Kurs AVR – lekcja 4

Rozwiązania zadań z ostatniego odcinka

Pierwszym zadaniem domowym z poprzedniego odcinka było generowanie dźwięku o wysokości zależnej od wciśniętego klawisza na klawiaturze matrycowej. Sygnał prostokątny miał być generowany z użyciem timera, bez funkcji opóźniającej. Na **listingu 1** znajduje się przykładowa realizacja tego zadania. Na początku kodu ustawiamy pin OC0 (PB3) do pracy jako wyjście. Inicjalizowane są piny klawiatury. W timerze Timer0 ustawiamy preskaler na 64, czyli licznik timera będzie zwiększany co 64 takty zegara systemowego. Włączany jest tryb CTC oraz ciągłe przestawianie pinu OC0 na stan przeciwny. W głównej pętli sprawdzamy, czy naciśnięty jest przycisk. Jeśli tak, to rejestr OCR0 odpowiedzialny za to, przy jakiej wartości licznik liczy od 0, ustawiany jest na dziesięciokrotność numeru wciśniętego klawisza. Jeśli przycisk nie jest wciśnięty, wyłączane jest taktowanie licznika, co przerywa generowanie dźwięku.

Drugim zadaniem było płynne zaświecanie i gaszenie diody. Inicjalizacja wygląda podobnie, ale jak pokazauje **listing 2**, zamiast trybu CTC używamy Fast PWM. Pin OC0 jest ustawiany w stan 0 na początku cyklu, a następnie od momentu osiągnięcia wartości z rejestru OCR0 do końca cyklu jest w stanie 1. Zmiana jasności osiągana jest przez zliczanie przepełnień licznika, sygnalizowanych ustawieniem flagi TOV0. Po 11 przepełnieniach (liczymy od 0) wykonywana jest zmiana wartości rejestru OCR0, który wyznacza wypełnienie generowanego sygnału PWM. W zależności od aktualnego kierunku zmian jest to zwiększenie o 1 lub zmniejszenie o 1. Po osiągnięciu przez rejestr wartości 0, czy to na skutek zmniejszania jego wartości, czy też na skutek przepełnienia podczas zwiększania, kierunek jest przełączany na przeciwny.

Timer/Counter1

Po omówieniu Timer0 przyszła pora na Timer1. Ogólnie działa on podobnie, ale ma dwie wyróżniające go cechy. Po pierwsze, jest 16-bitowy, czyli jego rejestr licznikowy może liczyć nie do 255, ale aż do 65535. 16-bitowe są też rejestry funkcji Output Compare i Input Compare timera. Przy takim samym sygnale zegarowym możemy więc odmierzać dłuższe okresy. Oznacza to też większą rozdzielczość licznika – możemy dokładniej kontrolować np. generowaną częstotliwość. Druga cecha Timer1 to wspomniana funkcja Input Compare. Pozwala ona zapamiętać stan licznika, jaki był w momencie wystąpienia zewnętrznego zdarzenia. Jest bardzo przydatna przy pomiarach częstotliwości.

Pracę timera Timer1 kontrolują następujące rejestry:

- TCCR1A – rejestr konfiguracyjny A
- TCCR1B – rejestr konfiguracyjny B
- TCNT1 – rejestr licznikowy, dostępny 8-bitowo jako TCNT1H i TCNT1L
- OCR1A – pierwszy rejestr funkcji Output Compare, dostępny 8-bitowo jako OCR1AH i OCR1AL
- OCR1B – drugi rejestr funkcji Output Compare, dostępny 8-bitowo jako OCR1BH i OCR1BL
- ICR1 – rejestr funkcji Input Compare, dostępny 8-bitowo jako ICR1H i ICR1L
- TIMSK – znany nam już rejestr konfiguracji przerwań timerów, współdzielony przez wszystkie timery
- TIFR – rejestr flag timerów, również omówiony przy okazji Timer0 i współdzielony przez wszystkie timery

Choć Timer1 ma 16-bitowe rejestry, to nasz mikrokontroler pozostaje 8-bitowy. Oznacza to, że nie może on wykonać na tych rejestrach operacji w jednej instrukcji. Musi to robić na raty, po 8 bitów. Dlatego dla rdzenia procesora rejestr licznikowy TCNT1 to tak naprawdę dwa 8-bitowe rejestry TCNT1H (starsze 8 bitów) i TCNT1L (młodsze 8 bitów). Na szczęście nasz kompilator pozwala w kodzie języka C odwoływać się do rejestru jako całości. Dopiero na etapie kompilacji wstawi do kodu maszynowego instrukcje odwołujące się do obu połówek rejestru. Dzieje się to niejako pod spodem i jest niewidoczne dla programisty. Jednak o tym, jak rzeczywiście wykonywane są odwołania do rejestrów, trzeba pamiętać, bo w niektórych przypadkach okazuje się to bardzo ważne. Przekonamy się o tym w odcinku poświęconym przerwaniom.

Dzięki temu, że Timer1 ma dwa rejestry dla funkcji Output Compare (OCR1A i OCR1B), może reagować na osiągnięcie przez licznik dwóch różnych wartości. Ma jakby dwa kanały, A i B. Gdy licznik, czyli rejestr TCNT1, osiągnie wartość z rejestru OCR1A, ustawiana jest flaga OCF1A, osiągnięcie wartości z rejestru OCR1B sygnalizowane jest flagą OCF1B. Możemy też generować dwa przebiegi PWM na pinach OC1A i OC1B (PD5 i PD4). Oczywiście będą miały one tę samą częstotliwość, bo licznik jest jeden, ale będą mogły mieć różne wypełnienie. Ponadto to, czy sygnał na pinie jest zanegowany, czy nie, też można ustawić niezależnie. Jeden pin może więc mieć wypełnienie normalne, a drugi zanegowane. Oczywiście można to zrobić programowo, wpisując dopełniające się wartości do reje-

```
#include <avr/io.h>
#include <util/delay.h>
uint8_t readKeyboard();
int main(void) {
 DDRB = _BV(DDB3);
 DDRA = _BV(DDA0) | _BV(DDA1) | _BV(DDA2) | _BV(DDA3);
 TCCR0 = _BV(CS01) | _BV(CS00) | _BV(WGM01) | _BV(COM00);
 while(1) {
  uint8_t key = readKeyboard();
  if (key) {
   TCCR0 |= _BV(CS01) | _BV(CS00);
   OCR0 = 10 * key;
  } else TCCR0 &= ~(_BV(CS01) | _BV(CS00));
 }
}
uint8_t readKeyboard() {
 for (uint8_t row = 0; row < 4; row++){
  PORTA = ~_BV(row);
  _delay_us(1);
  for (uint8_t column = 0; column < 4; column++){
   if (~PINA & _BV(column + 4)) {
    return row * 4 + column + 1;
   }
  }
 }
 return 0;
}
```

Listing 1

Listing 2

```
#include <avr/io.h>
int main(void){
 DDRB = _BV(DDB3);
 DDRA = _BV(DDA0) | _BV(DDA1) | _BV(DDA2) | _BV(DDA3);
 TCCR0 = _BV(WGM01) | _BV(WGM00) | _BV(COM01) | _BV(CS01);
 uint8_t count = 0;
 uint8_t direction = 1;
 while(1) {
  TIFR |= _BV(TOV0);
  while(!(TIFR & _BV(TOV0)));
  count++;
  if (count == 10) {
   if (direction) OCR0++; else OCR0--;
   if (OCR0 == 0) direction = !direction;
   count = 0;
  }
 }
}
```

COM1A1/COM1B1	COM1A0/COM1B0	Zachowanie pinu
0	0	Brak sterowania pinami OC1A i OC1B przez timer
0	1	Ustawienie pinu OC1A w stan przeciwny, OC1B nie jest sterowany
1	0	Tryb nieodwracający PWM
1	1	Tryb odwracający PWM

Tab. 2

COM1A1/COM1B1	COM1A0/COM1B0	Zachowanie pinu
0	0	Brak sterowania pinami OC1A i OC1B przez timer
0	1	Przestawienie pinu na stan przeciwny
1	0	Ustawienie w stan niski
1	1	Ustawienie w stan wysoki

Tab. 1

strów OCR1A i OCR1B, ale w ten sposób możemy osiągnąć to samo sprzętowo, bez obliczeń, wpisując tę samą wartość do obu rejestrów. W **tabelach 1 i 2** przedstawiono konfigurację bitów rejestru TCCR1A dla sterowania pinami w trybie normalnym oraz PWM.

Sygnał zegarowy dla Timer1 konfigurujemy za pomocą bitów rejestru TCCR1B, zgodnie z **tabelą 3**.

Jak widać, jest ona prawie identyczna z tabelą dla Timer0. Różne są oczywiście nazwy bitów, różny jest też pin zewnętrznego taktowania. Pin T1 to pin PB1, czyli drugi pin portu B. Jest to nóżka 2 dla obudowy przewlekanej PDIP.

Tryby pracy Timer1. Tryby pracy timera Timer1 są podobne do Timer0, ale jest ich dużo więcej. Dostępne tryby, konfigurowane rejestrami WGM13...10, przedstawione są w **tabeli 4**. Bity WGM11 i WGM10 ustawiamy w rejestrze TCCR1A, a bity WGM13 i WGM12 w rejestrze TCCR1B.

CS12	CS11	CS10	Taktowanie timera
0	0	0	Brak, timer wyłączony
0	0	1	clk/1
0	1	0	clk/8
0	1	1	clk/64
1	0	0	clk/256
1	0	1	clk/1024
1	1	0	Pin T1, zbocze opadające
1	1	1	Pin T1, zbocze narastające

Tab. 3

Jak widać z tabeli, licznik należący do Timer1 może być „skracany" do 8, 9 lub 10 bitów w trybach PWM Phase Correct i Fast PWM. Może też liczyć do wartości przechowywanych w rejestrach ICR1 lub OCR1A. Tak samo od tych dwóch rejestrów może być uzależniona także funkcja CTC (Clear Timer on Compare Match). Jeśli do wyznaczania wartości maksymalnej użyjemy rejestru OCR1A, to wartością odniesienia dla trybu CTC lub PWM będzie mógł być tylko rejestr OCR1B.

Oprócz trybów PWM z korekcją fazy dostępne są tryby z dodatkową korekcją częstotliwości. Polegają one na tym, że zmiany w rejestrach OCR1x brane są pod uwagę w chwili, gdy licznik ma wartość 0, a nie, gdy ma wartość maksymalną. Pozwala to uniknąć generowania impulsów i nieprawidłowej częstotliwości przy zmienianiu rozdzielczości licznika (wartości TOP).

Na **listingu 3** pokazana jest przykładowa konfiguracja Timer1 dla generowania dwóch sygnałów PWM: normalnego na nóżce OC1A (PD5) i odwróconego na nóżce OC1B (PD4). Maksymalną wartość licznika (TOP) ustawiamy rejestrem ICR1. Do obu rejestrów OCR1A i OCR1B wpisywana jest ta sama wartość. Stosunek tej wartości do wartości rejestru ICR1 da wypełnienie przebiegu PWM na OC1A. Na OC1B będzie wypełnienie przeciwne, czyli o wartości takiej jak na OC1A odjętej od 100%. Łatwo to sprawdzić, podłączając do obu wyjść LED-y i obserwując ich jasność przy zmianie wpisywanej wartości.

Funkcja Input Capture. Jak pamiętamy, funkcja Output Capture polegała na wygenerowaniu zdarzenia (wyzerowanie licznika, ustawienie flagi) przy osiągnięciu przez licznik wartości wpisanej w odpowiednim rejestrze. Input Capture działa niejako odwrotnie. Do odpowiedniego rejestru wpisuje wartość licznika w momencie wystąpienia zdarzenia. W przypadku Timer1 rejestrem tym jest ICR1. Natomiast zdarzenia mogą być dwa: zbocze na pinie ICP1 lub też zmiana stanu komparatora analogowego, który jest wbudowany w nasz mikrokontroler. Skupimy się teraz na tym pierwszym. Pin ICP1 to jednocześnie PD6, czyli siódmy pin portu D, nóżka 20 dla zwykłej przewlekanej obudowy PDIP. Wykrywane może być zbocze narastające na pinie lub opadające. Wyboru dokonujemy za pomocą bitu ICES1 w rejestrze TCCR1B. Jeśli ustawimy ten bit na 0, wykrywane będą zbocza opadające, gdy ustawimy na 1, wówczas narastające. Wykrywanie zboczy może mieć włączoną funkcję redukcji zakłóceń. Włączamy ją, ustawiając bit ICNC1 w rejestrze TCCR1B. Jeśli jest włączona, zbocze zostanie wykryte dopiero wtedy, gdy po zmianie stanu pinu jego nowa wartość utrzyma się przez cztery takty zegara mikrokontrolera. Można w ten sposób wyeliminować zakłócenia mające postać bardzo krótkich szpilek. Z drugiej strony wykrycie zbocza nastąpi z opóźnieniem wynoszącym właśnie te cztery takty, co czasem może mieć znaczenie.

Wykorzystajmy Input Capture do prostego pomiaru częstotliwości. Zapamiętywany przez nią stan licznika w momencie wystąpienia zbocza jest informacją o tym, kiedy ono wystąpiło. Gdy pojawi się kolejne zbocze, możemy porównać dopiero co przechwycony stan licznika z poprzednim. Znając okres taktowania licznika, będziemy mogli obliczyć okres taktowanego sygnału, a z niego częstotliwość. Załóżmy, że taktujemy mikrokontroler częstotliwością 1MHz i nie używamy preskalera dla timera/licznika. Gdy zostało

Tab. 4

Tryb pracy	WGM13	WGM12	WGM11	WGM10	Maksymalna wartość licznika	Aktualizacja rejestru OCR1x przy wartości	Ustawienie flagi TOV1 przy wartości
Normalny	0	0	0	0	0xFFFF	Natychmiastowo	MAX
PWM, Phase Correct, 8-bitowy	0	0	0	1	0x00FF	TOP	BOTTOM
PWM, Phase Correct, 9-bitowy	0	0	1	0	0x01FF	TOP	BOTTOM
PWM, Phase Correct, 10-bitowy	0	0	1	1	0x03FF	TOP	BOTTOM
CTC	0	1	0	0	OCR1A	Natychmiastowo	MAX
Fast PMW, 8-bitowy	0	1	0	1	0x00FF	BOTTOM	TOP
Fast PMW, 9-bitowy	0	1	1	0	0x01FF	BOTTOM	TOP
Fast PMW, 10-bitowy	0	1	1	1	0x03FF	BOTTOM	TOP
PWM, Phase and Frequency Correct	1	0	0	0	ICR1	BOTTOM	BOTTOM
PWM, Phase and Frequency Correct	1	0	0	1	OCR1A	BOTTOM	BOTTOM
PWM, Phase Correct	1	0	1	0	ICR1	TOP	BOTTOM
PWM, Phase Correct	1	0	1	1	OCR1A	TOP	BOTTOM
CTC	1	1	0	0	ICR1	Natychmiastowo	MAX
Zarezerwowany	1	1	0	1			
Fast PWM	1	1	1	0	ICR1	BOTTOM	TOP
Fast PWM	1	1	1	1	OCR1A	BOTTOM	TOP

zarejestrowane pierwsze zbocze, z rejestru ICR1 odczytaliśmy liczbę 10, a przy drugim zboczu liczbę 110. Różnica wynosi 100 taktów, co dla 1MHz daje 100 mikrosekund, a więc częstotliwość 10kHz. A skąd będziemy wiedzieć, że w ICR1 znajduje się przechwycona wartość licznika? Funkcja Input Capture ustawia flagę ICF1 w rejestrze TIFR. Gdy wykryjemy jej ustawienie, możemy odczytać rejestr ICR1. Tematykę flag poruszaliśmy już przy Timer0 przy okazji flag przepełnienia licznika oraz flagi funkcji Output Capture. Wstępna wersja przykładowego kodu źródłowego wygląda następująco (**listing 4**):

```
#include <avr/io.h>
int main(void) {
 DDRA = 0b11111111;
 TCCR1B = _BV(CS10);
 uint16_t prevICR1 = 0;
 while(1) {
  while(!(TIFR & _BV(ICF1)));
  TIFR |= _BV(ICF1);
  if (ICR1 > prevICR1) {
   uint16_t period = ICR1 - prevICR1;
   uint16_t freq = 1000 / period;
   PORTA = _BV(freq);
  }
  prevICR1 = ICR1;
 }
}
```

W programie tym do wyświetlania wyniku pomiaru wykorzystywana jest linijka diodowa sterowana przez port A. Wszystkie piny tego portu pracują jako wyjścia, wszystkie bity rejestru DDRA ustawiamy więc na 1. W rejestrze TCCR1B ustawiony jest bit CS10, aby źródłem taktowania licznika był zegar systemowy, bez żadnego podziału częstotliwości. W naszym przykładzie nie ma znaczenia, czy wyzwalanie funkcji Input Capture następuje zboczem narastającym, czy opadającym, bit ICES1 pozostawiamy nieustawiony. W rejestrze TCCR1A nie musimy nic ustawiać.

W głównej pętli programu czekamy na pojawienie się flagi ICF1. Gdy się pojawi, kasujemy ją. Następnie porównujemy stan rejestru ICR1 z poprzednio zapamiętaną wartością. Jeśli nowa wartość jest większa, czyli w międzyczasie nie nastąpiło przepełnienie licznika, odejmujemy nową wartość od starej i otrzymujemy okres sygnału, wyrażony w mikrosekundach. Aby otrzymać częstotliwość wyrażoną w kilohercach, dzielimy liczbę 1000 przez otrzymany okres. W celu wyświetlenia wyniku w postaci punktu, a nie liczby dwójkowej, stosowane jest znane nam już makro _BV().

Aby przetestować nasz program, podłączamy źródło sygnału prostokątnego do pinu ICP1 (PD6). Jeśli nie dysponujemy

```
#include <avr/io.h>
int main(void) {
 DDRD = _BV(DDD4) | _BV(DDD5);
 TCCR1A = _BV(WGM11) | _BV(COM1A1) | _BV(COM1B0) |_BV(COM1B1);
 TCCR1B = _BV(CS10) | _BV(WGM13);
 uint16_t val = 0x0f00;
 OCR1A = val;
 OCR1B = val;
 ICR1 = 0x0fff;
 while(1) { }
}
```

Listing 3

generatorem sygnałowym, możemy wykorzystać generator znajdujący się na naszej płytce testowej. W tym celu wystarczy połączyć pin GO złącza MISC z pinem PD6 złącza PORTD. Dodatkowo możemy połączyć piny Pot i GI, aby przestrajać generator za pomocą potencjometru znajdującego się również na płytce testowej.

Jak możemy odczytać częstotliwość? W zmiennej freq przechowywane są pełne kiloherce. Częstotliwości mniejsze niż 1 kHz dadzą więc liczbę 0, częstotliwość 1 kHz i wyżej, ale poniżej 2kHz da liczbę 1, i tak dalej. Następnie poprzez makro _BV() wyznaczana jest dioda, która ma zostać zaświecona. Jak pamiętamy, makro to po prostu wykonuje przesunięcie jedynki w lewo o daną liczbę pozycji. Dla częstotliwości poniżej 1kHz przesunięcie będzie zerowe, świecić więc będzie pierwsza dioda. Po przekroczeniu 1kHz zaświeci się druga dioda i tak dalej. Świecenie ósmej diody będzie oznaczało częstotliwość 7kHz lub wyższą, ale poniżej 8kHz. Przy 8kHz i wyżej nie będzie świecić już żadna dioda.

Nasz miernik możemy przeskalować, wystarczy zmienić liczbę 1000 na inną. Czym liczba ta będzie większa, tym zakres będzie niższy. Np. przy 10 000 ostatnia dioda włączy się już przy częstotliwości 700Hz. Z kolei zmniejszając ją, będziemy mogli mierzyć wyższe częstotliwości. Oczywiście nie możemy mierzyć dowolnie dużych lub dowolnie małych częstotliwości. Między „złapaniem" zboczy mikrokontroler musi zdążyć sprawdzić stan flagi, wyzerować ją, wykonać obliczenie częstotliwości i wyświetlić wynik. Ile czasu będzie na to potrzebował, zależy od stopnia optymalizacji w ustawieniach kompilatora oraz szybkości taktowania mikrokontrolera. Innymi słowy, „od góry" ogranicza nas szybkość przetwarzania danych przez mikrokontroler. Z kolei „od dołu" ograniczeniem jest rozmiar licznika. Nasz kod zakłada, że zdąży złapać dwa zbocza, zanim licznik się przepełni. W najlepszym wypadku ma na to 65 535 cykli, co przy taktowaniu 1MHz daje 65,5 milisekundy i częstotliwość graniczną 15,25Hz. I to przy założeniu, że pierwsze zbocze trafi się nam zaraz po zresetowaniu licznika. W praktyce więc częstotliwość minimalna będzie dwa razy wyższa, ok. 31Hz. Gdyby zaszła potrzeba pomiaru niższych częstotliwości, możemy skorzystać z preskalera naszego licznika i zmniejszyć jego taktowanie nawet 1024 razy, co da naprawdę niską częstotliwość graniczną rzędu 0,03Hz.

Timer2

Trzecim timerem dostępnym w mikrokontrolerze ATmega32 jest 8-bitowy Timer2. Jest on bardzo podobny do Timer0. Różnica polega na tym, że może być taktowany asynchronicznie, z wykorzystaniem typowego zegarkowego rezonatora kwarcowego 32768 Hz. Rezonator ten podłącza się do pinów TOSC1 i TOSC2 (PC6 i PC7) bezpośrednio, bez kondensatorów. Jeśli zastosujemy preskaler z podziałem przez 128, wówczas 8-bitowy licznik będzie się przepełniał dokładnie raz na sekundę (128 * 256 = 32768). Pracę asynchroniczną Timer2 uaktywnia się, ustawiając bit AS2 w rejestrze ASSR. Należy pamiętać, że po zapisie do rejestrów TCNT2, OCR2 i TCCR2 w trybie asynchronicznym konieczne jest poczekanie, aż wyzerują się flagi w rejestrze ASSR, odpowiednio TCN2UB, OCR2UB i TCR2UB.

Timer2 może więc służyć jako wygodne źródło sygnału zegarowego do odmierzania czasu. A czy nie można wykorzystać Timer0 lub Timer1? W przypadku Timer0 byłoby to niewygodne, gdyż ma on tylko 8 bitów. Przy taktowaniu kwarcem 16 MHz i maksymalnym preskalerze 1024 jego licznik przepełniałby się z częstotliwością ok. 61,035 Hz. Aby zejść do 1 sekundy, trzeba by np. wykorzystać preskaler 256, funkcjami CTC i Output Compare obniżyć maksymalną wartość licznika do 250 oraz zastosować dodatkową zmienną liczącą wystąpienie 250 przepełnień licznika (256 * 250 * 250 = 16 000 000). Z 16-bitowym Timer1 sytuacja jest lepsza. Jeśli mamy kwarc 16 MHz, jak w naszej płytce, to stosując preskaler 256 i skrócenie licznika do 62 500 cykli (256 * 62 500 = 16 000 000) funkcjami CTC i Output Compare osiągniemy cel. Jednak w trochę większych projektach dobrze jest mieć 16-bitowy Timer1 wolny do użycia w innych celach niż odliczanie sekund. Stąd przydatność trybu asynchronicznego w Timer2.

Tryby pracy Timer2. Tryby pracy Timer2 są takie same jak dla Timer0. Różnica polega na tym, że konfigurujemy bity WGM21 i WGM20 w rejestrze konfiguracyjnym TCCR2 - **tabela 5**.

Brak jest taktowania timera zewnętrznym sygnałem, za to jest więcej stopni podziału zegara w preskalerze. Bity CS22, CS21 i CS20 ustawiamy w rejestrze TCCR2 - **tabela 6**.

Tryb pracy	WGM21	WGM20	Maksymalna wartość licznika	Aktualizacja rejestru OCR0 przy wartości	Ustawienie flagi TOV2 przy wartości
Normalny	0	0	0xFF	Natychmiastowo	MAX
PWM, Phase Correct	0	1	0xFF	TOP	BOTTOM
CTC	1	0	OCR0	Natychmiastowo	MAX
Fast PWM	1	1	0xFF	BOTTOM	MAX

Tabela 5

Nazwa bitu	Numer	Opis	Wartość domyślna
OCDEN	7	Włącza debugowanie (On-Chip Debugging)	1 (niezaprogramowany, OCD wyłączone)
JTAGEN	6	Włącza JTAG	0 (zaprogramowany, JTAG włączony)
SPIEN	5	Włącza programowanie przez SPI	0 (zaprogramowany, programowanie przez SPI włączone)
CKOPT	4	Opcje rezonatora	1 (niezaprogramowany)
EESAVE	3	Zachowywanie pamięci EEPROM przy kasowaniu pamięci Flash	1 (niezaprogramowany, EEPROM niezachowywany)
BOOTSZ1	2	Rozmiar bloku bootloadera	0 (zaprogramowany)
BOOTSZ0	1	Rozmiar bloku bootloadera	0 (zaprogramowany)
BOOTRST	0	Włącza skok do bootloadera po resecie	1 (niezaprogramowany, bez uruchamiania bootloadera)

Tabela 7

Rejestrem licznikowym jest TCNT2, rejestrem funkcji Output Compare jest OCR2. Pinem, który może być sterowany przez Timer2, jest OC2 (PD7). Konfiguruje się go analogicznie jak przy Timer0, tylko zamiast bitów COM01 i COM00 używa się COM21 i COM20, a zamiast rejestru TCCR0 używa się TCCR2.

Zmiana prędkości pracy mikrokontrolera

Dotychczas nie zmienialiśmy domyślnego sposobu taktowania naszego mikrokontrolera – źródłem jego sygnału zegarowego był wewnętrzny generator RC o częstotliwości 1MHz. W wielu przypadkach będzie to wystarczające, jednak czasem potrzebna będzie zmiana taktowania. Kiedy?

Pierwsza odpowiedź: żeby mikrokontroler działał szybciej. Każdy z nas przecież chce mieć szybki komputer. W przypadku mikrokontrolerów nie jest to jednak takie oczywiste. Trzeba pamiętać, że wraz ze wzrostem częstotliwości pracy rośnie pobór prądu przez mikrokontroler. Gdy nasz układ jest zasilany z zasilacza sieciowego, nie ma to większego znaczenia. Jednak przy zasilaniu bateryjnym zależy nam na jak najdłuższym działaniu na jednym ładowaniu baterii. Szybkość taktowania musi być więc kompromisem pomiędzy potrzebną mocą obliczeniową a oszczędnością energii.

Chcąc mieć szybszy komputer, możemy wymienić w nim części na szybsze, ale też usunąć niepotrzebne lub źle napisane programy, które są zbędnym obciążeniem. Podobnie w przypadku kodu na mikrokontroler może się okazać, że coś jest napisane nieoptymalnie lub jakiś fragment pozostał po przeróbkach i nie pełni już żadnej funkcji. Oczywiście nie ma sensu spędzać przesadnie dużo czasu na optymalizacji kodu, gdy można po prostu przyspieszyć mikrokontroler. Warto jednak zwracać uwagę na „lekkość" kodu. Ma to szczególne znaczenie przy przerwaniach, których obsługa powinna zajmować jak najmniej czasu.

Nie zawsze rozważamy zmianę częstotliwości taktowania mikrokontrolera pod kątem szybkości wykonywania się kodu i ogólnego przyspieszenia czy też spowolnienia. Czasem potrzebujemy konkretnej, dokładnie określonej częstotliwości zegara ze względu na działające w mikrokontrolerze peryferie. Przykładem mogą być omówione właśnie timery. Ze względu na ich skończoną rozdzielczość oraz ograniczoną możliwość wyboru stopni podziału częstotliwości w preskalerach, nie zawsze da się osiągnąć dokładnie taką częstotliwość generowanych przez timery zdarzeń, jak byśmy chcieli.

Podobna sytuacja jest w przypadku portu szeregowego, który będziemy niedługo omawiać. Transmisja przez port szeregowy teoretycznie może odbywać się z dowolną szybkością, ale w praktyce wykorzystywane są określone, standardowe prędkości, np. 115 200 bitów na sekundę. Aby ją osiągnąć, mikrokontroler musi podzielić częstotliwość swojego głównego zegara. Ponieważ podział jest wykonywany przez liczby całkowite, nie zawsze jest możliwy bez odchyłu. Spójrzmy do karty katalogowej mikrokontrolera ATmega32. Tabele 68–71 pokazują zależność odchyłu częstotliwości pracy portu szeregowego od częstotliwości taktowania. Zauważyć można, że wysoka częstotliwość pracy mikrokontrolera wcale nie gwarantuje niskiego bądź zerowego błędu. Gwarantują go pozornie dziwne częstotliwości taktowania, np. 3,6864 MHz. Można je bowiem wygodnie podzielić: 3 686 400 / 32 = 115 200.

Tabela 8

Nazwa bitu	Numer	Opis	Wartość domyślna
BODLEVEL	7	Poziom wyzwalania funkcji Brown-out Detector	1 (niezaprogramowany)
BODEN	6	Włącza BOD (Brown-out Detector,) automatyczny reset przy spadku napięcia zasilającego, np. wywołanym zwarciem	1 (niezaprogramowany, BOD wyłączony)
SUT1	5	Czas startu oscylatora	1 (niezaprogramowany)
SUT0	4	Czas startu oscylatora	0 (zaprogramowany)
CKSEL3	3	Wybór zegara	0 (zaprogramowany)
CKSEL2	2	Wybór zegara	0 (zaprogramowany)
CKSEL1	1	Wybór zegara	0 (zaprogramowany)
CKSEL0	0	Wybór zegara	1 (niezaprogramowany)

CS22	CS21	CS20	Taktowanie timera
0	0	0	Brak, timer wyłączony
0	0	1	clk/1
0	1	0	clk/8
0	1	1	clk/32
1	0	0	clk/64
1	0	1	clk/128
1	1	0	clk/256
1	1	1	clk/1024

Tabela 6

W taktowaniu mikrokontrolera oprócz samej wartości częstotliwości ważna jest też jej dokładność i stabilność. Budując zegar, skorzystamy z rezonatora kwarcowego, zamiast bazować na wewnętrznym generatorze RC o dokładności 3%.

Fusebity

Źródło taktowania, jak i kilka innych ustawień mikrokontrolera, przechowywane jest w dwóch 8-bitowych komórkach nieulotnej pamięci Flash. W dokumentacji mikrokontrolerów AVR bity tych komórek nazywane są angielskim słowem fuse (dosł. bezpiecznik), kolokwialnie spolszczanym jako fusy lub fusebity i fusebajty. Dla odróżnienia obu bajtów, jeden nazywany jest wysokim (Fuse High Byte, **tabela 7**), a drugi niskim (Fuse Low Byte, **tabela 8**). W nazewnictwie ważne są pojęcia zaprogramowania i niezaprogramowania bitu i mogą być one nieintuicyjne. Zaprogramowanie oznacza ustawienie bitu w stan 0, niezaprogramowanie w stan 1.

Wyboru sygnału zegarowego dokonujemy za pomocą fusebitów CKSEL3..0, zgodnie z **tabelą 9**.

W praktyce najczęściej korzystamy z rezonatora kwarcowego, gdy potrzebujemy dokładnego, stabilnego taktowania, lub też z wewnętrznego oscylatora RC, gdy dokładny sygnał zegarowy nie jest istotny. Zewnętrzny sygnał zegarowy jest używany bardzo rzadko, natomiast często bywa ustawiany przez pomyłkę.

Tabela 9

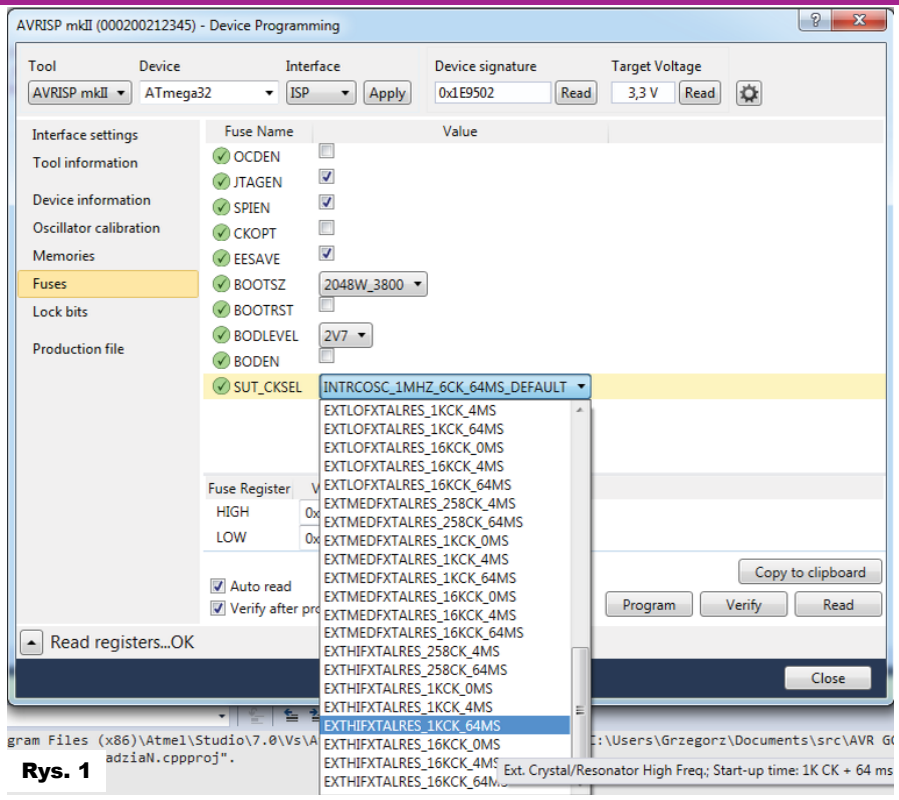
Źródło sygnału zegarowego	CKSEL3..0
Rezonator kwarcowy	1111 – 1010
Rezonator niskiej częstotliwości	1001
Zewnętrzny oscylator RC	1000 – 0101
Wewnętrzny oscylator RC	0100 – 0001
Zewnętrzny sygnał taktujący	0000

Jeśli zdecydujemy się na wewnętrzny generator RC, możemy wybrać jedną z czterech częstotliwości pracy, zgodnie z **tabelą 10**.

Ponieważ nie będzie wykorzystywany rezonator kwarcowy, fusebit CKOPT należy pozostawić niezaprogramowany (ustawiony na 1). Za pomocą fusebitów SUT ustawiamy opóźnienie, z jakim generator zacznie działać po włączeniu zasilania. Domyślnie wynosi ono 65 ms. Jeśli nasze źródło zasilania daje szybko narastające napięcie po włączeniu, opóźnienie można zmniejszyć do 4,1 ms. Gdy używana jest funkcja BOD, opóźnienia się nie stosuje. Zwykle używane będzie domyślne ustawienie (SUT1=1, SUT0=0) - **tabela 11**.

Wewnętrzny generator RC ma dokładność 3%. Dzięki dodatkowej kalibracji można osiągnąć 1%. Szczegóły tej operacji opisane są w dokumentach AVR053, AVR054 i AVR055 dostępnych na stronie firmy Atmel.

W przypadku rezonatorów kwarcowych o częstotliwości 1 MHz i wyżej, fusebity CKSEL3 oraz CKSEL0 powinny być ustawione na 1, pozostałe (CKSEL2 i CKSEL1) mogą być ustawione dowolnie. Ogólnie wszystkie fusebity CKSEL można ustawić na 1. Fusebit CKOPT musi być ustawiony na 0 (zaprogramowany). Z kolei fusebity SUT1 i SUT0 ustawia się nieco inaczej niż w przypadku wewnętrznego generatora RC, pokazuje to **tabela 12**. Zwykle najlepiej ustawić oba na 1.



Rys. 1

CKSEL3..0	Częstotliwość
0001	1 MHz
0010	2 MHz
0011	4 MHz
0100	8 MHz

Tabela 10

SUT1..0	Opóźnienie
00	-
01	4,1 ms
10	65 ms

Tabela 11

SUT1..0	Opóźnienie
01	-
10	4,1 ms
11	65 ms

Tabela 12

Atmel Studio pozwala ustawiać fusebity odpowiedzialne za taktowanie mikrokontrolera za pomocą gotowych szablonów, nie musimy ustawiać każdego bitu oddzielnie. W głównym oknie z menu Tools wybieramy znaną nam już pozycję Device Programming. Po połączeniu się z mikrokontrolerem przechodzimy do sekcji Fuses. Ostatnia pozycja na liście fusebitów to SUT_CKSEL. Znajduje się przy niej rozwijana lista gotowych kombinacji fusebitów SUT i CKSEL dla różnych rodzajów taktowania. Gdy zatrzymamy mysz nad daną opcją, wyświetli się jej opis. Zwykle korzystać będziemy z opcji z grupy INTRCOSC do taktowania wewnętrznym generatorem RC z wybraną częstotliwością oraz opcji EXTHIFXTALRES_16KCK_64MS dla rezonatora kwarcowego. Dla rezonatora kwarcowego trzeba jeszcze zaznaczyć bit CKOPT. Należy pamiętać, że w Atmel Studio zaznaczenie pola oznacza zaprogramowanie danego fusebitu, czyli ustawienie go na 0 (**rysunek 1**). Jeśli używamy makra F_CPU w naszych programach, po zmianie prędkości pracy mikrokontrolera musimy przestawić jego wartość w naszym projekcie, aby funkcje opóźniające nadal odmierzały prawidłowe czasy.

Gdy zmieniamy opcje taktowania, może się zdarzyć, że utracimy kontakt z mikrokontrolerem. Dzieje się tak, gdy korzystamy z programatora ISP i ustawimy taktowanie zewnętrznym sygnałem. Przy programatorze JTAG ten problem nie występuje, gdyż interfejs JTAG ma własny sygnał zegarowy. Rozwiązanie polega na dostarczeniu sygnału prostokątnego do nóżki XTAL1, o częstotliwości np. 1 MHz. Można w tym celu użyć nawet najprostszego generatora na jednej bramce logicznej. Na naszej płytki testowej mamy do dyspozycji generator na układzie NE555. Ma on stosunkowo niską częstotliwość, ok. 10 kHz przy niepodłączonym pinie GI, ale jeśli w okienku programowania, w sekcji Interface settings, przesuniemy suwak częstotliwości na najniższą częstotliwość i zatwierdzimy, klikając przycisk Set, to wystarczy. Następnie trzeba pin GO na złączu MISC połączyć z pinem XTAL1 (13) mikrokontrolera. Pin 13 nie jest wyprowadzony na płytce na żadne złącze, więc przewód połączeniowy trzeba będzie przytrzymać. Gdy połączenie jest gotowe, możemy zaprogramować właściwy zegar dla mikrokontrolera.

Zadania

1. Nasz prosty miernik częstotliwości działa wtedy, gdy dostaje sygnał. W przypadku jego braku wyświetla ostatni pomiar. Co zrobić, aby w takim przypadku świeciła pierwsza dioda, wskazująca niską częstotliwość, bliską zeru?
2. Poznaliśmy funkcje wszystkich trzech timerów mikrokontrolera ATmega32. Każdy z nich może generować sygnał PWM sterujący jasnością diody świecącej. Mamy więc możliwość niezależnego sterowania każdą z trzech diod wbudowanych w diodę RGB, dostępną na naszej płytce testowej i uzyskiwania różnych kolorów pośrednich. Zadanie polega na napisaniu programu pozwalającego dowolnie zmieniać wypadkowy kolor LED RGB z klawiatury.
3. Podobnie jak w zadaniu 2, ale z samoczynnym zmienianiem się koloru diody, w sposób mniej lub bardziej losowy.

Grzegorz Niemirowski
grzegorz@grzegorz.net