Kurs AVR – lekcja 1

Po skompletowaniu narzędzi przyszedł czas na napisanie i uruchomienie pierwszego programu. Aby nasze programy mogły działać, musimy odpowiednio podłączyć nóżki mikrokontrolera. W przypadku mikrokontrolerów AVR nie jest to trudne. Przede wszystkim podłączamy zasilanie 5V do wyprowadzeń GND oraz VCC i AVCC (**rysunek 1**). Płytka testowa AVT-3510 ma już końcówki zasilające odpowiednio podłączone.

Każdy mikrokontroler ma jakieś źródło sygnału zegarowego. ATmega32 ma wewnętrzny generator RC o dokładności 3% (1% po dodatkowej kalibracji), który może służyć jako źródło sygnału zegarowego dla mikrokontrolera. Znacznie większą dokładność zapewnia rezonator kwarcowy (dołączony do nóżek XTAL1 i XTAL2 wraz z kondensatorami 12–22 pF). W naszej płytce testowej do mikrokontrolera podłączony jest rezonator o częstotliwości 16MHz. Sam fakt podłączenia rezonatora nie znaczy jeszcze, że mikrokontroler będzie z niego korzystał. O tym, czy mikrokontroler korzysta z rezonatora, decydują bity konfiguracyjne mikrokontrolera (fuse bits).

Za pomocą przycisku S1 możemy zresetować mikrokontroler. Na płytce testowej przycisk ten nosi nazwę RESET.

W naszych ćwiczeniach wykorzystywać będziemy wyprowadzenia mikrokontrolera umożliwiające jego programowanie poprzez ISP lub JTAG.

Rysunek 1 przedstawia schemat z diodą świecącą LED1 używaną w prezentowanym dalej ćwiczeniu. Na płytce testowej diody świecące wraz z odpowiednimi rezystorami dostępne są na złączu LED. Wystarczy więc połączyć przewodem pin PA0 złącza PORTA z pinem 1 złącza LED. W ćwiczeniu tym obwody rezonatora kwarcowego nie będą używane.

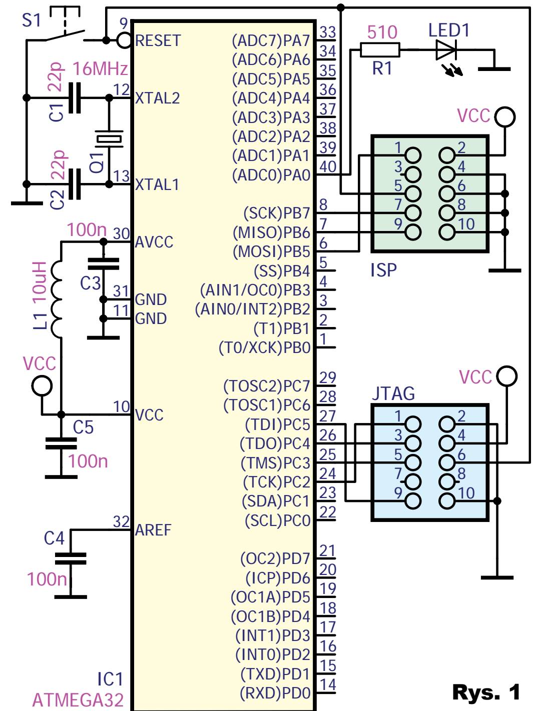
Tworzenie projektu

Pisanie programu zaczynamy od stworzenia projektu w Atmel Studio. Po uruchomieniu środowiska, na stronie startowej klikamy *New Project...* Ponieważ będziemy pisać w języku C i interesuje nas stworzenie programu, a nie biblioteki, wybieramy *GCC C Executable Project* (**rysunek 2**).

Uwaga! Oryginały wszystkich rysunków – zrzutów są dostępne w Elportalu wśród materiałów dodatkowych do tego numeru.

W polu na dole podajemy nazwę dla projektu oraz katalog, w którym zostanie utworzony podkatalog z naszym projektem. Po kliknięciu OK ukaże się nam kolejne okno, w którym wybieramy mikrokontroler, na który będziemy pisać program (**rysunek 3**). W naszym przypadku będzie to ATmega32, więc wybieramy go z listy. Po jego zaznaczeniu z prawej strony listy mikrokontrolerów wyświetli się link do karty katalogowej oraz lista programatorów obsługiwanych przez Atmel Studio dla wybranego mikrokontrolera. Kliknięcie OK powoduje wygenerowanie projektu (**rysunek 4**).

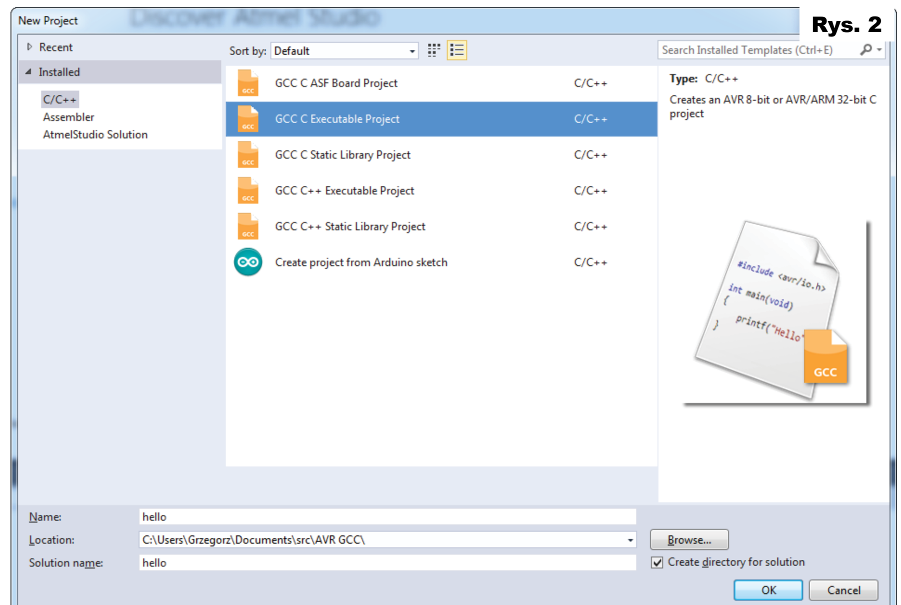
Znajduje się w nim jeden plik z kodem źródłowym, zawierającym funkcję ***main***. Jest też nieskończona pętla ***while***, w której należy umieścić własny kod. To tylko szablon, ale praktycznie każdy program na mikrokontroler ma podobną strukturę. Funkcja ***main***, zgodnie z założeniami języka C, jest głównym blokiem kodu źródłowego. To od niej zaczyna się wykonywanie programu i to ona uruchamia ewentualne inne funkcje. A po co pętla ***while***? Układy elektroniczne z reguły działają ciągle, bez przerwy. Mikrokontroler ma być stale gotowy na spływające do niego dane i reagować na zdarzenia takie jak upływ zadanego czasu czy też zmiana napięć na jego wyprowadzeniach. Naturalne jest więc wprowadzenie głównej pętli programu. Przed nią natomiast zwykle umieszcza się instrukcje inicjalizujące podsystemy mikrokontrolera.



Rys. 1

Pierwszy program

Tradycyjnie pierwszy program komputerowy wyświetla napis „Hello World!". W przypadku mikrokontrolerów sprawa jest nieco inna, ponieważ nie zawsze podłączony jest jakiś wyświetlacz. W świecie mikrokontrolerów odpowiednikiem „Hello World!" jest miganie diodą LED. U nas będzie to pierwszy (czyli noszący numer 0) pin portu A w naszym mikrokontrolerze ATmega32. Można wybrać inny pin i inny port, ale trzeba pamiętać, że niektóre piny są wykorzystywane do programowania mikrokontrolera i może powodować to niechciane efekty. Dlatego użyjemy portu A.



Rys. 2

TOP WWW SKLEP.SEMICON.COM.PL
- podzespoły elektroniczne - montaż SMT - materiały dla elektroniki
- szablony laserowe - diody laserowe - taśmy i kleje 3M

Obsługa portów wejścia/wyjścia

Domyślnie piny/nóżki portów mikrokontrolerów AVR są skonfigurowane jako wejścia. Żeby migać diodą musimy więc dany pin skonfigurować jako wyjście. Konfigurację portów oraz innych peryferii mikrokontrolera wykonujemy poprzez ustawianie odpowiednich bitów w rejestrach konfiguracyjnych tychże peryferii. Za to, czy piny portu A są wejściami, czy wyjściami, odpowiedzialny jest rejestr DDRA. Analogicznie są dostępne rejestry DDRB, DDRC i DDRD dla pozostałych portów układu ATmega32. Każdy z 8 bitów tego rejestru odpowiada jednemu pinowi portu A. Bit 0 konfiguruje pin 0, bit 1 konfiguruje pin 1 itd. Bit ustawiony na 0 powoduje, że odpowiedni pin staje się wejściem, bit ustawiony na 1 konfiguruje pin jako wyjście. Chcąc ustawić nóżkę 0 portu A jako wyjście, potrzebujemy do rejestru DDRA wpisać liczbę 00000001b (binarną), czyli po prostu 1. Na początku funkcji ***main***, przed główną pętlą, możemy więc napisać:

```
DDRA = 1;
```

Taki zapis jest poprawny, jednak nie jest zbytnio czytelny. Dobrze, aby kod był samokomentujący się, żeby patrząc na niego po pewnym czasie, nie trzeba było zgadywać, jaką mieliśmy intencję pisząc dany fragment. A nie chodzi nam w powyższej linijce o wpisanie tej czy innej liczby, tylko o konfigurację nóżki procesora. Dobrze byłoby to jakoś zaznaczyć. Kolejne bity rejestru DDRA noszą nazwy DDA0–DDA7 (brak literki R). Biblioteki AVR-GCC mają odpowiednie makra dla tych bitów. Moglibyśmy więc spróbować napisać:

```
DDRA = DDA0;
```

Niestety to nie zadziała, gdyż pod tymi makrami kryją się kolejne liczby od 0 do 7. A nam nie chodzi o zapis tych liczb, tylko ustawienie jedynki na określonej pozycji w rejestrze, wskazywanej przez te właśnie liczby. Np. dla nóżki 4 danego portu nie potrzebujemy liczby 4, tylko jedynki na pozycji 4. Binarnie będzie to 00010000, bo miejsca liczymy od prawej, a pierwsze miejsce ma numer 0. Dziesiętnie będzie to liczba 16. Dlatego stosuje się operator przesunięcia bitowego, który przesunie jedynkę o zadaną liczbę miejsc w lewo:

```
DDRA = 1 << DDA0;
```

W przypadku gdy chcemy ustawić kilka bitów, możemy wykonać sumę logiczną. Np. dla nóżek 0, 3 i 6:

```
DDRA =
1 << DDA0 | 1 << DDA3 | 1 << DDA6;
```

Jest to na pewno czytelniejsze od zapisu DDRA = 73. Wpisanie liczby 73 wydaje się prostsze, ale nie widać od razu, na których pozycjach w rejestrze znajdą się zera, a na których jedynki.

Nadal jednak chodzi nam o ustawianie konkretnych bitów w rejestrze, a sposób realizacji tej operacji nie jest tak bardzo istotny. Możemy w związku z tym wykorzystać makro _BV (od słów *bit vector*), które wykona przesunięcia bitowe, ale niejako je ukryje i pozwoli się skupić na samych bitach:

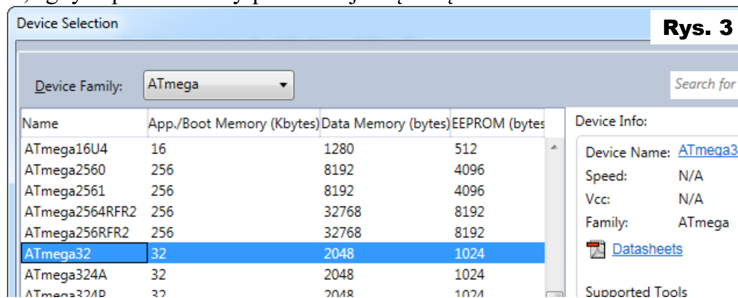
```
DDRA = _BV(DDA0) | _BV(DDA3) |
_BV(DDA6);
```

Przejdźmy teraz do właściwego sterowania diodą. W naszym przykładzie włączyliśmy ją pomiędzy nóżkę mikrokontrolera a masę, oczywiście poprzez rezystor ograniczający prąd. Włączana będzie więc logiczną jedynką.

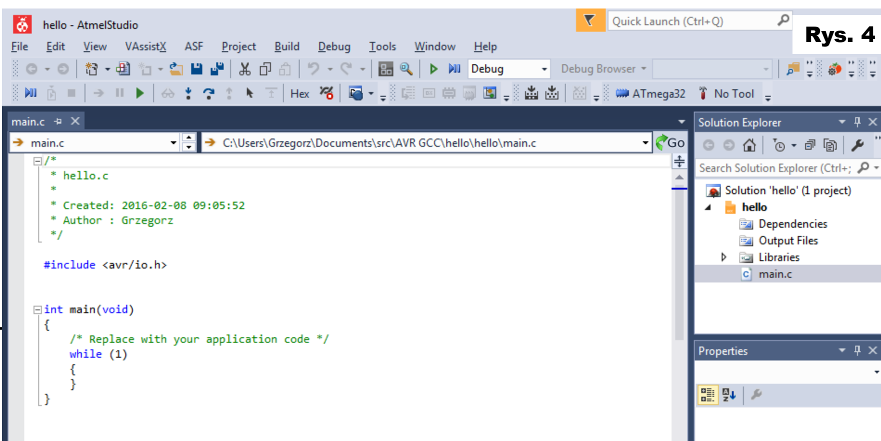
Za wartości wystawiane na portach odpowiadają rejestry PORTx. Ustawienie jedynki na nóżce zero wykonujemy więc następująco:

```
PORTA = _BV(PA0);
```

Oczywiście można by też napisać PORTA = 1; ale jak wspomniano, przy większych projektach takie pójście na skróty potem staje się uciążliwe.



Rys. 3



Rys. 4

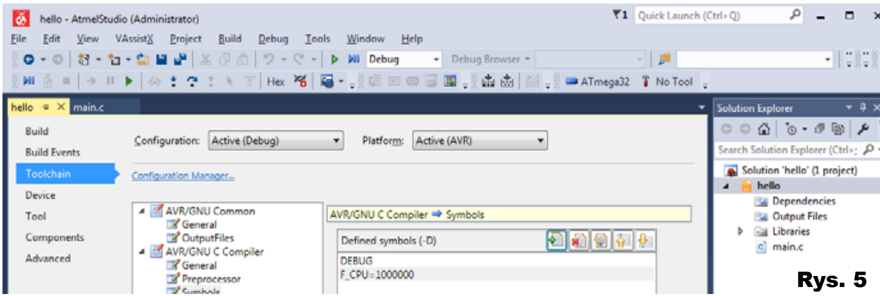
Uważni Czytelnicy zauważą, że wpisując wartość do rejestru portu, ustawiamy wartości wszystkich jego bitów. Powyższa linijka ustawi bit PA0, ale wyzeruje wszystkie inne bity. Innymi słowy ich wartości nie zostaną zachowane. W naszym przykładzie to jednak nie przeszkadza. Po pierwsze nie używamy pozostałych pinów portu, a ponadto po starcie mikrokontrolera i tak są wyzerowane. W bardziej rozbudowanych programach jednak bywa potrzebne modyfikowanie tylko niektórych bitów rejestru i omówimy to w dalszej części kursu.

Wprowadzanie opóźnienia

OK, LED się świeci. Aby otrzymać miganie, trzeba dodać odczekanie krótkiej chwili, zgaszenie LED-a, kolejne odczekanie i zacząć pętlę od początku. Gaszenie wykonujemy, resetując bit PA0 rejestru PORTA, wpisując do tegoż rejestru po prostu wartość 0. Rejestr PORTA zmieni więc swój stan z 00000001b na 00000000b. A jak wykonać opóźnienie?

Metod jest kilka. Jedna z nich to pętla opóźniająca, zwykle pętla ***for***, która wykona jakąś operację tak wiele razy, że wprowadzi to pożądane opóźnienie. Taka metoda nie jest jednak zbyt elegancka. Zależy ona od ustawień optymalizacyjnych kompilatora, nie daje łatwo przewidywalnych opóźnień i nie uwzględnia częstotliwości taktowania mikrokontrolera. Kolejny sposób to wykorzystanie przerwań. Będziemy je omawiać w dalszej części kursu. Wreszcie można użyć wbudowanych funkcji bibliotecznych, dostarczanych wraz z AVR-GCC.

Taką funkcją jest ***_delay_ms()*** z biblioteki *<util/delay.h>*. Jest to de facto pętla opóźniająca, jednak zapewniająca w miarę dokładne i przewidywalne opóźnienie. Jako parametr przyjmuje liczbę milisekund, przy czym liczba ta musi być znana w momencie kompilacji – nie może być zmienną. Wymaga też, aby w opcjach kompilatora była włączona optymalizacja przynajmniej na poziomie –O1. Ponadto musi być zdefiniowane makro *F_CPU* zawierające częstotliwość taktowania mikrokontrolera. W naszym przykładzie będzie ona wynosiła 1 MHz, bo skorzystamy z wewnętrznego generatora RC mikrokontrolera jako źródła sygnału. Dzięki temu opóźnienie będzie w miarę dokładne. Optymalizację ustawiamy we właściwościach projektu, które można otworzyć, korzystając z menu *Project* bądź też klikając prawym przyciskiem na naszym projekcie w okienku



Rys. 5

Solution Explorer. Nazwa projektu jest pogrubiona, nie klikamy na *Solution*.

Ustawienia optymalizacji znajdziemy w gałęzi *AVR/GNU C Compiler -> Optimization*. Domyślnie ustawiona jest na –O1 i nie trzeba jej przestawiać. Z kolei makro *F_CPU* możemy zdefiniować na dwa sposoby: w kodzie programu lub w gałęzi *AVR/GNU C Compiler -> Symbols* (**rysunek 5**). Ten drugi sposób jest globalny, dotyczy wszystkich plików danego projektu. Natomiast jeśli umieścimy makro w kodzie, będzie ono dostępne w zależności od jego położenia. Jeśli umieścimy je w pliku .c, to w obrębie tego pliku. Jeśli w pliku .h, to w plikach, które będą włączać ten plik. Z tego względu wpisanie makra w konfiguracji projektu wydaje się lepsze. Trzeba tylko pamiętać, że jest ono tam zdefiniowane i nie widać go, gdy się przegląda kod. Kolejna zaleta to to, że można stworzyć kilka konfiguracji (domyślne to Debug i Release) i łatwo się między nimi przełączać. Można w ten sposób np. łatwo kompilować program na mikrokontrolery z różnymi kwarcami bez zmieniania wartości częstotliwości w kodzie. Dopisujemy więc definicję F_CPU=1000000 obok istniejącej już definicji makra DEBUG.

Gotowy kod

Cały nasz program, z makrem F_CPU zdefiniowanym w konfiguracji projektu, wygląda następująco:

```
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
    DDRA = _BV(DDA0);
    while(1)
    {
        PORTA = _BV(PA0);
        delay_ms(250);
        PORTA = 0;
        delay_ms(250);
    }
}
```

Kompilujemy go, wciskając F7. Rezultaty kompilacji wyświetlane są w okienku Output. Jeśli wszystko wykonaliśmy poprawnie, pojawi się napis:

```
Build succeeded.
========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ==========
```

Ściągawka z C

`#include <avr/io.h>`

Dyrektywa #include jest poleceniem dla preprocesora. Informuje ona preprocesor, że zanim zacznie się kompilacja programu, ma być do niego wstawiona/dołączona zawartość danego pliku. Dołączany jest plik io.h z katalogu avr. Powodujemy tym, że do naszego kodu zostają dodane informacje o naszym mikrokontrolerze. Dzięki temu jesteśmy w stanie w kodzie naszego programu używać nazw takich jak PORTA czy DDA0 i będą one znane kompilatorowi, gdy się na nie natknie. Nawiasy trójkątne mówią, że chodzi o biblioteki dostarczone z kompilatorem, a nie pisane przez nas. Druga dyrektywa #include powoduje, że kompilatorowi znana staje się funkcja _delay_ms(). Bibliotekę <avr/io.h> będziemy dołączać w każdym naszym programie dla ATmega32.

`int main(void)`

Tą linijką rozpoczynamy główną funkcję naszego programu. Musi ona mieć nazwę **main** i być w każdym naszym programie, bo od niej mikrokontroler zaczyna swoją pracę. Tradycyjnie zwraca ona typ całkowitoliczbowy int, ale w przypadku mikrokontrolerów nie ma to znaczenia, bo mikrokontroler ma pracować cały czas i nie działają na nim inne programy. Funkcja **main** nie przyjmuje też parametrów, co zaznaczamy słowem kluczowym void.

`{}`

Nawiasy klamrowe oznaczają blok kodu, mówią, odkąd dokąd jest treść funkcji, pętli lub warunku. W naszym przykładzie używamy nawiasów klamrowych do oznaczenia, które linijki należą do funkcji main, a które do pętli while. Pętla while jest wewnątrz main, mamy więc do czynienia z tzw. zagnieżdżeniem.

;

Średnikiem oznaczamy koniec instrukcji

=

Operator przypisania wartości. Nie należy go mylić z matematycznym operatorem równości. Mówi on, że wartość znajdująca się z jego prawej strony ma być umieszczona w zmiennej lub rejestrze wskazanym po jego lewej stronie. Np. a = b + 5; oznacza odczytanie wartości zmiennej b, dodanie do niej liczby 5 i zapis wyniku w zmiennej a. Zapis a = a + 5; spowoduje zwiększenie zmiennej a o 5, bo oznacza dodanie liczby 5 do bieżącej wartości zmiennej a i zapis wyniku w tej samej zmiennej. Można go skrócić do postaci a += 5; Takie skróty można też stosować do wielu innych operatorów, np. -=, *=, <<=, >>=, &=, |=

| oraz &

Bitowe operatory odpowiednio sumy logicznej i iloczynu logicznego. Działają dokładnie jak bramki OR i AND, tylko operują na wszystkich, odpowiadających sobie parach bitów dwóch liczb dwójkowych.

<<

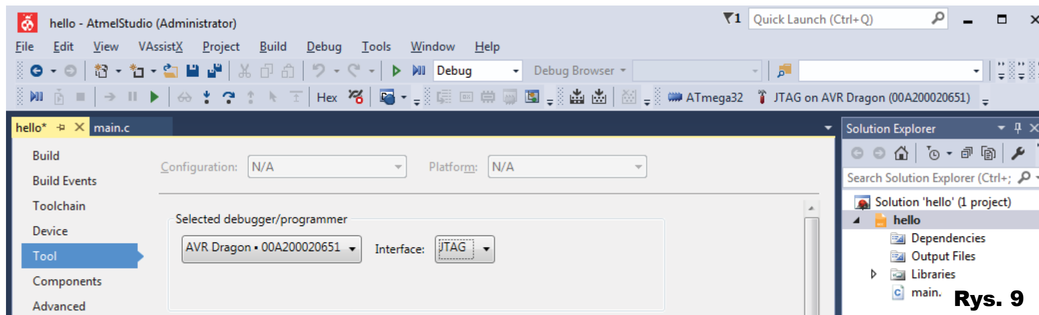
Operator przesunięcia bitowego przesuwa bity w liczbie dwójkowej o daną liczbę pozycji w lewo lub w prawo. A << B oznacza „przesuń bity w A w lewo o B pozycji", A >> B oznacza „przesuń bity w A w prawo o B pozycji". Z punktu widzenia matematycznego przesunięcie bitów o B pozycji powoduje pomnożenie wartości A przez 2 do potęgi B, a przesunięcie bitów w prawo spowoduje podzielenie A przez 2^B.

`_delay_ms(250);`

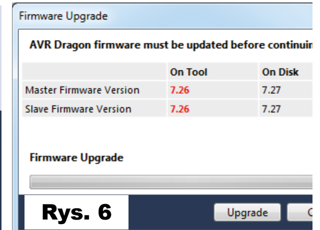
Wywołanie funkcji _delay_ms() z parametrem 250. Funkcje są fragmentami kodu, które wykonują określoną czynność, tutaj jest to po prostu opóźnienie. Funkcje mogą przyjmować parametry i/lub zwracać wartość. _delay_ms() jako parametr przyjmuje czas opóźnienia w milisekundach, nie zwraca wartości. Zazwyczaj skompilowana funkcja znajduje się w określonym miejscu pamięci i w momencie jej wywołania procesor skacze do tego miejsca, a po zakończeniu jej wykonywania powraca do kodu wywołującego. Funkcja _delay_ms() jest akurat szczególnym przypadkiem, gdyż jest to funkcja inline, która przez kompilator jest „wklejana" w miejsce jej wywołania, nie ma skoku w inne miejsce pamięci programu.

while(1)

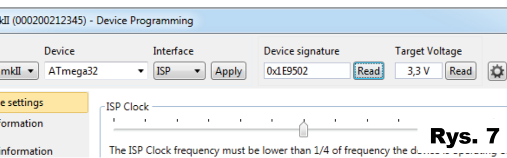
Linijka ta oznacza, że dany fragment kodu ma być wielokrotnie powtarzany, czyli wykonywany w pętli. Fragment ten oznaczony jest nawiasami {}, czasem opuszczanymi, gdy fragment to tylko jedna linijka. Powtórzenia mają miejsce tak długo, jak długo podany w nawiasach okrągłych warunek jest różny od zera. W naszym przykładzie warunkiem jest liczba 1, która od zera będzie oczywiście różna zawsze i pętla nigdy się nie skończy.



Rys. 9



Rys. 6



Rys. 7

Programowanie pamięci mikrokontrolera

Mając skompilowany program, musimy zapisać go w pamięci Flash mikrokontrolera za pomocą posiadanego przez nas programatora. Podłączamy go do komputera oraz do naszej płytki testowej. W przypadku programatora AVR Dragon łączymy przewodem taśmowym złącze JTAG w programatorze ze złączem JTAG w płytce testowej. Należy uważać na orientację przewodu: programator ma pin numer 1 od strony krawędzi płytki, płytka testowa ma pin 1 od strony złącza MISC. Jeśli mamy AVT-5388, łączymy przewodem taśmowym złącze KANDA programatora ze złączem ISP płytki testowej. Rozmieszczenie pinów złącza KANDA jest opisane na płytce programatora – pin 1 jest od strony LED-ów. Na płytce testowej pin 1 złącza ISP znajduje się od strony złącza JTAG. W programatorze AVT-5388 zworki należy ustawić w pozycjach 5V oraz EX. Gdy wszystko jest podłączone, a płytka testowa ma doprowadzone napięcie zasilające, możemy przejść do programowania mikrokontrolera z poziomu Atmel Studio.

Ponieważ jest to nasze pierwsze podejście do zapisania programu w pamięci mikrokontrolera, zrobimy to niejako na piechotę. Zaczynamy od wybrania z menu *Tools* pozycji *Device Programming*. W tym momencie może się zdarzyć, że Atmel Studio wykryje nieaktualne oprogramowanie (firmware) programatora (**rysunek 6**). Mamy wtedy możliwość jego aktualizacji. Wyjątkiem jest AVT-5388, który należy aktualizować zgodnie z załączoną do niego instrukcją, opublikowaną także w lutowym numerze EdW.

Gdy pojawi się okienko *Device Programming*, trzeba z listy Tool w lewym górnym rogu wybrać posiadany programator (**rysunek 7**).

AVT-5388 będzie widoczny jako AVRISP mkII. Po wyborze programatora powinien ustawić się automatycznie odpowiedni dla niego interfejs na liście *Interface*. Dla AVR Dragona będzie to JTAG, a dla AVT-5388 ISP. Teraz klikamy Apply, co spowoduje zatwierdzenie ustawień. Dla przetestowania, że wszystko jest poprawnie podłączone i skonfigurowane, kliknijmy przycisk Read, aby odczytać sygnaturę mikrokontrolera. Dla ATmega32 będzie to 0x1E9502. Jeśli odczyt sygnatury nie powiedzie się, trzeba skontrolować poprawność połączeń i zasilania płytki testowej.

Gdy sygnatura jest poprawnie odczytywana, możemy przejść dalej. Ale zanim zaprogramujemy pamięć Flash naszego mikrokontrolera, sprawdźmy ustawienia fuse bitów. Są to dwa specjalne bajty nieulotnej pamięci, w której przechowywane są ustawienia procesora, takie jak źródło sygnału zegarowego, włączenie interfejsu JTAG czy napięcie, poniżej którego następuje reset procesora (funkcja Brown-out Detector). Nie będziemy ich teraz szczegółowo omawiać. Ważne, żeby były ustawione tak, jak na **rysunku 8,** czyli bajt HIGH ma mieć wartość 0x99, a bajt LOW wartość 0xE1. Są to domyślne wartości dla ATmega32.

Po skontrolowaniu fuse bitów warto zajrzeć do sekcji *Memories*. Daje ona możliwość wykasowania pamięci mikrokontrolera (przycisk Erase now) oraz zaprogramowania pamięci Flash i EEPROM za pomocą wskazanych plików. Na razie pozostawiamy wszystkie ustawienia tak jak są domyślnie. Ścieżkę do pliku z naszym skompilowanym programem mamy wybraną automatycznie. Zapis programu wykonujemy, klikając *Program*. Gdy programowanie się zakończy, program zacznie działać na mikrokontrolerze. Jeśli nie popełniliśmy błędu, dioda na płytce testowej powinna zacząć migać z częstotliwością 2Hz.

W ten sposób napisaliśmy i uruchomiliśmy nasz pierwszy program.

Gdy programowanie naszego mikrokontrolera przebiega bez zakłóceń, a programator mamy wybrany we właściwościach projektu (**rysunek 9**), programowanie mikrokontrolera możemy wykonać na skróty. Po skompilowaniu programu nie trzeba otwierać okienka *Device Programming*, tylko wystarczy skorzystać z funkcji *Start Without Debugging*, która znajduje się w menu *Debug*. Można ją uruchomić także skrótem klawiszowym Ctrl+Alt+F5 lub klikając w przycisk z ikoną zielonego trójkąta (ale tego bez wypełnienia) na pasku narzędzi. Spowoduje to automatyczne załadowanie naszego programu do mikrokontrolera oraz jego uruchomienie.

Rys. 8

Zadania

W tym odcinku nauczyliśmy, się jak skonfigurować wybrany pin jako wyjście, jak ustawić jego stan i jak wykonać opóźnienie. Wiedzę tę wykorzystaliśmy do migania jedną diodą. Ponieważ w nauce najważniejsze są własne eksperymenty, zachęcam do modyfikacji naszego pierwszego programu:

- miganie kilkoma LED-ami naraz
- miganie dwoma LED-ami na przemian
- zmiana szybkości migania
- zmiana stosunku czasu zgaszenia do czasu świecenia

Rozwiązania zostaną przedstawione w kolejnym numerze EdW.

Grzegorz Niemirowski
grzegorz@grzegorz.net

Zawsze znajdziesz, przejrzysz i kupisz aktualny numer „Elektroniki dla Wszystkich”
(zarówno w wersji papierowej, jak i elektronicznej)
na ***www.UlubionyKiosk.pl***