AVR-GCC: kompilator C dla mikrokontrolerów AVR, część 1

Rozpoczynamy 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.

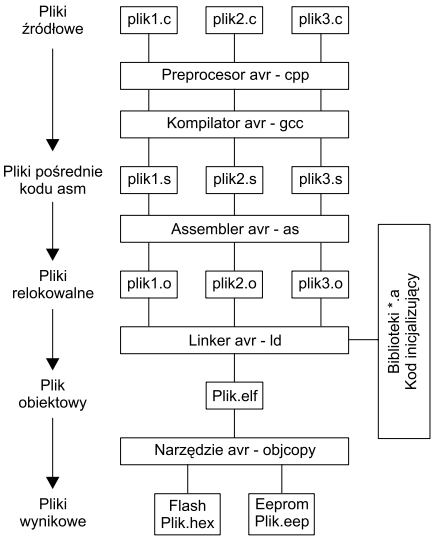
Zanim przejdziemy do konkretów na początek kilka słów uzasadniających taki właśnie wybór narzędzi.

Po pierwsze: po co w ogóle język wysokiego poziomu skoro mikrokontrolery świetnie programuje się w assemblerze, który daje pełną kontrolę nad kodem i zasobami procesora i pozwala na uzyskanie maksymalnej szybkości i zwięzłości? Otóż cały problem leży w skali. W przypadku małych układów z niewielkimi zasobami, wykonujących niezbyt złożone zadania (proste pomiary, sterowania czy transmisje) asembler rzeczywiście będzie całkowicie wystarczający (a czasem wręcz niezastąpiony). Gdy jednak program się nam rozrasta i komplikuje (obróbka większych ilości danych, bardziej złożone przeliczenia i konwersje, zaawansowane algorytmy sterowania itp.), prędzej czy później dochodzimy do progu, powyżej którego nasze dzieło staje się coraz mniej czytelne i coraz trudniejsze do opanowania. Nagle stwierdzamy, że dotychczasowe doświadczenia w asemblerze to za mało żeby szybko i skutecznie rozwiązać szerszy zakres problemów (taka zresztą była w ogóle geneza stworzenia języków wyższego poziomu). Tutaj przechodzimy do drugiego pytania.

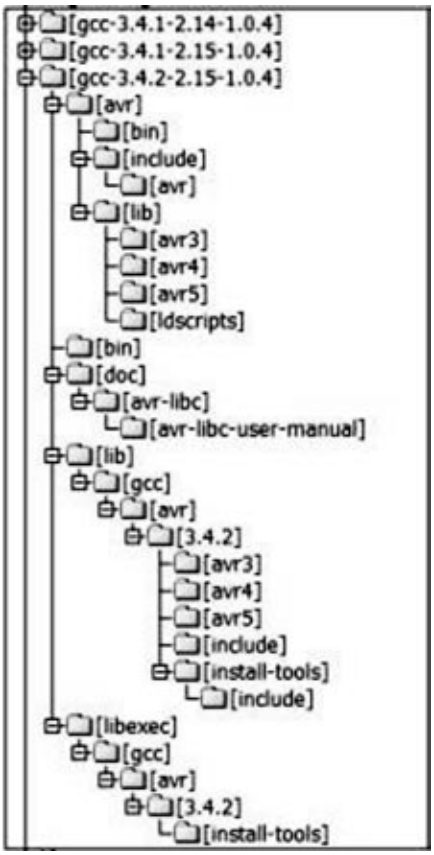
Po drugie: dlaczego właśnie C a nie np. BASIC czy Pascal? Otóż C był od początku projektowany jako język możliwie maksymalnie zbliżony do sprzętu i generujący kod niewiele odbiegający od samodzielnie pisanego w asemblerze. W połączeniu z optymalizatorem kodu (składnik każdego dobrego kompilatora) pozwala to na uzyskanie zaskakująco zwartego, krótkiego i szybkiego programu wynikowego. Oczywiście mamy również możliwość dopisania w czystym asemblerze fragmentów krytycznych czasowo (jak np. obsługa przerwań) jeśli nie zadowala nas kod generowany automatycznie. W ten sposób możemy bez problemu połączyć największe zalety obu sposobów programowania. Następna sprawa to przenośność. Znaczne fragmenty kodu (a szczególnie algorytmy, przeliczenia, konwersje itp. - czyli elementy nie korzystające bezpośrednio ze specyficznych zasobów i interfejsów danego mikrokontrolera) możemy łatwo zastosować w programie dla zupełnie innej kostki (praktycznie każda rodzina mikrokontrolerów posiada opracowany kompilator C - z innymi językami nie jest tak dobrze). Tu od razu przechodzimy do następnej zalety C: rozpowszechnienia. C jest od lat ogólnie przyjętym standardem programowania z czym wiąże się ogromna ilość dostępnych materiałów: bibliotek, przykładowych kodów, opisów, tutoriali, gotowych rozwiązań sprzętowo - programowych. W wielu przypadkach wystarczy dobrze poszukać w zasobach sieciowych żeby znaleźć prawie gotowe rozwiązania własnych zadań programowych.

Po trzecie: dlaczego kompilator avr-gcc (o którym krążą opinie, że jest niewdzięczny i trudny w konfiguracji i obsłudze) a nie jakieś inne rozpowszechnione narzędzie - jak np. CodeVision czy ICC? Jednym z koronnych argumentów jest oczywiście fakt, że avr-gcc jest bezpłatny. Ale to nie wszystko: jest to narzędzie dostępne dla wielu platform (więc bez problemu możemy przenosić się z naszymi projektami pomiędzy np. Windows a Linuksem). Przy tym avr--gcc jest produktem *open-source*, z czym wiąże się cały szereg udogodnień: mamy cały czas dostęp do najnowszych uaktualnień, pełnej informacji o wykrytych błędach, a także do ogromnych zasobów bardziej lub mniej zaawansowanego kodu tworzonych i oferowanych do swobodnego wykorzystywania, możemy też cały czas liczyć na wsparcie i podpowiedzi na aktywnie działających międzynarodowych forach, grupach i listach. A w dodatku - jak zaraz się przekonamy - przy użyciu dodatkowych narzędzi wspomagających avr-gcc staje się bardzo poręcznym i wygodnym w użyciu instrumentem. Można też dodać - już poza kontekstem stosowania w AVR - że gcc ma wersje (tzw. porty) dla wielu innych mikroprocesorów (np. MSP 430 czy ARM) więc raz opanowany znacznie ułatwi ewentualne „przesiadki".

Jednak żeby nie wyglądało to jak reklamowa laurka należy też powiedzieć o mankamentach. Główny z nich to nie do końca „rozpracowana" obsługa przez gcc nieciągłej przestrzeni adresowej AVR (dokładniej omówimy to w trakcie prezentacji przykładów). Zaleta jaką jest ciągły rozwój gcc może też stać się wadą w momencie wprowadzenia bardziej radykalnych zmian wymagających korekt we wcześniej działających projektach. Czasem pojawiają się drobniejsze błędy widoczne tylko w specyficznych sytuacjach (być może dlatego przeoczone w trakcie prac



Rys. 1. Przykładowy przebieg kompilacji kodu źródłowego



Rys. 2. Struktura folderów kompilatora avr-gcc

nad kompilatorem). Czy te wady są bardzo uciążliwe - ocenimy sami po pierwszych próbach programowania.

Nasze środowisko uruchomieniowe

Nasz „kursowy" zestaw do programowania AVR składa się z następujących pakietów narzędzi:

– właściwego kompilatora avr-gcc, który jest programem typu CLI (*command line interface*) i bez dodatkowego wsparcia musiałby być obsługiwany z poziomu konsoli tekstowej;
– graficznego środowiska eliminującego powyższą niedogodność - w tej roli występuje bezpłatne AvrSide pozwalające na szybkie i intuicyjne wykonanie podstawowych operacji (wersja PL);
– najnowszej, mocno ostatnio rozwiniętej i unowocześnionej wersji firmowego pakietu Atmela AvrStudio, który służy jako symulator i debugger do testowania naszych przykładowych projektów.

Całość jest zainstalowana na platformie Windows. Dopuszczalne są wersje 98, ME, XP, 2000. AvrSide nie da się uruchomić pod NT i 95 ze względu na brak obsługi magistrali USB (koniecznej dla wbudowanego w środowisko programatora). Wymogi sprzętowe nie są krytyczne - jednak rzecz jasna szybkość i komfort pracy będą mocno zależeć od mocy komputera. Prototyp opisywanego zestawu używany przy opracowaniu kursu został uruchomiony na platformie Athlon XP 2600+512 MB+Windows XP Pro PL, która zapewniła rzeczywiście wygodną i bezproblemową pracę.

Zauważmy, że nie ma na razie mowy o żadnym współpracującym układzie sprzętowym mikrokontrolera - wrócimy do tego tematu nieco później.

Skąd to wszystko wziąć? Ponieważ artykuły przygotowywane są ze sporym wyprzedzeniem, a potrzebne programy są wciąż rozwijane - najlepiej sięgnąć do źródłowych witryn projektów po najnowsze wersje. AvrSide znajdziemy na http://www.avrside.fr.pl albo na mirrorze http://www.avrside.ep.com.pl. Instalujemy najpierw podstawową wersję D5 a następnie zastępujemy plik wykonawczy *AvrSide.exe* najnowszym dostępnym. Aktualne szczegóły znajdziemy w dołączonych opisach i bezpośrednio na w/w stronach.

AvrStudio (w chwili pisania w wersji 4.10) jest udostępnione do bezpłatnego pobrania z firmowej witryny Atmela http://www.atmel.com.

Trochę więcej zastanowienia wymaga sam kompilator avr-gcc. Sztandarową dystrybucją avr-gcc dla Windows jest pakiet WinAvr Erica Weddingtona. Jednak nie należy utożsamiać avr-gcc z WinAvr jak to jest często w uproszczeniu podawane. WinAvr jest ogromnym zestawem wszelkich narzędzi *open-source* przydatnych w programowaniu AVR. Oprócz aktualnej wersji avr-gcc znajdziemy tam notatnik programisty wspomagający tworzenie projektów, symulator simulavr, debugger avr-gdb z graficznym interfejsem Insight, programator avrdude, Avarice - interfejs komunikacji pomiędzy avr-gdb a sprzętowym adapterem JTAG, szeroki wachlarz małych pomocniczych programów narzędziowych ogólnego stosowania, generator plików *makefile* MFile oraz ogromny zbiór pomocy, manuali i dokumentacji. WinAvr jest aktualizowane co kilka miesięcy i dostępne na http://winavr.sourceforge.net.

Wielką zaletą WinAvr jest wszechstronność dostarczonego materiału, który pozwala na dogłębne zapoznanie się z metodyką programowania, a także na indywidualny wybór najbardziej „pasującego" zestawu narzędzi. Jednak dla początkującego ta wszechstronność może zmienić się w poważną wadę: po prostu trudno się w tym wszystkim połapać. Dodatkowym utrudnieniem jest konieczność zapoznania się (przynajmniej w ogólnym zarysie) z zasadami działania i używania managera procesów *make*, stosowanego standardowo do uruchamiania kompilatora.

Ale w naszym środowisku ten nadmiar w niczym nie przeszkadza. Wykorzystamy po prostu tylko sam kompilator (AvrSide nie używa *make*), gdy zaś zechcemy poczytać dokumentację albo wypróbować inne techniki opracowania projektów - wszystko będzie pod ręką. AvrSide było zresztą początkowo dedykowane do współpracy z WinAvr i domyślnie jest instalowane w jego folderze. Dopiero później pojawiła się własna dystrybucja avr-gcc (także na stronie http://www.avrside.fr.pl). Obecnie mamy więc do wyboru dwie możliwości:

– instalacja kompletnego WinAvr a następnie AvrSide domyślnie w folderze głównym WinAvr;
– uproszczona instalacja samego AvrSide we własnym niezależnym folderze np. c:\AvrSide i uzupełnienie kompilatorem (właśnie tak jest skonfigurowane środowisko używane na potrzeby artykułu).

W obu przypadkach AvrStudio instalujemy całkiem niezależnie zgodnie z zaleceniami Atmela.

Zanim zabierzemy się do pisania pierwszych programów zapoznajmy się ogólnie z działaniem elementów środowiska programistycznego. Znakomicie ułatwi to dalszą pracę i analizę pierwszych przykładów. Można oczywiście na razie te zagadnienia przejrzeć tylko pobieżnie - będziemy wielokrotnie do nich wracać.

Jak działa avr-gcc

Avr-gcc określamy ogólnym mianem kompilatora - jednak w rzeczywistości jest to cały szereg współpracujących ze sobą narzędzi i bibliotek używanych w odpowiedniej kolejności i z potrzebnymi opcjami („właściwy" kompilator *avr-gcc*, zestaw narzędziowy *avr-binutils* oraz biblioteki dla rodziny AVR *avr-libc*). Celem jest przetworzenie kodu źródłowego C zapisanego w jednym lub wielu plikach projektu na wynikowe pliki: kodu wykonawczego wpisywanego do pamięci *Flash* wybranego

modelu mikrokontrolera oraz (ewentualnie) danych wpisywanych do wewnętrznej pamięci *EEPROM*. W trakcie powstaje także szereg pomocniczych plików zawierających rozmaite informacje potrzebne do debugowania oraz pozwalające ocenić efekty pracy kompilatora. Przykładowy przebieg przetwarzania jest pokazany na **rys. 1**.

Najpierw pliki źródłowe poddawane są działaniu preprocesora, który realizuje zmiany w kodzie nakazane wpisanymi przez nas dyrektywami:
- dołącza dodatkowe pliki z dyrektyw *#include*,
- wstawia definicje oraz rozwija makroinstrukcje z dyrektyw *#define*,
- uwzględnia odpowiednie fragmenty kodu z dyrektyw kompilacji warunkowej *#if #else #endif*.

Tak przygotowany kod poddawany jest właściwej kompilacji czyli przekodowaniu zapisu C na ciąg instrukcji asemblera (z przeprowadzeniem optymalizacji - automatycznego uproszczenia i skrócenia kodu), wynikiem tego są pośrednie pliki *.s.

Pliki asemblerowe podlegają następnie przetworzeniu na kod maszynowy (asemblacji). Powstałe pliki *.o są na razie tzw. relokowalne (przemieszczalne) - adresy zmiennych i funkcji nie są jeszcze konkretnie ustalone (pozostają nadal określone jedynie nazwami symbolicznymi użytymi przez nas w programie).

Pliki relokowalne są łączone (linkowane, konsolidowane) w następnej operacji - polega to właśnie na ulokowaniu wszystkich porcji kodu w obszarze pamięci programu oraz wyliczeniu i nadaniu symbolom określonych adresów. Jednocześnie zostają dołączone wszelkie niezbędne funkcje biblioteczne wykorzystywane przez nas (jawnie lub pośrednio) w programie, a także specyficzny dla danego typu mikrokontrolera plik kodu inicjalizującego.

Wynikiem powyższych zabiegów jest tzw. plik obiektowy (zazwyczaj nadaje mu się rozszerzenie .elf), który zawiera wszelkie informacje o projekcie (kod wynikowy, informacje dla debuggera, wykaz symboli, rozmiary poszczególnych sekcji kodu). Z informacji tych możemy korzystać według potrzeb dekodując potrzebne fragmenty za pomocą zbioru narzędzi binutils. Rysunek pokazuje tylko podstawowe zastosowanie - utworzenie plików wykonywalnych wpisywanych przy pomocy programatora do pamięci *Flash* oraz *EEPROM* mikrokontrolera.

Do tych - na razie bardzo skrótowych i ogólnych informacji - będziemy wielokrotnie wracać w trakcie omawiania przykładowych projektów. Na razie zobaczmy jeszcze jak są rozmieszczone poszczególne elementy kompilatora i gdzie szukać wymienionych wcześniej narzędzi.

Rys. 2 przedstawia drzewko folderów kompilatora. Nie wszystkie subfoldery są dla nas jednakowo istotne ale do niektórych będziemy często zaglądać.

Główny folder został nazwany na podstawie użytych wersji narzędzi (avr-gcc w wersji 3.4.2, binutils w wersji 2.15 i biblioteki avr-libc w wersji 1.0.4). W WinAvr powyższa nazwa głównego katalogu nie występuje - rolę tę pełni po prostu \WinAvr\, jednak zasadnicza struktura drzewa subfolderów pozostaje taka sama.

Dla użytkownika - programisty AVR najbardziej istotne są podkatalogi:
- *\bin* - zawierający zestaw bezpośrednio używanych programów narzędziowych avr-gcc i avr-binutils,
- *\avr\include* (oraz *\avr\include\avr*) - grupujący pliki nagłówkowe (*headers*) biblioteki avr-libc, w szczególności pliki opisujące zasoby poszczególnych kostek, do których często zaglądamy dla sprawdzenia np. nazw rejestrów czy wektorów przerwań,
- *\avr\lib\ldscripts* - zawiera skrypty sterujące pracą konsolidatora (linkera), w szczególności opisujące wielkości poszczególnych obszarów pamięci, jej podział na sekcje, adresy początkowe i końcowe,
- *\lib\gcc\avr\3.4.2\include* - znajdziemy tu ogólne pliki nagłówkowe gcc, nie powiązane bezpośrednio z kostkami AVR ale często używane (np. przy zastosowaniu zmiennych logicznych *bool*),
- *\doc\avr-libc\avr-libc-user-manual* - mieści aktualny opis funkcji bibliotecznych avr-libc oraz wiele istotnych szczegółowych informacji dotyczących różnych aspektów programowania (jest to pozycja obowiązkowa - tu zaglądamy prawie codziennie).

Pozostałe subfoldery zawierają wewnętrzne podprogramy i biblioteki gcc, wywoływane podczas kompilacji w tle bez naszego bezpośredniego udziału.

Zwróćmy uwagę na dość tajemnicze w pierwszej chwili oznaczenia: *avr3 - avr4 - avr5*. Wynikają one z przyjętego podziału szeregu kostek AVR na różne typy architektury związane z wielkością zasobów danego mikrokontrolera. Każdy typ posiada oddzielny zestaw bibliotek. Jednak nie musimy się tym kłopotać - na podstawie podanej w wywołaniu nazwy kostki konsolidator samoczynnie wybiera odpowiedni zestaw z właściwego subfolderu (więcej na ten temat powiemy przy opisach działania avr-ld).

Teraz pozostaje już tylko pytanie jak użyć tych wszystkich narzędzi dla uzyskania pożądanego efektu. Najbardziej podstawowym sposobem będzie kolejne wpisywanie potrzebnych komend w linii poleceń tekstowej konsoli. Taka metoda - chociaż dobra do przeprowadzenia eksperymentów edukacyjnych - jest zbyt uciążliwa i powolna przy praktycznym programowaniu - potrzebna jest nam automatyzacja całego procesu kompilacji. Tradycyjnym i bardzo rozpowszechnionym rozwiązaniem jest zastosowanie programu narzędziowego *make*. Mówiąc w wielkim skrócie jest to uniwersalny zarządca procesów, który wykonuje po kolei zadania określone podanymi mu przy wywołaniu zasadami (*rules*). Zasady te zapisujemy w określony sformalizowany sposób w tekstowych plikach *makefile* przygotowanych dla każdego projektu. *Make* jest bardzo wszechstronnym i potężnym narzędziem pozwalającym na praktycznie dowolne kształtowanie jego działania. Jednak ta zaleta potrafi stać się poważną wadą i przeszkodą przy rozpoczynaniu nauki programowania - musimy na wstępie opanować dodatkowy spory zasób wiadomości. Poza tym *make* jest narzędziem typowo tekstowym - nie stanowi to żadnej przeszkody dla miłośników Linuksa, ale przyzwyczajeni do interfejsów graficznych użytkownicy Windows mają prawo mieć inne zdanie (absolutnie nie mam zamiaru powracać tu do odwiecznego sporu o wyższość jednego systemu nad drugim - stwierdzam po prostu fakt). W naszym kursie realizowanym na platformie Windows wykorzystamy więc inne narzędzie. Oczywiście w miarę nabierania doświadczenia opanowanie *make* będzie również bardzo wskazane - ale to oddzielny temat i w obecnym cyklu artykułów nie będziemy się tym zajmować.

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