AVR–GCC: kompilator C mikrokontrolerów AVR, część 4

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.

Dyrektywy kompilacji warunkowej

Na chwilę przerwiemy omawianie typów zmiennych i obejrzymy dokładniej zastosowane polecenie kompilacji warunkowej *#if #else #endif*. Dyrektywy kompilacji warunkowej nie wchodzą w skład wynikowego kodu programu. Są analizowane na samym początku przez preprocesor, który zgodnie z nimi modyfikuje kod źródłowy poddawany następnie kompilacji (sięgnijmy do wcześniejszego ogólnego opisu działania AVR–GCC). Powyższy zapis powoduje wstawienie do kodu tylko bloku zgodnego z narzuconym warunkiem (zdefiniowane makro) i pomija blok niezgodny. Pomimo dosyć podobnej składni dyrektywa nie ma więc nic wspólnego z programową instrukcją warunkową *if() else*; wykonywaną dopiero w trakcie działania programu.

Dyrektywy warunkowe mogą mieć kilka odmian. Warunek może być sformułowany jak powyżej: *#ifdef MACRO* – wtedy sprawdza po prostu czy *MACRO* zostało wcześniej zdefiniowane (lub odwrotnie gdy użyjemy *#ifndef MACRO*). Bardziej uniwersalną formą jest *#if WARUNEK* gdzie warunek może być stałą liczbową, stałą znakową albo dowolnym spełniającym reguły C wyrażeniem (operacją arytmetyczną, logiczną, bitową). If jest realizowane gdy *WARUNEK !=0*. W wyrażeniach możemy także sprawdzać zdefiniowanie makra – służy do tego oddzielny operator defined. Zapis *#ifdef MACRO* będzie więc równoważny z *#if defined(MACRO)*.

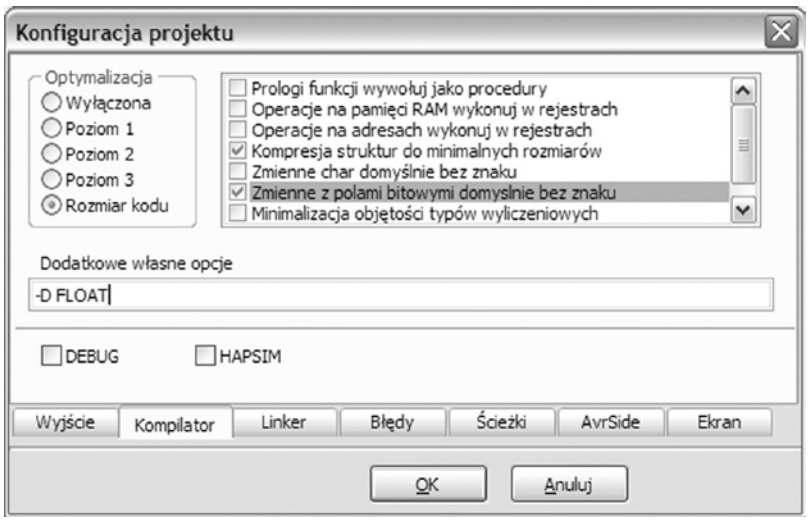
```
Dyrektywa może być pojedyncza:
#ifdef MACRO
Blok kodu.
#endif
albo złożona:
#ifdef MACRO
Blok kodu 1 (blok uwzględniany, gdy MA-
CRO zdefiniowane)
#else
Blok kodu 2 (blok uwzględniany, gdy MA-
CRO nie zdefiniowane)
#endif
```

Nieco bardziej kłopotliwe jest sprawdzenie kilku oddzielnych warunków. Wymaga to zagnieżdżenia kolejnych instrukcji *if*:

```
#if WARUNEK1
Blok kodu 1 (WARUNEK1 spełniony)
#else // (WARUNEK1 niespełniony)
#if WARUNEK2
Blok kodu 2 (WARUNEK1 niespełniony, WA-
RUNEK2 spełniony)
#else
Blok kodu 3 (WARUNEK1 niespełniony
i WARUNEK2 nie spełniony)
#endif  // zakończenie obsługi WARUNEK2
#endif  // zakończenie obsługi WARUNEK1
```

Dla ułatwienia wprowadzono dodatkowy operator *elif* – z jego pomocą zapiszemy to samo znacznie prościej:

```
#if WARUNEK1
Blok kodu1
#elif WARUNEK2
Blok kodu 2
#else
Blok kodu 3
#endif
```



Rys. 11. Definiowanie makra w opcjach wywołania kompilatora

W naszym przykładzie makro *FLOAT* zdefiniowaliśmy za pomocą dyrektywy *#define FLOAT* na początku pliku źródłowego. W przypadku konieczności zastosowania makra w wielu plikach wygodniej będzie umieścić definicję w jakimś wspólnym pliku nagłówkowym. Jeszcze inną możliwością jest dopisanie definicji do opcji wywołania kompilatora. W okienku edycyjnym dodatkowych opcji AvrSide (zakładka Kompilator) wpiszmy *–D FLOAT* i sprawdźmy, że działanie będzie takie samo (**rys. 11**). Po zmianie samych opcji (bez zmiany treści kodu) dla ponownego przekompilowania użyjmy polecenia *Build* (**CTRL+SHIFT+F9**) gdyż polecenie *Make* (**F9**) pominie kompilację nie zmienionego pliku źródłowego.

Jednym z typowych zastosowań może być wstawienie do kodu fragmentów używanych tylko przy uruchamianiu i testowaniu aplikacji. Powszechne jest też stosowanie w plikach nagłówkowych warunku:

```
#ifndef PLIK
#define PLIK
Treść pliku nagłówkowego.
#endif
```

Zabezpiecza to przed omyłkowym wielokrotnym wstawieniem treści pliku do kodu dyrektywami *#include*, po pierwszym wstawieniu makro PLIK jest już zdefiniowane i każda następna próba zostaje zablokowana.

Checkboxy *DEBUG* i *HAPSIM* powodują zdefiniowanie odpowiednio makr *DEBUG* i *HAPSIM*, które są pomocne w kodzie przy uruchamianiu i symulacji z użyciem Hapsima.

Zmienne logiczne, flagi bitowe, obsługa linii wejść/wyjść

Zmienne logiczne to zmienne przyjmujące tylko dwie wartości: prawda i fałsz. Są wygodne pod-

Rys. 12. Podpowiedź deklaracji makra w AvrSide

czas rozpatrywania rozmaitych rozgałęzień warunkowych. W języku C wydzielenie tych zmiennych ma charakter raczej umowny i służący przejrzystości kodu. Każda, bowiem wartość niezerowa jest traktowana jako logiczna „prawda", a zero jest równoważne z logicznym „fałszem". Np. w nieskończonej pętli naszego pierwszego testu użyliśmy po prostu warunku *while (1)* – jedynka jest zawsze prawdą i pętla będzie zawsze wykonana. Właśnie dla przejrzystości zdefiniowany został dodatkowy typ bool oraz jego wartości true oraz false. Aby je wykorzystać musimy dołączyć systemowy plik nagłówkowy *stdbool.h* (*#include <stdbool.h>*). Wtedy możemy używać bardzo czytelnego i jednoznacznego zapisu (np. *bool mybool=true;*). Typ bool nie jest niestety rozumiany przez AvrStudio, z czego można jednak łatwo wybrnąć definiując własny typ zgodny z 1–bajtowym char (i pozostawiając nazewnictwo *true* – *false* ze *stdbool.h*). Użyjemy do tego bardzo pożytecznego operatora typedef. Składnia jest bardzo prosta:

```
typedef    określenie_typu    własna_
nazwa_nowego_typu;
```

Zapiszmy więc w naszym kodzie np.:

```
typedef char boolean;
```

W ten sposób określiliśmy własny nowy typ boolean – całkowicie zgodny z char ale wyróżniający się w kodzie oddzielną nazwą – którego teraz możemy używać do definiowania zmiennych logicznych, np.:

```
boolean mybool = true;
```

AvrStudio poprawnie identyfikuje typy zdefiniowane za pomocą *typedef*. Natomiast AvrSide pozwala na dołączenie nowej nazwy do listy słów kluczowych, a tym samym jej odpowiednie kolorowanie w kodzie (po zaznaczeniu nazwy – np. dwukrotnym kliknięciem na niej – używamy skrótu **CTRL+K**). Lista dodatkowych słów kluczowych jest dostępna w dialogu Ustawienia na zakładce AvrSide (w tym miejscu można zbędne słowa usuwać z listy poprzez zaznaczenie pozycji i klawisz **DEL**, całą listę zerujemy natomiast skrótem **CTRL+DEL**). Pamiętajmy tylko, że długość bufora listy jest ograniczona – po jego wypełnieniu dalsze wpisy nie będą dokonywane.

Zmienne typu *bool* są dosyć rozrzutne – zajmują cały bajt tylko po to, aby określić stan jednego bitu (gdyż *true* i *false* to odpowiednio po prostu 1 i 0). Dlaczego więc nie używać tak popularnych w rodzinie ‚51 flag (zmiennych jednobitowych), które maksymalnie oszczędzają pamięć? Na przeszkodzie stoją następujące względy:

AVR nie ma przeznaczonego do ogólnego stosowania obszaru pamięci adresowanej bitowo (nie można tu uwzględniać rejestrów I/O, które mają całkiem inne przeznaczenie, po trzy takie rejestry ogólnego przeznaczenia mają tylko niektóre najnowsze Atmegi),

AVR–GCC niestety nie obsługuje dość popularnej w innych kompilatorach wygodnej składni dostępu do poszczególnych bitów (np. zmienna. X , PORTA.2 itp.) – wg obecnych informacji nie zanosi się tutaj na szybką zmianę.

Jednak wbrew tym ograniczeniom wcale nie musimy z flag bitowych rezygnować. Wymaga to tylko zastosowania nieco innych (ale standardowych dla C) metod. Działania na pojedynczych bitach będziemy wykonywać za pomocą operatorów bitowych (iloczyn *and* **&**, suma *or* **|** i suma wyłączna *ex–or* **^** – nie pomylmy jej z popularnym zapisem potęgowania!) oraz tzw. masek bitowych. Maska to po prostu liczba całkowita o potrzebnym rozmiarze (*char*, *int*, *long*) z ustawionymi (1) tylko określonymi bitami. Dla „zapalenia" jednego wybranego bitu w zmiennej sumujemy ją bitowo z maską o zgodnym rozmiarze, w której tylko ten bit jest ustawiony, np. dla bitu nr 3:

```
1100 0010  or  0000 1000 = 1100 1010
```

Dla zgaszenia tego samego bitu używamy iloczynu zmiennej z maską zanegowaną (z wartością wszystkich bitów zmienioną na przeciwną: ~0000 1000 = 1111 0111):

```
1100 1010 & ~0000 1000 = 1100 1010 &
1111 0111 = 1100 0010
```

Jeśli nie potrzebujemy konkretnego stanu bitu, a chcemy tylko zmienić jego wartość na przeciwną posłużymy się maską w operacji ex–or:

```
1100 0010 ex-or 0000 1000 = 1100 1010
1100 1010 ex-or 0000 1000 = 1100 0010
```

Maski potrzebne w powyższych działaniach uzyskujemy przesuwając w lewo liczbę jeden (czyli z ustawionym bitem nr 0) o ilość miejsc zgodną z lokalizacją bitu poddawanego działaniu:

```
0000 0001 << 1 = 0000 0010
0000 0001 << 5 = 0010 0000
```

i ewentualnie poddając je negacji (użyta w powyższych przykładach maska będzie więc wyrażona jako 1<<3).

Biblioteka avr–libc przewidziała dla tworzenia masek pomocnicze makro *_BV(numer_bitu)* – jest ono całkowicie równoważne z zapisem (1<<numer_bitu) i może być stosowane zamiennie według własnych preferencji (użycie *_BV* wymaga dołączenia na początku kodu systemowego pliku *io.h*, *#include <avr/io.h>*). Systemowe pliki nagłówkowe opisu poszczególnych mikrokontrolerów zawierają także nazwy poszczególnych bitów w rejestrach SFR, co pozwala podczas konfiguracji SFR na użycie symboli zgodnych z dokumentacją Atmela zamiast niewiele mówiących numerów bitu.

Zróbmy kilka testowych działań bitowych na zmiennej long z użyciem różnych masek (w oknie podglądu AvrStudio ustawmy format na hex żeby łatwo ocenić wynik), np.:

```
volatile unsigned long flagi;
```

(Ponownie zwróćmy uwagę na deklarację *volatile*, która nie pozwala optymalizatorowi na usunięcie z kodu operacji pośrednich, nie mających wpływu na końcową wartość zmiennej).

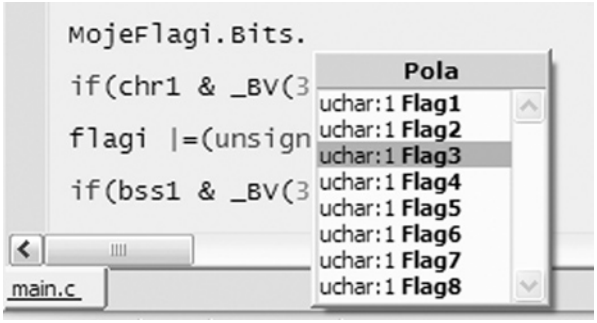
```
flagi |= _BV(5);
flagi |= _BV(12);
```

Oczywiście dla czytelności możemy w każdej chwili zdefiniować własną nazwę dla konkretnego bitu i używać jej zamiast liczby:

```
#define Flaga1  7
flagi |= _BV(Flaga1);
flagi &=~ _BV(Flaga1);
```

Zwróćmy przy okazji uwagę na kilka pułapek:

BV zwraca domyślnie typ *si-*



Rys. 13. Automatyczna lista pól unii / struktury

gned int, przesunięcie o 15 pozycji (ustawiony najstarszy bit) powoduje potraktowanie wyniku jako liczby ujemnej – rozwiązaniem jest rzutowanie typu na wymagany: *flagi |= (unsigned int) _BV(15 ;*

Przy przesunięciach większych niż 15 otrzymujemy ostrzeżenie o przekroczeniu rozmiaru typu. Dzieje się tak dlatego, że jedynka w wyrażeniu (1 << n) jest domyślnie traktowana jako int o szerokości 16 bitów. Makro *_BV* nie da sobie już z tym przypadkiem rady, użyjmy jawnego przesunięcia z odpowiednim rzutowaniem typu: *flagi |= (unsigned long)1<<16;*

Jest to o tyle mniej istotne, że *_BV()* zostało w zasadzie przeznaczone do obsługiwania 8–bitowych rejestrów SFR gdzie takie problemy nie wystąpią – jednak dobrze ilustruje różnego rodzaju niespodzianki związane z typami i zakresami zmiennych.

W podobny sposób sprawdzamy stan bitu: tworzymy iloczyn zmiennej i odpowiedniej maski i kontrolujemy czy jest równy zero czy nie. Na przykład w instrukcji warunkowej możemy bezpośrednio skorzystać z reguły, że każda wartość niezerowa odpowiada logicznej prawdzie:

if(flagi & _BV(3)) { coś wykonujemy } – wykonanie nastąpi przy ustawionym bicie 3 w zmiennej flagi (przy okazji obejrzyjmy wygenerowany kod, przekonamy się, że dla typu long jest on mocno skomplikowany więc w miarę możliwości w praktyce ograniczajmy rozmiar zmiennych używanych w takim celu; dla flagi typu char kod jest już króciutki i przejrzysty).

Identyczne reguły zalecane są przy obsłudze linii wejść / wyjść portów mikrokontrolera (i ogólnie przy dostępie do rejestrów SFR). Popularne dawniej pomocnicze makra sbi, cbi, outp, inp są obecnie wycofane z avr–libc (rozdział Deprecated List w podręcznku) – w zamian pojawiła się niedostępna wcześniej możliwość użycia SFR bezpośrednio w instrukcjach przypisania. Nie będziemy więc pisać np. outp (0x55, DDRB); ale po prostu DDRB=0x55; (należy zauważyć, że ceną tego postępu mogą być czasem niestety kłopoty ze starymi projektami). Zwróćmy uwagę, że kompilator samodzielnie wykonuje rozróżnienie pomiędzy rejestrami IO a rejestrami rozszerzonymi i stosuje odpowiednie instrukcje. Np. jeśli zmienimy typ procesora na Atmega 128 (domyślna w AvrSide Atmega 8 nie używa rozszerzonych SFR) i wypróbujemy zapis do portu B (przestrzeń IO) oraz G (adres rozszerzony 0x65) dostaniemy kod:

```
PORTB=0x55;
252: 25 e5          ldi   r18, 0x55   ;
85
254: 28 bb          out   0x18, r18   ;
24
PORTG=0x55;
256: 20 93 65 00    sts   0x0065, r18
```

Dla portu B użyty został skrócony rozkaz out. Dla portu G byłby on nieprawidłowy i kompilator stosuje zwykły zapis do pamięci sts. Podobnie jest przy obsłudze pojedynczych linii, np. dla ustawiania i gaszenia:

```
PORTB |= _BV(PB2);
252: c2 9a          sbi   0x18, 2  ; 24
PORTG |= _BV(PG2);
254: 80 91 65 00    lds   r24, 0x0065
258: 84 60          ori   r24, 0x04   ; 4
25a: 80 93 65 00    sts   0x0065, r24
```

oraz

```
PORTB &=~ _BV(PB2);
252: c2 98             cbi   0x18, 2  ;
24
PORTG &=~ _BV(PG2);
254: 80 91 65 00    lds   r24, 0x0065
258: 8b 7f             andi  r24, 0xFB
; 251
25a: 80 93 65 00    sts   0x0065, r24
```

albo dla sprawdzania stanu linii:

```
if(PINB & _BV(PB2)) PORTA |= _BV(PA0);
252: b2 99          sbic  0x16, 2  ; 22
254: d8 9a          sbi   0x1b, 0  ; 27
if(PING & _BV(PG2)) PORTA |= _BV(PA1);
256: 80 91 63 00    lds   r24, 0x0063
25a: 82 fd          sbrc  r24, 2
25c: d9 9a          sbi   0x1b, 1  ; 27
```

Powyższe operacje możemy dla nabrania wprawy prześledzić w AvrStudio, które w pełni wspiera obsługę oraz podgląd linii we/wy portów (potrzebna będzie oczywiście również zmiana procesora albo utworzenie nowej sesji).

W avr–libc (*\include\avr\sfr_defs.h*) znajdziemy kilka dodatkowych (zrealizowanych za pomocą opisanych powyżej operacji bitowych) makr obsługi linii:

bit_is_set(sfr, bit) sprawdza ustawienie (1) bitu nr bit w rejestrze sfr

bit_is_clear(sfr, bit) to samo tylko dla bitu zgaszonego (0)

loop_until_bit_is_set(sfr, bit) pętla oczekiwania na ustawienie (1) bitu nr bit w rejestrze sfr

loop_until_bit_is_clear(sfr, bit) to samo tylko oczekiwanie na zgaszenie (0) bitu.

Pętle oczekujące należy stosować rozważnie, – jeśli wprowadzimy warunek, który nigdy nie zaistnieje, zatrzymamy cały program (ewentualnie zresetujemy mikrokontroler o ile jest włączony watchdog).

Przy okazji należy podkreślić, żeby nie mylić makra z funkcją, chociaż są często bardzo podobne w składni.

Funkcja jest wywoływana dynamicznie w trakcie działania programu z chwilowymi, nieznanymi w chwili kompilacji, wartościami argumentów. Makro jest wykonane (rozwinięte) jednorazowo w trakcie kompilacji przez preprocesor, a wstawiane argumenty muszą być z góry określone w kodzie.

Dyrektywy definiujące oraz makra pozwalają bardzo mocno poprawić czytelność kodu programu. Na przykład podłączmy do linii PB0 i PB1 dwa ledy, zielony i czerwony, zasilane z V_{cc} przez rezystory ograniczające – czyli zapalane przy niskim stanie linii. Zamiast cały czas pamiętać o tej konfiguracji i każdorazowo używać uniwersalnych instrukcji opisanych powyżej, po prostu zdefiniujmy potrzebne operacje odpowiednio je nazywając (nazwy makr zwyczajowo pisze się dużymi literami):

```
#define LED_Z PB0
#define LED_CZ PB1
#define ZAPAL_Z  (PORTB &=~ _BV(LED_Z))
#define ZGAS_Z  (PORTB |= _BV(LED_Z))
#define PRZELACZ_Z  (PORTB ^= _BV(LED_
Z))
#define ZAPAL_CZ  (PORTB &=~ _BV(LED_CZ))
#define ZGAS_CZ  (PORTB |= _BV(LED_CZ))
#define PRZELACZ_CZ  (PORTB ^= _BV(LED_
CZ))
```

Wpis w kodzie np. *ZGAS_CZ;* jest jednoznaczny i czytelny, a dodatkowo zmniejsza ryzyko wystąpienia błędu przy wielokrotnym użyciu. Dodatkowym ułatwieniem stosowania jest system podpowiedzi AvrSide: klawisz **F1** przy karetce ustawionej na nazwie symbolu wyświetla okienko z jego deklaracją (**rys. 12**), natomiast skrót **SHIFT + F1** wyszukuje wszystkie miejsca występowania symbolu w kodzie.

Język C dostarcza jeszcze jeden sposób używania flag – są to pola bitowe. Zadeklarowanie flag jest

w tym przypadku niestety bardziej skomplikowane (szczegółowe informacje o stosowanych tu strukturach oraz uniach znajdziemy w każdym podręczniku C):

Najpierw definiujemy typ struktury z potrzebną liczbą jednobitowych flag (np. 8, wtedy flagi zajmują dokładnie jeden bajt:

```
typedef struct
{
 unsigned char Flag1:1;
 unsigned char Flag2:1;
 unsigned char Flag3:1;
 unsigned char Flag4:1;
 unsigned char Flag5:1;
 unsigned char Flag6:1;
 unsigned char Flag7:1;
 unsigned char Flag8:1;
} FlagBits;
```

(zamiast przy każdym polu wpisywać oddzielnie typ unsigned możemy w opcjach kompilacji zaznaczyć pozycję „zmienne z polami bitowymi domyślnie bez znaku").

Następnie definiujemy typ unii, w której jedną z możliwych zawartości jest nasza struktura z polami bitowymi a drugą (zamienną) zwyczajny bajt bez znaku:

```
typedef union
{
 FlagBits  Bits;
 uchar  Byte;
} Flags;
```

Jeśli teraz z programie zadeklarujemy zmienną typu Flags to możemy się odwoływać zarówno jednorazowo do całego bajtu poprzez pole Byte (może być to przydatne przy operacjach wspólnych np. szybkim wyzerowaniu wszystkich flag) jak i do poszczególnych flag (bitów) poprzez pola *Bits.FlagX*:

```
Flags MojeFlagi;
//...................
MojeFlagi.Byte=0;      // wyzerowanie
wszystkich flag
MojeFlagi.Bits.Flag1 = true;// ustawie-
nie pierwszej flagi
MojeFlagi.Bits.Flag5 = false;  // wyze-
rowanie 5. flagi
```

W takich zapisach znów dopomoże system podpowiedzi AvrSide wyświetlający po kropce listę wyboru dostępnych w unii/strukturze pól (**rys. 13**) (dla działania wymaga zapisania pliku po zadeklarowaniu unii lub struktury).

Dodatkowo możemy nazwać każdą flagę w czytelny sposób (*#define MOJA_FLAGA MojeFlagi.Bits.Flag1* itp.) i używać jej wtedy jak każdej innej zmiennej logicznej *(MOJA_FLAGA = true; if(MOJA_FLAGA) {})*. Jeśli zajrzymy do wygenerowanego kodu, to stwierdzimy, że pomimo skomplikowanych deklaracji jest on bardzo krótki i efektywny:

```
MojeFlagi.Bits.Flag3=true;
64:   80 91 78 00    lds   r24, 0x0078
68:   84 60    ori   r24, 0x04   ; 4
6a:   80 93 78 00    sts   0x0078, r24
```

Jak widać pozornie zawikłany mechanizm jest ostatecznie bardzo efektywny i przejrzysty w działaniu jednocześnie maksymalnie oszczędzając pamięć danych. Ma on jednak jeden – istotny w naszym środowisku uruchomieniowym – mankament: AvrStudio nie potrafi (przynajmniej w chwili pisania artykułu) obsługiwać pól bitowych. Jeśli więc zależy nam na podglądzie zmiennych logicznych w trakcie debugowania użyjemy ich w wersji tradycyjnej. Można też ewentualnie – zanim zespół Atmela wprowadzi odpowiednie udoskonalenia do AvrStudio – oglądać bezpośrednio pole Byte unii w zapisie heksadecymalnym, ale nie jest to zbyt wygodne.

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.