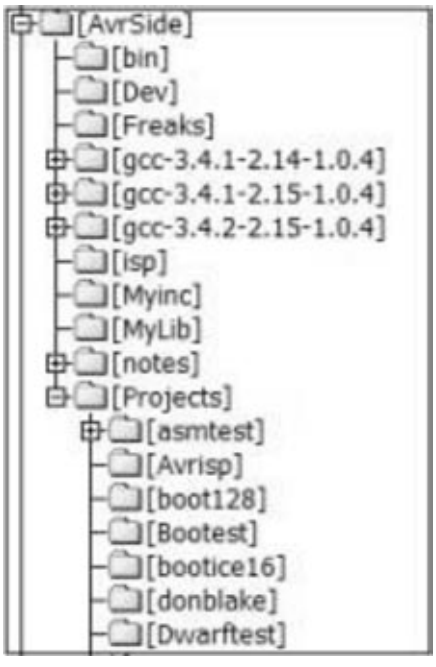
AVR-GCC: kompilator C mikrokontrolerów AVR, część 2

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.

Środowisko avrside

Uproszczone zintegrowane środowisko programistyczne (**Avr S**-imple **I**-ntegrated **D**-evelopment **E**-nvironment) powstało właśnie w celu łatwego rozpoczęcia nauki programowania mikrokontrolerów AVR Atmela z użyciem avr-gcc w systemie Windows. Zawiera edytor kodu źródłowego z kolorowaniem składni oraz systemem podpowiedzi i wyszukiwania. Pozwala na wygodne ustawianie w oknie dialogowym podstawowych opcji kompilacji. Uruchamia - podobnie jak *make* - kolejno potrzebne podprogramy avr-gcc i raportuje ewentualne błędy wykonania z możliwością ich lokalizacji w kodzie źródłowym. Jednocześnie zaś umożliwia dokładne zapoznanie się z wywołaniami avr-gcc i wynikiem pracy kompilatora, z czego będziemy na bieżąco korzystać.

Szczegóły obsługi AvrSide najlepiej chyba będzie objaśniać w trakcie realizacji kolejnych przykładów. Na początek warto jednak spojrzeć - podobnie jak w przypadku avr-gcc - na strukturę katalogów programu aby nie stanowił on tajemniczej „czarnej skrzynki". Drzewo katalogów (dotyczące instalacji AvrSide we własnym niezależnym od WinAvr folderze) jest przedstawione na **rys. 3**.



Rys. 3. Struktura folderów AvrSide

W podkatalogu [*bin*] umieszczony jest plik wykonawczy *AvrSide.exe* oraz pliki konfiguracyjne:

- *desktop.cfg* - zapis ogólnej konfiguracji środowiska (np. położenie i rozmiar okien),
- *autocpl.cfg* - plik tekstowy szablonów autokompletacji kodu,
- *libfunc.cfg* - plik tekstowy autokompletacji funkcji avr-libc,
- *ftd2xx.dll* - dodatkowo biblioteka FTDI niezbędna do uruchomienia AvrSide jeśli nie mamy w systemie zainstalowanego sterownika *direct* dla układów Ft8uxx.

W podkatalogu [*dev*] przechowywane są pliki **.pdf* firmowej dokumentacji mikrokontrolerów. Nazwane są zgodnie z typem mikrokontrolera ustalonym w AvrSide, co pozwala na ich otwieranie z poziomu menu pomocy programu.

Dalej widzimy omówione wcześniej foldery kompilatora avr-gcc w różnych wersjach. Folder [*isp*] zawiera pliki programatora AvrProgrammer Adama Dybkowskiego. Foldery [*Myinc*] oraz [*MyLib*] są przeznaczone na własne - bardziej uniwersalne i stosowane w większej liczbie projektów - pliki nagłówkowe oraz biblioteki.

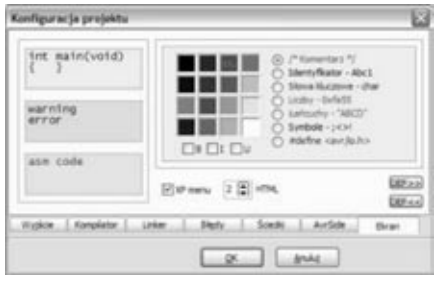
Folder [*Projects*] grupuje podkatalogi (a także drzewa podkatalogów) poszczególnych projektów. Każdy nowy projekt powinien być zakładany w oddzielnym podkatalogu (np. *\Projects\Nowy_projekt*) lub drzewie (np. *\Projects\Kurs\Przyklad_01*). Takie rozwiązanie umożliwia zachowanie porządku i przejrzystości we własnych archiwach projektowych, a także umożliwia korzystanie z wbudowanej przeglądarki projektów.

W każdym subfolderze projektu zapisujemy pliki źródłowe (**.c*, **.s* oraz **.h*) oraz plik konfiguracji projektu (*nazwa_projektu.gcp*) określający wszystkie opcje, ustawienia, ścieżki itp. użyte w projekcie. W tym samym subfolderze powstają wspomniane wcześniej pliki wynikowe działania kompilatora. W głównym folderze [*Projects*] znajdziemy też plik *lastwork.cfg* zapisujący dane projektu używanego w chwili zamykania AvrSide (co pozwala na natychmiastowe przywrócenie środowiska pracy podczas ponownego uruchomienia) oraz - tylko w przypadku używania AvrProgrammera - pliki *eeprom.hex* i *flash.hex* wykorzystywane przez ten programator.

Foldery [*Freaks*] i [*Notes*] nie są bezpośrednio związane z AvrSide ani avr-gcc - przechowuję tam różne notatki, noty aplikacyjne, wypisy z forum i listy avr-gcc itp. - w ten sposób nie „rozbiega" mi się to po całym dysku.

Tworzymy pierwszy testowy projekt

Po zainstalowaniu wszystkich wcześniej wymienionych składników na dysku pora na wypróbowanie działania środowiska. Wszystkie przytaczane przykłady dotyczą AvrSide ulokowanego w folderze *c:\AvrSide* i korzystającego z wersji avr-gcc z sub-



Rys. 4. Opcje ustawień ekranowych AvrSide

Rys. 5. Opcje ustawień konfiguracji plików wynikowych i typu mikrokontrolera

folderu *\AvrSide\gcc-3.4.2-2.15-1.0.4*, bez użycia pakietu WinAvr. Należy je więc każdorazowo dopasować do ustawionych u siebie lokalizacji.

Podczas pierwszego uruchomienia AvrSide otwiera w domyślnym położeniu okno pustego projektu NONAME z pojedynczą zakładką pustego pliku źródłowego NoName. Wywołajmy skrótem **Ctrl+J** wykaz szablonów kodu i wybierzmy „**mainmod** - szablon modułu głównego" - zostanie wpisany najprostszy, podstawowy program z pustą pętlą:

```
// główny moduł projektu
#define _MAIN_MOD_ 1
// pliki dołączone (include):
// dane:
// funkcje:
//====================
// funkcja main()
int main(void)
{
 // inicjalizacja
 // pętla główna
 while (1)
  {
  }
}
```

Teraz od razu wybierzmy lokalizację i nazwijmy plik oraz projekt:

– z menu „Plik" wybieramy komendę „Zapisz" albo „Zapisz jako" (w przypadku nowego pliku są one równoważne);
– w typowym windowsowym oknie dialogowym zapisywania pliku otwartym na folderze [*Projects*] tworzymy nowy podkatalog [*Kurs*] a w nim kolejny subfolder [*Przyklad-01*] (używamy zwykłych narzędzi Windows czyli menu kontekstowego, możemy też jednak przygotować sobie te foldery wcześniej w jakimkolwiek managerze plików);
– przejdźmy do [*Przyklad-01*] nazwijmy plik *main.c* (rozszerzenie jest dodawane domyślnie) i potwierdźmy; okno zapisu przełączy się samoczynnie na zachowywanie projektu - nazwijmy go np. *Test01* i zapiszmy;
– nasze nazwy zostały wprowadzone do projektu: plik na zakładce (*tab*) edytora, a projekt na belce tytułowej okna (w managerze plików możemy też od razy sprawdzić poprawność zapisu - subfolder *\Projects\Kurs\Przyklad-01* powinien zawierać dwa pliki: *main.c* oraz *Test01.gcp*);
– nazwy oczywiście możemy wybrać dowolnie, jedynym ograniczeniem jest aby plik projektu **.gcp* nie nazywał się tak samo jak jakikolwiek plik źródłowy, gdyż spowoduje to nadpisywanie plików wynikowych listingu.

Rys. 6. Wynik kompilacji przykładowego projektu

Zanim ruszymy dalej z kompilacją zajmijmy się przez chwilę skonfigurowaniem środowiska oraz ustawieniem opcji projektu. Okno główne umieszczamy na ekranie według potrzeb - jego pozycja będzie odtąd na stałe zapamiętana we wspomnianym wcześniej pliku ogólnej konfiguracji *desktop.cfg*. Następnie wybieramy z menu komendę „Projekt>>Ustawienia" i przełączamy widok kolejno na potrzebne pozycje:

1. „Ekran" (**rys. 4**) - tutaj ustawimy sobie:
   – kolory i style dla podswietlanych elementów składni kodu źródłowego (wybieramy w prawym panelu posługując się typowymi windowsowymi kontrolkami);
   – rodzaj i wielkość czcionki oraz kolor tła dla głównych okienek AvrSide (edytor, komunikaty błędów oraz podgląd asemblera); używamy do tego kontekstowego menu (prawy-klik) trzech małych paneli po lewej stronie;
   – wygląd menu głównego (tradycyjny lub zgodny z XP);
   – rozmiar czcionki w generowanym kodzie HTML;
   – ustawienia dotyczą naszego pojedynczego projektu, jeśli chcemy ich używać w każdym nowym projekcie zapiszmy je jako domyślne (w *desktop.cfg*) przy pomocy przycisku **DEF>>** (podobnie przyciskiem **DEF<<** możemy później przywrócić ustawienie domyślne w dowolnym inaczej skonfigurowanym projekcie);
2. „Ścieżki" - tu na razie ustawiamy tylko „namiar" na pliki wykonawcze narzędzi kompilatora, w naszym przykładzie będzie to *C:\AvrSide\gcc-3.4.2-2.15-1.0.4\bin* (jeśli zrobimy błąd i AvrSide nie znajdzie kompilatora zgłosi to przy próbie kompilacji);
3. „AvrSide" - wyłączmy na razie opcję „Samoczynnie zamykaj okno postępu kompilacji", będzie to pomocne w zapoznaniu się z przebiegiem kompilacji; pozostałe opcje można zostawić jako domyślne (starsze PC mogą także wymagać wyłączenia opcji „Fast reading" co umożliwi pracę z małym RAM-em, ale kosztem szybkości działania);
4. „Wyjście" (**rys.5**) - większość pozycji pozostawiamy na razie jako domyślne, sprawdźmy tylko czy są zaznaczone opcje: „Zbiorczy listing projektu" i „Debug format - dwarf-2".

Opcje kompilatora i linkera pozostawiamy bez zmian (będą oczywiście przedstawiane bardziej szczegółowo w dalszych przykładach).

Po zamknięciu dialogu opcji możemy zapisać zmiany w projekcie („Projekt>>Zapisz projekt") ale nie jest to bezwzględnie konieczne gdyż zostaną samoczynnie zachowane przy przełączaniu projektu albo zamknięciu AvrSide.

Po tych wszystkich wstępnych czynnościach możemy wreszcie wypróbować działanie kompilatora. Skrót klawiaturowy **F9** spowoduje wykonanie niezbędnego szeregu operacji (jest to odpowiednik pozycji menu „Projekt>>Make" - jednak nazwa ta nie ma nic wspólnego z klasycznym programem *make*), pokazanych w oknie postępu (celowo wyłączyliśmy jego samoczynne zamykanie aby je sobie spokojnie obejrzeć) - **rys. 6**.

Rys. 7. Okno konfiguracyjne sesji debuggera w AvrStudio

Jeśli teraz przełączymy się do managera plików znajdziemy w folderze naszego projektu szereg nowych plików utworzonych w trakcie działania kompilatora (sięgnijmy do rys. 1):
- *main.o:* plik relokowalny modułu main.c,
- *Test01.hex:* plik wynikowy kodu programu (*flash*),
- *Test01.map:* plik informacyjny konsolidatora,
- *Test01.lst:* plik assemblerowego listingu całego projektu,
- *Test01.elf:* plik obiektowy projektu,
- *Test01.smb:* wykaz symboli użytych w projekcie,
- *Test01.txt:* rejestracja komend wykonanych w trakcie kompilacji.

Zauważmy, że nie ma tu pliku pośredniego *main.s* - dla przyśpieszenia działania nie jest on zapisywany na dysku, a powstaje w chwilowym buforze pamięci operacyjnej. Pliki relokowalne **.o* zachowują nazwy odpowiadających modułów źródłowych, natomiast zbiorczym plikom wynikowym AvrSide nadaje nazwę projektu z odpowiednim rozszerzeniem.

W ostatnim z plików - *Test01.txt* - znajdziemy zapis kolejnych komend kompilatora odpowiadających pozycjom okienka postępu. Nie będziemy się w nie teraz na początku zagłębiać, ale wykorzystamy ten plik wielokrotnie podczas dalszego poznawania avr-gcc.

Z punktu widzenia elektronika - konstruktora najważniejszy jest oczywiście plik kodu **.hex*, który za pomocą dowolnego programatora ładujemy do mikrokontrolera. Odłóżmy to jednak na nieco później i najpierw obejrzyjmy trochę dokładniej wygenerowany kod. Skorzystamy w tym celu z wbudowanych udogodnień AvrSide. Skrót klawiaturowy **F7** otworzy nam okno podglądu kodu asm odtworzonego („zdeasemblowanego") z pliku relokowalnego bieżącego modułu (u nas mamy tylko jeden - main.o). Widzimy następującą treść (sekcja *.text* określa obszar pamięci programu *Flash*):

```
main.o:     file format elf32-avr
Disassembly of section .text:
00000000 <main>:
/====================
// funkcja main()
int main(void)
{
   0:  c0 e0          ldi r28, 0x00  ; 0
   2:  d0 e0          ldi r29, 0x00  ; 0
   4:  de bf          out 0x3e, r29  ; 62
   6:  cd bf          out 0x3d, r28  ; 61
 // inicjalizacja
 // pętla główna
 while (1)
   8:  ff cf          rjmp   .-2        ; 0x8
```

Funkcja main() ustawia na samym początku wskaźnik stosu (rejestry 0x3e,0x3d - na etapie pliku relokowalnego nie są jednak jeszcze wstawiane konkretne wartości) i od razu przechodzi do pustej nieskończonej pętli *while (1)*.

Jednak coś się tu nie zgadza: ten kod ma najwyżej kilka bajtów długości, a wynik kompilacji na pasku statusu pokazał nam 102 bajty. Aby znaleźć różnicę zajrzyjmy do pliku Test01.lst, który zawiera ostateczny kod po zakończeniu konsolidacji (do tego posłuży nam skrót **Ctrl+F7**):

```
Test01.elf:     file format elf32-avr
Disassembly of section .text:
00000000 <__vectors>:
   0:  12 c0          rjmp   .+36       ; 0x26
   2:  2b c0          rjmp   .+86       ; 0x5a
   4:  2a c0          rjmp   .+84       ; 0x5a
   6:  29 c0          rjmp   .+82       ; 0x5a
   8:  28 c0          rjmp   .+80       ; 0x5a
   a:  27 c0          rjmp   .+78       ; 0x5a
   c:  26 c0          rjmp   .+76       ; 0x5a
   e:  25 c0          rjmp   .+74       ; 0x5a
  10:  24 c0          rjmp   .+72       ; 0x5a
  12:  23 c0          rjmp   .+70       ; 0x5a
  14:  22 c0          rjmp   .+68       ; 0x5a
  16:  21 c0          rjmp   .+66       ; 0x5a
  18:  20 c0          rjmp   .+64       ; 0x5a
  1a:  1f c0          rjmp   .+62       ; 0x5a
  1c:  1e c0          rjmp   .+60       ; 0x5a
  1e:  1d c0          rjmp   .+58       ; 0x5a
  20:  1c c0          rjmp   .+56       ; 0x5a
  22:  1b c0          rjmp   .+54       ; 0x5a
  24:  1a c0          rjmp   .+52       ; 0x5a

00000026 <__ctors_end>:
  26:  11 24          eor r1, r1
  28:  1f be          out 0x3f, r1   ; 63
  2a:  cf e5          ldi r28, 0x5F  ; 95
  2c:  d4 e0          ldi r29, 0x04  ; 4
  2e:  de bf          out 0x3e, r29  ; 62
  30:  cd bf          out 0x3d, r28  ; 61

00000032 <__do_copy_data>:
  32:  10 e0          ldi r17, 0x00  ; 0
  34:  a0 e6          ldi r26, 0x60  ; 96
  36:  b0 e0          ldi r27, 0x00  ; 0
  38:  e6 e6          ldi r30, 0x66  ; 102
  3a:  f0 e0          ldi r31, 0x00  ; 0
  3c:  02 c0          rjmp   .+4        ; 0x42

0000003e <.do_copy_data_loop>:
  3e:  05 90          lpm r0, Z+
  40:  0d 92          st  X+, r0

00000042 <.do_copy_data_start>:
  42:  a0 36          cpi r26, 0x60  ; 96
  44:  b1 07          cpc r27, r17
  46:  d9 f7          brne   .-10       ; 0x3e

00000048 <__do_clear_bss>:
  48:  10 e0          ldi r17, 0x00  ; 0
  4a:  a0 e6          ldi r26, 0x60  ; 96
  4c:  b0 e0          ldi r27, 0x00  ; 0
  4e:  01 c0          rjmp   .+2        ; 0x52

00000050 <.do_clear_bss_loop>:
  50:  1d 92          st  X+, r1

00000052 <.do_clear_bss_start>:
  52:  a0 36          cpi r26, 0x60  ; 96
  54:  b1 07          cpc r27, r17
  56:  e1 f7          brne   .-8        ; 0x50
  58:  01 c0          rjmp   .+2        ; 0x5c

0000005a <__bad_interrupt>:
  5a:  d2 cf          rjmp   .-92       ; 0x0

0000005c <main>:

/====================
// funkcja main()
int main(void)
{
  5c:  cf e5          ldi r28, 0x5F  ; 95
  5e:  d4 e0          ldi r29, 0x04  ; 4
  60:  de bf          out 0x3e, r29  ; 62
  62:  cd bf          out 0x3d, r28  ; 61
 // inicjalizacja

 // pętla główna
 while (1)
  64:  ff cf          rjmp   .-2        ; 0x64
```

Teraz zgadza się wszystko: ostatni bajt kodu ma adres 0x65=101 - przy numeracji od zera długość wynosi 102 (przy okazji zauważmy, że gcc stosuje adresowanie bajtowe w przeciwieństwie do notacji Atmela używającej adresów dwubajtowych słów - dlatego adresy podawane w notach i dokumentacjach należy dla potrzeb avr-gcc mnożyć przez dwa). Dodatkowy kod spełnia kilka zadań związanych z inicjalizacją pracy programu (będziemy później wracać do tego tematu bardziej szczegółowo):

1. Sekcja <__vectors> ustawia obszar wektorów przerwań odpowiedni dla użytej kostki. Przerwanie pod adresem 0x0 (czyli reset) zawiera skok na koniec obszaru wektorów. Pozostałe wektory domyślnie (o ile nie została zdefiniowana obsługa danego przerwania) wskazują na błędne przerwanie.
2. Sekcja <__ctors_end> kończy obszar wektorów - w tym miejscu „ląduje" skok z przerwania spowodowanego resetem. Sekcja wykonuje kilka dodatkowych czynności: zeruje rejestr r1 (co jest wymagane przez standard wykorzystywania rejestrów w avr-gcc), zeruje rejestr stanu i ustawia rejestry stosu (zauważmy, że stos jest zatem ustawiany dwukrotnie - jest to jakaś „zaszłość rozwojowa" kompilatora, która będzie prawdopodobnie usunięta w dalszych wersjach).
3. Sekcje <__do_copy_data_xx> służą do załadowania wartościami zmiennych, które w programie zdefiniowaliśmy jako zainicjalizowane.
4. Sekcje <__do_clear_bss_xx> służą z kolei do wyzerowania wszystkich zmiennych nie zainicjalizowanych (które w standardzie C muszą mieć domyślnie wartość zero).
5. Sekcja <__bad_interrupt> stanowi (jak wspomniano wyżej) domyślną obsługę błędnego przerwania, jest to po prostu skok pod adres zero czyli na początek programu.

Cały ten blok kodu jest zawarty w relokowalnych plikach *crtxx.o* umieszczonych w podkatalogu (wróćmy do rys. 2) *\avr\lib* kompilatora. Komenda wywołania konsolidatora:

avr-gcc.exe -mmcu=atmega8 -Wl,--Map=*Test01.map*,--cref -o *Test01.elf main.o*

(znajdziemy ją we wspominanym już pliku *Test01.txt*) zawiera w sobie obowiązkowo opcję typu procesora. Na tej podstawie konsolidator dołącza odpowiedni plik *crtxx.o* oraz wybiera odpowiedni skrypt (ze skryptu pochodzi m.in. docelowa wartość rejestru stosu: 0x45F, która - jak łatwo sprawdzić w dokumentacji Atmega 8 - odpowiada końcowi obszaru pamięci SRAM).

Obecność bezparametrowej funkcji main(void) wynika z zasad języka C - każdy program musi ją mieć jako podstawową „ramkę" obejmującą wszystkie wewnętrzne działania. Jednak aplikacja działająca w małym mikrokontrolerze będzie się dość istotnie różnić od „zwykłego" programu komputerowego. Taki program jest uruchamiany przez system operacyjny, zakończenie funkcji main() jest równoznaczne z zakończeniem programu i powrotem do systemu. Wstawienie takiej jak u nas zamkniętej pętli jest kardynalnym błędem, gdyż uniemożliwia wyjście z main() co jest równoznaczne z „zawieszeniem się" aplikacji. Zapis musi wyglądać nieco inaczej (zazwyczaj wstawimy jeszcze jakieś oczekiwanie na klawisz, żeby zamknięcie nie nastąpiło natychmiast):

```
int main(void)
{ return 0; }
```

Natomiast w atmedze oczywiście nie ma żadnego nadrzędnego systemu operacyjnego, któremu można przekazać sterowanie. Wyjście z funkcji main() spowoduje po prostu opuszczenie obszaru kodu programu i utratę kontroli nad mikrokontrolerem - czyli znów kardynalny (chociaż zupełnie innej natury) błąd.

Spróbujmy skompilować nasz test w takiej wersji. W kodzie wynikowym czeka nas miła niespodzianka:

```
int main(void)
{
   5c:  cf e5          ldi r28, 0x5F  ; 95
   5e:  d4 e0          ldi r29, 0x04  ; 4
   60:  de bf          out 0x3e, r29  ; 62
   62:  cd bf          out 0x3d, r28  ; 61
 // inicjalizacja
 // pętla główna
 //while (1)
  //{
  //}
 return 0;
}
   64:  80 e0          ldi r24, 0x00  ; 0
   66:  90 e0          ldi r25, 0x00  ; 0
   68:  00 c0          rjmp   .+0         ; 0x6a

0000006a <_exit>:
   6a:  ff cf          rjmp   .-2         ; 0x6a
```

Jak widać avr-gcc przewidział taką sytuację i samoczynnie (po zwrocie w rejestrach r25,r24 deklarowanej wartości zero) dodał sekcję <_exit>, która znów jest zamkniętą pętlą. Oczywiście mikrokontroler „zaniemówi" w oczekiwaniu na reset, niemniej jego zachowanie pozostaje całkowicie przewidywalne.

Zwróćmy przy okazji uwagę, że w wersji „atmegowej" kompilator uwzględnia fakt nie opuszczania main() i nie generuje ostrzeżenia o braku zwracanej wartości (*control reaches end of non-void function*) nawet jeśli pomijamy końcową instrukcję return.

Powyższy przykład wskazuje też na konieczność zachowania dodatkowej ostrożności przy stosowaniu „pecetowych" umiejętności programowania w C dla potrzeb mikrokontrolerów - cały czas trzeba mieć świadomość, że przenosimy się do środowiska o radykalnie innych możliwościach i wymaganiach.

Pojawi się niechybnie wątpliwość, czy tak długi kod dla „nic-nie-robiącego" programu nie jest marnowaniem cennej pamięci *Flash* mikrokontrolera? No cóż - zawsze trzeba jakoś zapłacić za wygodę i standaryzację. Poza tym - pisząc w C będziemy i tak sięgać raczej po kostki o większych możliwościach aby nie przejmowac się ograniczonymi zasobami. Dla przykładowej ATmega8 inicjalizacja zajmuje ok. 1% dostępnej pamięci programu, co jest wartością nieznaczną. Dla Atmega128 z 16-krotnie większym Flashem jest to już zupełnie pomijalne. Natomiast w ATtiny13 z 1 kB programu oraz 64 B SRAM rzeczywiście C może się zachowywać jak przysłowiowy słoń w składzie porcelany. W tym przypadku wybór narzędzia (C czy asm) będzie podyktowany rzeczywistymi wymaganiami naszej konkretnej aplikacji.

Uruchamiamy sesję debuggera

Na koniec tego wstępnego rozdziału sprawdźmy jeszcze jak działa AvrStudio 4.10 w roli symulatora i debuggera. Z poziomu AvrSide po prostu wciskamy **F11** (albo używamy odpowiedniej komendy menu). Jednak za pierwszym razem zamiast startu AvrStudio zostaje wyświetlone okienko informacyjne ze wskazówkami dalszego postępowania. Zgodnie z nimi uruchamiamy ręcznie AvrStudio z „pustą" sesją a następnie:

- otwieramy plik *C:\AvrSide\Projects\Kurs\Przyklad-01\Test01.elf*,
- w wyświetlonym oknie konfiguracji (**rys. 7**) wybieramy platformę debugera - AVR Simulator,
- oraz wybieramy typ układu - Atmega 8 (może się zdarzyć, że wybór układu jest zablokowany - świadczy to o braku w systemie odpowiedniej wersji parsera XML, należy przeprowadzić update z witryny Microsoftu - znajdziemy o tym wzmiankę w opisie usterek w pomocy AvrStudio).

Po chwili projekt zostanie załadowany i w oknie edycji pojawi się nasz kod ze wskaźnikiem programu ustawionym na funkcji main(). Nasz skrajnie uproszczony projekt nie daje żadnych praktycznych możliwości debugowania, możemy jednak sprawdzić działanie: startu i wstrzymania wykonywania, resetu oraz pracy krokowej. Wybierzmy też sobie pasujący układ okien i schemat kolorowania składni.

Jeśli teraz zamkniemy AvrStudio parametry sesji zostaną zachowane w pliku *Test01_elf.aps* (ulokowanym w folderze naszego projektu), który posłuży do jej samoczynnego wznawiania. Ponowne użycie F11 z poziomu AvrSide spowoduje uruchomienie AvrStudio od razu z naszym projektem (w zależności od wydajności sprzętu może to chwilę potrwać). Natomiast przy działającym już Studiu efektem będzie po prostu przywołanie jego okna. Warto poświęcić chwilę czasu na przejrzenie możliwości, opcji i „klawiszologii" AvrStudio - to znakomicie ułatwi później analizę kolejnych przykładów.

Należy jednak cały czas mieć na uwadze, że najnowsze wersje AvrStudio wyposażone w możliwość bezpośredniego odczytu plików **.elf* (z pominięciem dodatkowych konwersji elf >> coff, które zawsze były piętą achillesową współpracy z avr-gcc) są cały czas rozwijane i dopracowywane - mogą się więc przytrafiać rozmaite błędy. Dlatego warto na bieżąco sprawdzać czy nie pojawiły się jakieś aktualizacje (chodzi głównie o poprawione wersje bibliotek, często przekazywane poprzez forum avrfreaks).

Jeśli wszystko poszło bez problemów, to mamy po tej części artykułu:

- przygotowane środowisko do tworzenia oraz symulacji i debugowania projektów,
- wstępne rozeznanie jak są rozmieszczone i jak współpracują jego poszczególne komponenty,
- orientację w typach plików używanych w trakcie budowy projektu,
- podstawowe informacje o zasadach działania kompilatora i uzyskiwanym wynikowym kodzie.

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

Niektóre przydatne linki

http://winavr.sourceforge.net - strona domowa WinAvr
http://www.avrfreaks.net - największe forum mikrokontrolerów AVR
http://www.atmel.com - firmowa strona producenta AVR
http://www.nongnu.org/avr-libc/ - strona domowa projektu avr-libc
http://gcc.gnu.org - strona główna kompilatora gcc
http://sources.redhat.com/binutils/ - strona narzędzi binutils
http://www.avrside.fr.pl - strona środowiska AvrSide
http://www.avrside.ep.com.pl - strona środowiska AvrSide
http://www.amwaw.edu.pl/~adybkows/elka/ispprog.html - programator ISP LPT Adama Dybkowskiego