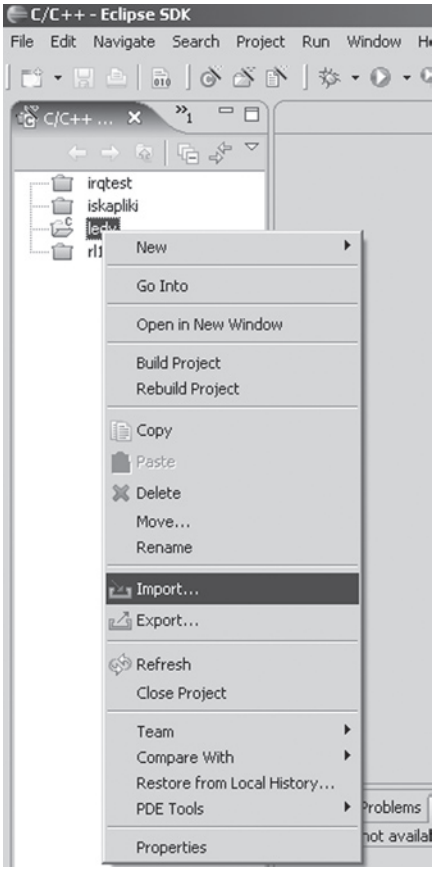
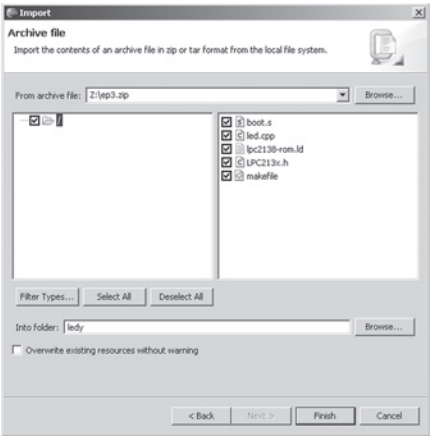
Mikrokontrolery z rdzeniem ARM, część 8
Pierwszy projekt

Przechodzimy do pierwszego projektu na ARM-y – będą to banalne efekty „choinkowe" na diodach świecących, ale w pełni ilustrujące sposób korzystania z możliwości środowiska Eclipse.

Pisząc programy dla mikrokontrolerów AVR z użyciem kompilatora *avr-gcc* musieliśmy stworzyć tylko plik *makefile* określający zależności kompilacji pomiędzy plikami, oraz pliki samego projektu. Pozostałe elementy niezbędne do prawidłowego przebiegu kompilacji (plik startowy oraz skrypty linkera) zawarte były w samym pakiecie *WinAVR*. Na podstawie opcji –m określającej typ procesora były automatycznie pobierane odpowiednie pliki startowe dostosowane do konkretnego modelu. Inaczej jest w przypadku wykorzystania pakietu gnuarm, który jest przeznaczony nie dla konkretnego typu mikrokontrolerów jednego producenta, ale dla różnych odmian pochodzących od różnych producentów. Jedyną ich cechą wspólną jest rdzeń ARM natomiast sposób inicjalizacji, peryferia, mapa pamięci mogą być zupełnie odmienne, dlatego pliki startowe musimy stworzyć samodzielnie. Pliki, jakie musimy stworzyć to: Skrypt linkera, w którym zawarte są informacje o umieszczeniu w pamięci poszczególnych sekcji danych oraz kodu programu. Plik startowy, w którym zawarty jest startowy kod inicjalizujący peryferia mikrokontrolera takie jak pętla PLL czy kontroler pamięci MAM oraz inicjalizujący obszary pamięci zgodnie ze standardem ANSI C/C++. Pisząc nowy projekt nie będziemy musieli tworzyć od nowa plików startowych, wystarczy, że stworzymy je jednorazowo dla danej rodziny mikrokontrolerów, a później będziemy je kopiować do nowego projektu. Pierwszym programem, jakim zajmiemy się na łamach niniejszego kursu będzie przykładowy program zapalający po kolei diody LED umieszczone w zestawie ZL6ARM (*www.kamami.pl*). Będzie on nam mógł później posłużyć jako wzorzec przy tworzeniu własnych projektów. Wykorzystywać będziemy pliki startowe oraz plik reguł *makefile* tak, aby za każdym razem nie tworzyć ich od nowa. Na początek opiszemy, w jaki sposób stworzyć najprostszy projekt w *Eclipse*, a następnie zaimportujemy do niego wszystkie pliki z przykładowego projektu błyskającego diodami, skompilujemy go, oraz zaprogramujemy pamięć Flash mikrokontrolera obserwując efekt działania programu. Po uruchomieniu Eclipse wybieramy z menu polecenie *File->New->Project*.



Rys. 20.

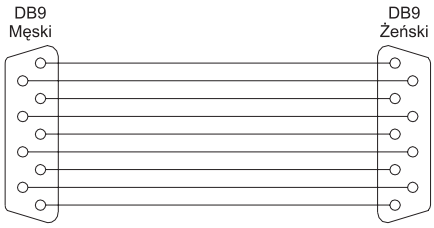


Rys. 21.

Pokaże nam się okno dialogowe, z którego wybieramy opcję: *Standard Make C++ Project*. Następnie klikamy przycisk *Next>*, pojawi się wówczas kolejne okno dialogowe. W polu tekstowym *Project Name* wpisujemy nazwę projektu np. *Ledy* i wciskamy klawisz *Finish*, co spowoduje utworzenie nowego pustego projektu. Kolejną czynnością jest import plików do projektu. Możemy tego dokonać klikając prawym przyciskiem myszy na otwartym projekcie (tak jak przedstawiono na **rys. 20**), a następnie wybrać polecenie *Import*.

Pojawi się okno dialogowe, z którego wybieramy opcję *Archive File*, wówczas zobaczymy kolejne okno, w którym klikamy przycisk *Browse* i wybieramy plik *ep3.zip* z przykładowym projektem, który można ściągnąć ze strony EP w dziale *Download*. Po prawej stronie pokaże się lista plików znajdujących się w archiwum (**rys. 21**).

Ponieważ wykorzystywać będziemy gotowy program pozostawiamy zaznaczone wszystkie pliki i wciskamy klawisz *Finish*. Gdybyśmy mieli zamiar tworzyć własny projekt powinniśmy odznaczyć plik *led.cpp*, który jest przykładowym programem mrugającym diodami LED. Pozostałe pliki są plikami niezbędnymi do przeprowadzenia

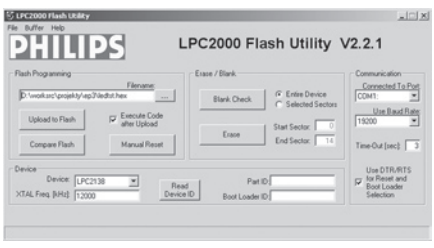


Rys. 22.

procesu kompilacji. Po tej czynności możemy przystąpić do kompilacji projektu, czego możemy dokonać wydając z menu polecenie *Project->Build All*. Jeżeli wszystko przebiegło poprawnie projekt zostanie skompilowany w wyniku, czego powstanie plik *ledtst.hex***.** Jeżeli wystąpił błąd podczas kompilacji, w zakładce *Problems* pojawi się opis błędu. Po kliknięciu na wybranym komunikacie zostaniemy przeniesieni linii, w której wystąpił błąd.

Bardzo ciekawą i przydatną funkcją Eclipse są inteligentne podpowiedzi, które podpowiadają programiście, jakie metody i pola znajdują się w danej klasie/strukturze. Po prawej stronie umieszczona jest zakładka *Outline*, w której w sposób graficzny przedstawiono poszczególne klasy, zmienne, funkcje definicje itp. Pozostało nam teraz zaprogramowanie mikrokontrolera, w tym celu za pomocą kabla łączymy port COM0 zestawu ZL6ARM z wybranym portem szeregowym komputera. Schemat kabla połączeniowego przedstawiono na **rys. 22**.

Po podłączeniu zestawu uruchamiamy program *LPC2000 Flash Utility*.W polu tekstowym *Filename* wpisujemy ścieżkę do pliku *ledtst.hex* a następnie wciskamy przycisk *Upload to Flash***.** Po chwili program zostanie załadowany i uruchomiony. Na diodach LED (D4…D11) zestawu ZL6ARM powinien ukazać się efekt punktu świetlnego. Przy pierwszym uruchomieniu programu należy skonfigurować program *LPC2000 Flash Utility***.** Na rysunku



Rys. 23.

Tab. 4. Pliki projektu i ich funkcje

Nazwa pliku	Przeznaczenie
boot.s	Plik startowy inicjalizujący układy peryferyjne mikrokontrolerów LPC213x oraz inicjalizujący pamięć RAM zgodnie z wymaganiami standardu ANSI C/C++
lpc2138-rom.ld	Plik dla linkera określający, w jaki sposób rozmieścić poszczególne segmenty w pamięci. Jest on skonfigurowany tak, aby program i dane stałe były umieszczone w pamięci Flash, natomiast zmienne umieszczone zostaną w pamięci RAM.
lpc213x.h	Plik nagłówkowy, zawierający definicję rejestrów mikrokontrolerów LPC213x
makefile	Plik konfiguracyjny dla narzędzia make służącego do określenia zależności pomiędzy plikami projektu
led.cpp	Przykładowy program mrugający diodami (efekt węża świetlnego)

rys. 23 przedstawiono proponowany sposób konfiguracji programu.

Należy wybrać typ mikrokontrolera, wpisać częstotliwość kwarcu mikrokontrolera oraz zaznaczyć opcję używania linii DTR/RTS służących do uruchamiania i zerowania mikrokontrolera. W przypadku zmiany pliku HEX (np. w wyniku dopisania fragmentu kodu) nie musimy po każdej kompilacji ponownie wybierać nazwy pliku, gdyż przed każdym poleceniem załadowania pamięci Flash jest on odczytywany od nowa, co jest istotną zaletą programu.

Pliki projektu

W **tab. 4** przedstawiono pliki, jakie znalazły się w naszym pierwszym projekcie oraz funkcje, jakie pełnią.

Przebiegiem kompilacji projektu zarządza narzędzie *GNU make*, które na podstawie pliku konfiguracyjnego *makefile* określa na podstawie odpowiednich zależności, co i w jakiej kolejności ma być kompilowane. Poza tym *make* kompiluje tylko te pliki, które uległy zmianie. Tworząc plik makefile musimy zbudować odpowiednie zależności i reguły określające, w jaki sposób powstać ma plik wynikowy. Przed zapoznaniem się z budową makefile najpierw przedstawimy przebieg procesu kompilacji naszego pierwszego projektu (**rys. 24**).

Poprzez kompilację plików *led.cpp* oraz *lpc213x.h* powstaje plik wynikowy *led.o*, natomiast z pliku *boot.s* tworzony jest plik wynikowy *boot.o*. Z plików *led.o boot.o* na podstawie skryptu *lpc2138-rom.ld* powstaje ostateczny plik wynikowy *ledtst.elf*., z którego za pomocą programu konwertującego tworzony jest plik *ledtst.hex*. W pliku *makefile* będziemy zapisywać przebieg procesu kompilacji, co przy wykorzystaniu standardowego szablonu jest bardzo proste. Poniżej przedstawiono *makefile* naszego projektu. Nie będziemy się tutaj zagłębiać w szczegółowy opis narzędzia *make*. Zainteresowanych odsyłam do literatury lub Internetu. Przedstawimy tylko fragmenty, które będziemy musieli zmieniać tworząc własne projekty:

```
#tutaj wpisz nazwe pliku hex
TARGET   = ledtst
```

W powyższej linii wpisujemy nazwę pliku wynikowego HEX, jaki zostanie wygenerowany w wyniku kompilacji:

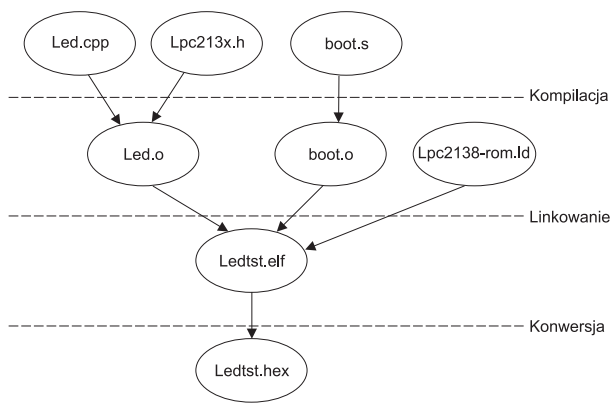
```
CFLAGS  = -Os -fpack-struct -mcpu=ar-
m7tdmi -Wall -gstabs
```

Polu *CFLAGS* przypisujemy opcje kompilatora, jakie będą użyte przy tworzeniu projektu.

Flaga –Os określa stopień optymalizacji kompilatora na najmniejszy rozmiar pliku wynikowego.

Inna często wykorzystywana opcja to *–O3*, generująca najszybszy kod wynikowy.

Opcja *–fpack-struct* zakazuje kompilatorowi wyrównywania struktur pustymi bajtami, a więc



Rys. 24.

struktura ma dokładnie taką postać jak jest zadeklarowana.

Opcja *–mcpu=arm7tdmi* określa typ procesora.

Opcja *–Wall* nakazuje kompilatorowi generowanie szczegółowych ostrzeżeń kompilacji.

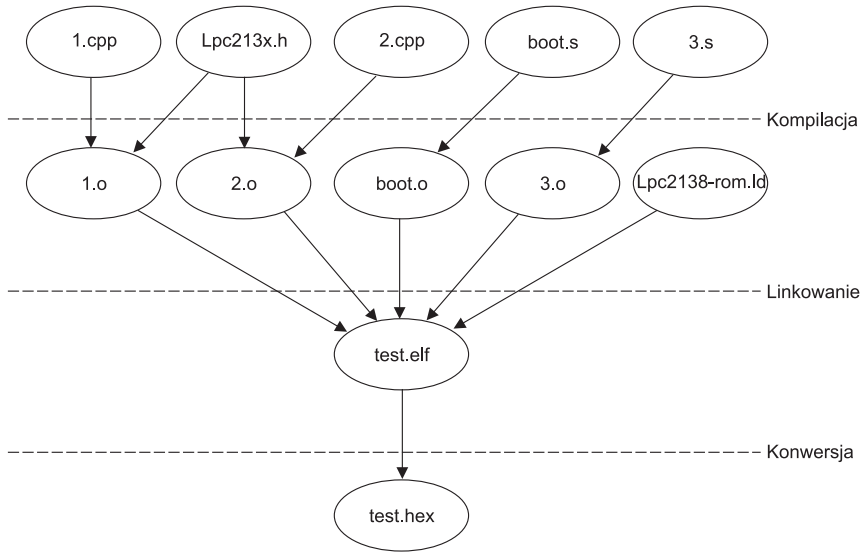
Opcja *–gstabs* nakazuje generowanie informacji dla debugera.

```
#pliki zrodlowe
SRC = led.cpp
#pliki assemblerowe
ASRC = boot.s
```

W polu *SRC* wpisujemy listę wszystkich plików źródłowych **.cpp* występujących w projekcie oddzielając je spacjami natomiast w polu *ASRC* wpisujemy listę plików asemblerowych **.s*:

```
#Zaleznosci pomiedzy plikami w C
boot.o: boot.s
led.o:  led.cpp lpc213x.h
#zaleznosci pomiedzy plikami konco-
wymi
$(TARGET).elf: boot.o led.o lpc2138-
rom.ld
```

Powyższe linie opisują wcześniej wspomniane zależności pomiędzy plikami źródłowymi oraz wynikowymi. Odczytujemy je w sposób następujący: plik *boot.o* zależy od (powstaje z) pliku *boot.s*. Plik *led.o* zależy od plików *led.cpp* oraz *lpc213x.h*. Plik wynikowy `$(TARGET).elf` zależy od plików *boot.o led.o lpc2138-rom.ld*. W ten właśnie prosty sposób możemy budować nawet bardzo skomplikowane projekty bez obawy o prawidłowy przebieg kompilacji. Tworząc własne konfiguracje na podstawie powyższego szablonu musimy pamiętać, aby poprawnie zapisać wszystkie zależności, gdyż w przeciwnym przypadku program wynikowy może działać niepoprawnie.



Rys. 25.

Lucjan Bryndza SQ7FGB, EP
lucjan.bryndza@ep.com.pl