{-6}$)=75/64 (doliczamy domyślne 2^0).

Ostateczny wynik=–256*75/64=–300 – zgodnie z naszą instrukcją w kodzie.

Nietrudno teraz zauważyć, że najmniejszą możliwą do zapisania w ten sposób liczbą będzie 2^{-126} (minimalny wykładnik –126 i minimalna mantysa 2^0). A co z liczbami mniejszymi oraz zerem? Dla nich przewidziano właśnie wartość 0000 0000 w polu wykładnika. Obowiązuje wtedy oddzielna reguła przeliczania: wykładnik jest nadal równy –126 natomiast w mantysie uwzględniamy tylko część ułamkową, zaś 2^0 zostaje pominięte.

Z kolei druga zarezerwowana wartość pola wykładnika: 1111 1111 służy nie do przedstawiania konkretnej liczby ale stanowi sygnał, że w procesie obliczeń przekroczone zostały możliwe do zapisu zakresy. Spróbujmy wykonać eksperyment dzieląc liczbę dodatnią i ujemną przez 0. Uzyskamy właśnie takie rezultaty, interpretowane przez AvrStudio jako 1.#INF oraz –1.#INF (czyli + i – nieskończoność), a więc wartości nieokreślone. Dla ścisłości: nie jest to do końca zgodne ze standardem przewidującym dla takiego przypadku oddzielny sygnał NaN (*Not–a––Number* – to–nie–jest–liczba), ale w przeważajacej większości typowych zastosowań mikrokontrolerów nie ma to praktycznie żadnego znaczenia.

Dużo istotniejszym praktycznym aspektem stosowania liczb rzeczywistych jest zdawanie sobie sprawy, że przedstawiony powyżej zapis binarny jest tylko przybliżony i nie zawsze dokładnie odzwierciedli wartość liczby. Nasz przykład z liczbą –300 był pod tym względem przyjazny, ale np. zwyczajne 0,01 już się nie da przedstawić dokładnie (możemy to obejrzeć w podglądzie zmiennych AvrStudio: otrzymujemy 0,0099999998). Użycie takich przybliżonych liczb w dłuższych pętlach obliczeniowych może prowadzić do znaczącego – i niemożliwego do wyeliminowania błędu.

Znając już strukturę liczby rzeczywistej stwierdzimy, że nie jest żadnym problemem przesyłanie jej w różny sposób pomiędzy urządzeniami. Np. w komunikacji szeregowej po prostu transmitujemy kolejne 4 bajty, musimy jedynie pamiętać o ich takiej samej kolejności w buforze nadajnika i odbiornika oraz o jednakowej interpretacji (deklaracji) liczby w obu programach.

Bardzo uniwersalne i wygodne od strony programowej liczby *float* mają jednak w świecie małych mikrokontrolerów jedną zasadniczą wadę – pochłaniają dużo ograniczonych zasobów pamięci. Zróbmy znów szybki eksperyment wpisując do naszego testowego programu deklarację:

```
#ifdef FLOAT
double li1 = 2500;
double li2 = 400;
double wynik;
#else
long li1 = 2500;
long li2 = 400;
long wynik;
#endif
```

i wykonajmy gdzieś w *main()* prostą operację:

wynik=li1*li2;

Po skompilowaniu sprawdźmy wielkość kodu – nie jest zbyt duża (w przykładowym teście wynosiła 4% pojemności Flasha Atmega 8). A teraz przed powyższą deklaracją zdefiniujmy makro FLOAT:

#define FLOAT

i spróbujmy jeszcze raz. Kod gwałtownie rozrósł się do 21%!! Tyle kosztuje dolinkowanie potrzebnych funkcji obsługujących liczby zmiennoprzecinkowe (możemy je sobie przejrzeć w podglądzie **CTRL + F7**). Wniosek jest prosty: typ *float* pozostanie zarezerwowany dla większych kostek, w małych trzeba sobie radzić za pomocą liczb całkowitych.

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