Mikrokontrolery z rdzeniem ARM, część 9
Pierwszy projekt

W tej części kończymy prezentację sposobu przygotowania narzędzi do kompilacji pierwszego projektu. Przedstawione informacje mają charakter uniwersalny i będą pomocne także w przypadku kompilowania innych projektów, także składających się z wielu plików źródłowych.

Gdy kompilowany projekt zawiera wiele plików (jak na przykład przedstawiony na rys. 25 w EP7/2006) *makefile* budujemy w sposób przedstawiony poniżej.

W linii:

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

wpisujemy nazwę *test*, ponieważ plikiem, jaki chcemy otrzymać jest *test.hex*. W linii SRC wpisujemy nazwę wszystkich plików źródłowych napisanych w języku C/C++. Natomiast w linii ASRC wpisujemy listę wszystkich plików napisanych w asemblerze. Tak, więc w naszym przypadku linie te będą wyglądać w sposób następujący:

```
#pliki zrodlowe
SRC = 1.cpp 2.cpp
#pliki assemblerowe
ASRC = boot.s 3.s
```

Ostatnią czynnością będzie stworzenie zależności pomiędzy plikami, mające doprowadzić do otrzymania pliku wynikowego. Dla naszego przykładu zależności będą wyglądać następująco:

```
#Zaleznosci pomiedzy plikami w C
boot.o: boot.s
3.o: 3.s
1.o: 1.cpp lpc213x.h
2.o: 2.cpp lpc213x.h
#zaleznosci pomiedzy plikami konco-
wymi
$(TARGET).elf: boot.o 1.o 2.o 3.o
lpc2138-rom.ld
```

W przypadku, gdy projekt zawiera większą liczbę plików źródłowych sposób postępowania przy tworzeniu *makefile* jest analogiczny.

Kolejnym plikiem wykorzystywanym w projekcie jest skrypt linkera *lpc2138–rom.ld*, który przypisuje poszczególne segmenty kodu i danych generowanych przez kompilator w odpowiednie miejsca pamięci mikrokontrolera. W **tab. 5** przedstawiono poszczególne segmenty tworzone przez kompilator *gcc*.

W przypadku mikrokontrolerów LPC213x/214x skrypt możemy napisać w taki sposób, aby kod programu i dane były umieszczane w pamięci Flash, natomiast pozostałe dane umieszczane będą w pamięci RAM. Skrypt możemy napisać również tak, aby kod programu i dane znalazły się w pamięci RAM, jednak taka konfiguracja może być przydatna najwyżej do testowania niewielkich programów. Plik rozpoczyna się od definicji obszarów i adresów pamięci RAM i ROM:

```
/* Konfiguracja pamieci */

MEMORY
{
  CODE (xr) : ORIGIN = 0x00000000,
LENGTH = 512K
  DATA (rw) : ORIGIN = 0x40000000,
LENGTH = 32K
}
```

Sekcja MEMORY zawiera opis konfiguracji pamięci. Nazwie CODE przypisano obszar pamięci Flash mikrokontrolera, który znajduje się od adresu 0 (ORIGIN = 0x00000000), a jego wielkość określona jest na 512 kB (LENGTH = 512K). Ponieważ pamięć Flash jest tylko do odczytu przypisano jej atrybuty: wykonywalny oraz tylko do odczytu ((xr)). Nazwie DATA przypisano obszar pamięci RAM mikrokontrolera znajdujący się od adresu 0x40000000 (ORIGIN = 0x40000000), a jego wielkość określono na 32 kB (LENGTH = 32K). Obszarowi pamięci RAM przypisano atrybuty: zapis oraz odczyt ((rw)). Obszar pamięci został tutaj skonfigurowany dla mikrokontrolera LPC2138. W przypadku, gdy będziemy tworzyć program dla innych mikrokontrolerów np. *LPC2131* wystarczy zmienić parametr *LENGTH*. Dalszej części skryptu konfiguracyjnego nie będziemy musieli zmieniać. Znajdują się tam przypisania odpowiednich sekcji do obszarów pamięci np.:

```
/* sekcja .rodata  zawiera dane sta-
le */
  .rodata :
  {
   *(.rodata)
   *(.rodata.*)
   *(.gnu.linkonce.r.*)
  } >CODE
```

Do obszaru pamięci CODE (czyli pamięci Flash) przypisana jest sekcja .rodata zawierająca dane stałe, które w języku C są zadeklarowane jako const.

Do omówienia pozostał nam, jeszcze plik *led.cpp*, który zawiera program generujący efekt węża świetlnego na diodach LED zestawu ZL6ARM. Pomimo, iż zastosowanie języka C++ do problemu poruszanego w przykładzie może być traktowane jako przerost formy nad treścią, to zdecydowano się na taki krok, gdyż w ogólnym przypadku jest to język znakomicie nadający się do pisania programów na mikrokontrolery ARM. Na początku definiujemy porty, do których są podłączone diody LED:

```
//Definicja LEDow
#define LEDS (0xFF<<16)
#define LEDDIR IO1DIR
#define LEDSET IO1SET
#define LEDCLR IO1CLR
```

Kod generujący efekt węża świetlnego zaimplementowano

Tab. 5. Segmenty wynikowe tworzone przez kompilator gcc

Nazwa segmentu	Opis
.text	Segment zawierający kod programu
.rodata	Segment zawierający dane tylko do odczytu
.ctors	Segment zawierający kody konstruktorów C++
.dtors	Segment zawierający kody destruktorów C++
.data	Sekcja zawierająca dane zainicjalizowane
.bss	Sekcja zawierająca dane nie zainicjalizowane

w klasie *CShiftLed*, zawierającej metodę *Run()*, która powinna być wywołana w pętli głównej programu. Klasa jest rozszerzeniem pojęcia struktury z języka C. Posiada ona w sobie zmienne, które tutaj dla odróżnienia nazywa się polami, ale dodatkowo posiada w sobie funkcje zwane metodami, które operują na polach. Metody są po to, żeby można było chronić dane i wykonywać operacje na danych chronionych, oraz ułatwić operacje na polach tylko tej danej klasy w danym obiekcie. Metoda, jest to prawie to samo, co funkcja, z tym, że ma ona dostęp do pól klasy. Konstruktor klasy jest to specjalna metoda, która jest wywoływana zawsze, gdy klasa jest inicjowana. Konstruktor jest potrzebny do przydzielenia polom wartości początkowych lub wykonania innych czynności koniecznych przy tworzeniu obiektu klasy. Konstruktor i destruktor nie zwraca żadnej wartości. Nazwa konstruktora jest dokładnie taka sama jak nazwa klasy. Destruktor jest wywoływany, gdy klasa nie będzie więcej używana i jest ona niszczona. Jest on po to, żeby np. zwolnić pamięć przydzieloną dynamicznie. W klasach występują stopnie ochrony danych określające dostępność danego pola lub metody. Stopień *private*, jest największym stopniem ochrony. Dostęp do metod i pól chronionych tym stopniem jest ograniczony tylko do metod będących częścią tej klasy, co oznacza, że są one niedostępne na zewnątrz. Natomiast stopień ochrony *public* daje swobodny dostęp z dowolnego miejsca do metod i pól znajdujących się pod kontrolą tego modyfikatora. Implementację klasy przedstawiono na **list. 4**.

W konstruktorze klasy *CLedShift* linie P1.16...P1.24 portu P1 są ustawiane jako wyjściowe. Pole *mShift* zadeklarowane jako składowa prywatna klasy, przechowuje informację o tym, która z 8 diod LED ma zostać zapalona. Metoda *Run()* powinna być wywoływana cyklicznie w pętli głównej programu. Realizuje ona cykliczne przesuwanie pola *mShift* o jeden bit w lewo oraz przepisanie jego do rejestru portu P1, co w efekcie powoduje wyświetlenie stanu bitów pola *mShift* na diodach LED zestawu ZL6ARM. Na zakończenie cyklu wywoływana jest pętla opóźniająca, tak abyśmy mogli dostrzec zmianę stanu diod. Warto tutaj zwrócić uwagę na licznik pętli opóźniającej, który zlicza do miliona. Widać tutaj jak dużą mocą obliczeniową dysponuje zastosowany mikrokontroler.

List. 4. Implementacja klasy (program migający diodą LED)

```
class CLedShift
{
public:
  //Konstruktor klasy
  CLedShift()
  {
     //Piny jako wyjscia
     LEDDIR |= LEDS;
  }
  //Metoda Run
  void Run()
  {
     //Pierwsza dioda
     mShift = 1;
     for(int i=0;i<8;i++)
     {
        //Zapal wybrana diode
        LEDSET = mShift << 16;
        //Zgas pozostale diody
        LEDCLR = ~mShift << 16;
        //Petla opozniajaca
        for(int d=0;d<1000000;d++);
        //Przesun bit o 1
        mShift <<= 1;
     }
  }
private:
  //Zmienna przechowujaca biezacy
LED
  unsigned char mShift;
};
```

```
//Funkcja glowna main
int main(void)
{
  //Klasa diod swiecacych
  CLedShift led1;
  //Petla glowna
  while(1)
  {
     //Wywolanie cyklicznie metody
RUN
     led1.Run();
  }
}
```

W funkcji *main()* tworzony jest obiekt klasy *CShiftLed*, a następnie w pętli nieskończonej wywoływana jest metoda *Run()* tworząca efekt węża świetlnego.

Mam nadzieję udało mi się czytelnikom przybliżyć, chociaż w stopniu podstawowym procedurę instalacji oraz posługiwanie się środowiskiem *Eclipse*. Nic jednak nie jest w stanie zastąpić samodzielnych prób i eksperymentów, dlatego do nich gorąco zachęcam. W kolejnych odcinkach zajmiemy się poszczególnymi układami peryferyjnymi mikrokontrolerów LPC213x/214x.

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