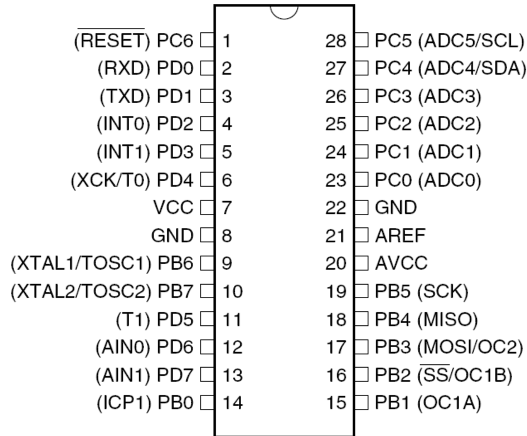
Podstawowe urządzenia peryferyjne mikrokontrolera ATmega8

Spis treści

1. Konfiguracja pinów....2
2. ISP....2
3. I/O Ports....3
4. External Interrupts....4
5. Analog Comparator....6
6. Analog-to-Digital Converter....6
7. USART....6
8. EEPROM....6
9. SPI....7
10. Timer/Counter....7
    10.1.Timer/Counter0....7
    10.2.Timer/Counter1....8
11. AVR Libc....9
    11.1.Funkcje opóźniające....9
    11.2.Pamięć Flash....10
    11.3.Stan uśpienia....10

1. Konfiguracja pinów

ATmega8 posiada 28 pinów, z czego 23 są programowalnymi pinami wejścia-wyjścia.



VCC	Zasilanie 4.5-5.5V lub 2.7-5.5V (wersja ATmega8L)
GND	Masa
AVCC	Zasilanie modułu A/D
AREF	Wejście/wyjście napięcia referencyjnego dla modułu A/D
PB5..0, PC5..0, PD7..0	Programowalne porty, które mogą posiadać kilka funkcji.
PC6 ($\overline{\text{RESET}}$)	Standardowo działa jako sygnał $\overline{\text{RESET}}$, może jednak być zaprogramowany przez odpowiedni fuse bit, jako zwykły port, ale zablokuje to późniejsze programowanie przez SPI. Sygnał jest aktywny stanem niskim, więc w czasie normalnej pracy ma być podawany sygnał wysoki. Zapewnia to wewnętrzny pull-up, dlatego reset nie musi zostać podłączony lub może zostać podłączony bezpośrednio do switch'a zwierającego do masy.
PB7..6 (XTAL2..1)	Standardowo działa jako zwykły port, ale po zaprogramowaniu odpowiednich fuse bitów można tutaj podpiąć zewnętrzny kwarc.

2. ISP

ISP (In-System Programming) umożliwia programowanie układu nawet już wewnątrz zbudowanego urządzenia. Programowanie najczęściej odbywa się przez interface SPI. Wymaga on podpięcia układu do programatora przez kilka przewodów:

$\overline{\text{RESET}}$	PC6	Reset układu, wymuszenie programowania
SCK	PB5	Zegar do transmisji
MISO	PB4	Wyjście danych
MOSI	PB3	Wejście danych
GND		Masa
VCC		Zasilanie układu (jeżeli układ nie posiada innego źródła zasilania)

Ten interface umożliwia programowanie pamięci programu (FLASH), pamięci EEPROM, lock bitów oraz fuse bitów. Lock bity programuje się raczej rzadko, głównie w urządzeniach przeznaczonych do sprzedaży, aby zablokować dostęp do pamięci programu. Czasami istnieje potrzeba zaprogramowania fuse bitów, najczęściej aby zmienić źródło lub częstotliwość zegara. Dokładny opis fuse bitów w rozdziale „Memory Programming” dokumentacji.

3. I/O Ports

Najczęściej używanymi peryferiami są porty wejścia-wyjścia. Można dzięki nim bezpośrednio sterować stanami logicznymi na portach lub je odczytywać. Do ich obsługi służy kilka rejestrów (gdzie *x* = B, C lub D; *n* = 7..0):

DDR*x*

7	6	5	4	3	2	1	0
DD*x*7	**DD*x*6**	**DD*x*5**	**DD*x*4**	**DD*x*3**	**DD*x*2**	**DD*x*1**	**DD*x*0**

Określa kierunek portu. 0 na bicie DD*xn* oznacza, że port P*xn* jest wejściem, 1 oznacza wyjście.

PORT*x*

7	6	5	4	3	2	1	0
PORT*x*7	**PORT*x*6**	**PORT*x*5**	**PORT*x*4**	**PORT*x*3**	**PORT*x*2**	**PORT*x*1**	**PORT*x*0**

PORT*xn* ma dwa znaczenia:
- gdy DD*xn* == 1 (wyjście), to bit PORT*xn* steruje wyjściem portu P*xn*.
- gdy DD*xn* == 0 (wejście) i PORT*xn* == 1, to zostaje włączony wewnętrzny rezystor pull-up.

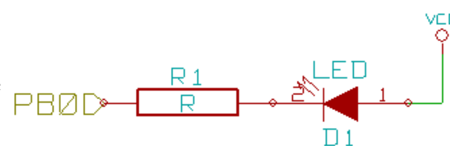
PIN*x*

7	6	5	4	3	2	1	0
PIN*x*7	**PIN*x*6**	**PIN*x*5**	**PIN*x*4**	**PIN*x*3**	**PIN*x*2**	**PIN*x*1**	**PIN*x*0**

PIN*xn* służy do odczytu stanu na porcie *xn*.

Przykłady

A. Chcielibyśmy zapalać i gasić diodę LED na porcie PB0.



Na początek ustawiamy bit DDB0 na 1, czyli port PB0 na wyjście (wystarczy wykonać to tylko raz).

```
DDRB |= (1<<DDB0);
```

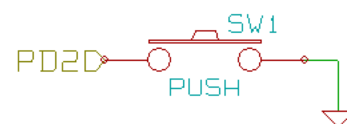
Następnie, gdy chcemy zapalić diodę ustawiamy na PB0 stan 0, aby płyną prąd, wiec PORTB0 na 0.

```
PORTB &= ~(1<<PORTB0);
```

Aby zgasić należy ustawić PORTB0 na 1.

```
PORTB |= (1<<PORTB0);
```

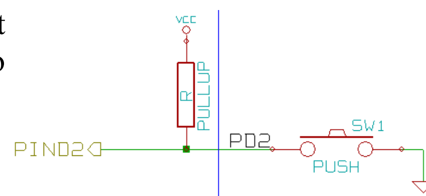
B. Odczytujemy wciśnięcie przycisku na porcie PD2.



Na początek ustawiamy bit DDD2 na 0, czyli port PD2 na wejście (wystarczy wykonać to tylko raz).

```
DDRD &= ~(1<<DDD2);
```

Musimy włączyć wewnętrzny pull-up, ponieważ, gdy przycisk jest rozłączony mielibyśmy nieokreślony stan (wystarczy wykonać to tylko raz).




```
PORTD |= (1<<PORTD2);
```

Teraz możemy sprawdzać czy przycisk został wciśnięty za pomocą bitu PIND2.

```
if (PIND & (1<<PIND2)) {
    // stan 1 na porcie => przycisk puszczony
} else {
    // stan 0 na porcie => przycisk wciśnięty
}
```

C. Odczytujemy wyjście z innego układu cyfrowego podłączonego do PD2.

Procedura wygląda tak samo jak dla przycisku, z tym wyjątkiem, że nie jest nam potrzebny pull-up, wiec ustawiamy 0 na PORTD2.

D. Zapisujemy bit na wejście innego układu cyfrowego.

Procedura wygląda tak samo jak dla zapalania LED'a.

4. External Interrupts

Niekiedy na pewne zdarzenia musimy zareagować natychmiast. Do tego służą przerwania. Gdy wystąpi określone zdarzenie procesor przerywa wykonywanie aktualnego kodu i przeskoczy natychmiast do określonej funkcji, gdzie programista może obsłużyć to zdarzenie. Po jej zakończeniu procesor powraca do poprzednio wykonywanego kodu. Przerwania mogą być wewnętrzne (generowane przez peryferia) lub zewnętrzne (generowane przez porty).

ATmega8 posiada dwa przerwania zewnętrzne: INT0 (PD2) oraz INT1 (PD3). Do ich obsługi służą trzy rejestry opisane w dokumentacji w rozdziale „External Interrupts”.

Przykład

Zliczmy ilość zmian stanu na porcie INT0 (PD2).

```
#include <avr/io.h>
#include <avr/interrupt.h>

volatile long i;

ISR(INT0_vect) {
    i++;
}

int main() {
    i = 0;
    DDRD &= ~(1<<DDD2);
    PORTD &= ~(1<<PORTD2);
    MCUCR = (MCUCR & (~(3<<ISC00))) | (1<<ISC00);
    GICR |= (1<<INT0);
    sei();
    while (1);
    return 0;
}
```

```
volatile long i;
```

Zmienna, która zawiera wartość licznika. Została zadeklarowana z atrybutem „volatile”, aby pominąć optymalizację. Zabezpiecza to przed np. taką sytuacją: W programie czekamy, aż licznik przekroczy 1000: `while (i<=1000);`. Kompilator podczas optymalizacji może przenieść zmienną *i* do rejestru, w tym czasie przerwanie aktualizuje zmienną *i* w RAM'ie, rejestr się nie zmienia, wiec pętla się będzie wykonywać bez końca.

```
ISR(INT0_vect) {
	i++;
}
```

Tak wygląda najprostsza deklaracja przerwania: zamiast typu, nazwy i parametrów funkcji wpisujemy ISR(*xxx*_vect), gdzie *xxx* to nazwa przerwania. W ciele funkcji przerwania zwiększamy zmienną i, nic nie zwracamy.

Więcej o przerwaniach jest na http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html.

```
	DDRD &= ~(1<<DDD2);
	PORTD &= ~(1<<PORTD2);
```

Ustawiamy port na wejście bez pull-up'a.

```
	MCUCR = (MCUCR & (~(3<<ISC00))) | (1<<ISC00);
```

Ustawiamy, aby przerwanie było wywoływanie na obu zboczach.

```
	GICR |= (1<<INT0);
```

Włączamy przerwanie INT0.

```
	sei();
```

Pozwalamy na wykonywanie przerwań. Po starcie wykonywanie przerwań jest zabronione. Możemy wywołać makro sei(), aby to zmienić. Makro cli() ponownie zabrania wykonywania. Po wywołaniu cli() nie zostanie wykonane żadne przerwanie, zaległe przerwania wykonają się dopiero po sei().

```
	while (1);
```

Nieskończona pętla. Wszystko robi za nas przerwanie.

UWAGA!!! W nowszych wersjach avr-gcc można spotkać bug'a w obsłudze przerwań. Ujawnia się w bardziej skomplikowanych funkcjach obsługi przerwania. Można go ominąć pisząc, np.:

```
ISR(INT0_vect) {
	asm volatile (
		"push r18\n push r19\n push r20\n"
		"push r21\n push r22\n push r23\n"
		"push r24\n push r25\n push r26\n"
		"push r27\n push r30\n push r31\n"
	);

	// Tutaj kod przerwania
	// ...

	asm volatile (
		"pop r31\n pop r30\n pop r27\n"
```

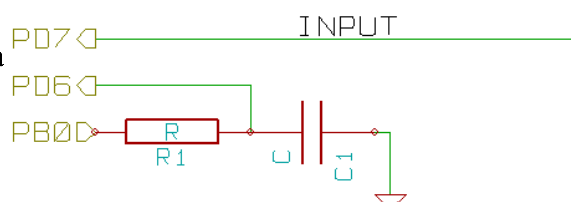
```
            "pop r26\n pop r25\n pop r24\n"
            "pop r23\n pop r22\n pop r21\n"
            "pop r20\n pop r19\n pop r18\n"
      );
}
```

5. Analog Comparator

Porównuje napięcia na analogowych wejściach AIN0 (PD6) i AIN1 (PD7). Najważniejszy bit to bit ACO w rejestrze ACSR, który jest wyjściem komparatora.

Przykład

Stabilizacja napięcia kondensatora C1. Kondensator ma mieć napięcie równe napięciu INPUT.




```
DDRD &= ~(1<<DDD6) & ~(1<<DDD7);
PORTD &= ~(1<<PORTD6) & ~(1<<PORTD7);
DDRB |= (1<<PORTB0);
while (1)
      if (ACSR & (1<<ACO)) {
            PORTB &= ~(1<<PORTB0);
      } else {
            PORTB |= (1<<PORTB0);
      }
}
```

Program porównuje oba napięcia na wejściach komparatora i w zależności od wyniku ładuje lub rozładowuje kondensator.

6. Analog-to-Digital Converter

...

7. USART

USART pozwala na komunikację przez interface szeregowy, np. RS-232 używany w komputerach PC (port COM*n*: w Windowsie).

Przykład

...

8. EEPROM

Pozwala na zapis nieulotnych danych w mikrokontrolerze. Najprościej użyć funkcji z biblioteki standardowej, bez wgłębiania się w rejestry. Ich opis znajduje się na http://www.nongnu.org/avr-libc/user-manual/group__avr__eeprom.html. Deklaracje znajdują się w <avr/eeprom.h>.

Przykład

Chcemy przechowywać w EEPROM'ie zadaną prędkość silnika. Definiujemy zmienną umiejscowioną w EEPROMie.

```
unsigned int EEMEM predkosc = 123;
```

Makro EEMEM informuje kompilator, że zmienna znajduje się w pamięci EEPROM. Tej zmiennej nie można używać jak zwykłej zmiennej. Jej odczyt i zapis należy realizować odpowiednimi funkcjami, np.:

```
unsigned int x = eeprom_read_word(&predkosc);
x++;
eeprom_write_word(&predkosc, x);
```

W efekcie zmienna predkosc zostanie odczytana z EEPROMa do zmiennej x, następnie zwiększona o 1. Na koniec nowa wartość zostanie zapisana z powrotem do pamięci EPPROM.

Po kompilacji zostanie wygenerowany plik z obrazem pamięci EEPROM (najczęściej z rozszerzeniem eep), który zawiera początkowe wartości zmiennych. W tym przykładzie plik zawiera zmienną predkosc z wartością 123.

9. SPI

...

10. Timer/Counter

ATmega8 posiada trzy tego typu peryferia. Każdy z nich jest niestety inny.

10.1. Timer/Counter0

Najprostszy z liczników zlicza od 0 do 255. Po zliczeniu do 255 zaczyna zliczanie od początku oraz ustawia flagę przepełnienia w rejestrze, która może być źródłem przerwania. Wartość licznika może być dowolnie zmieniana. Źródłem zegara może być główny zegar lub zewnętrzny – podłączony do T0 (PD4).

Obsługiwany jest czterema rejestrami:

TCCR0 – źródło zegara
- 0 – brak (licznik nie działa)
- 1 – zegar główny
- 2 – zegar główny / 8
- 3 – zegar główny / 64
- 4 – zegar główny / 256
- 5 – zegar główny / 1024
- 6 – port T0 na opadającym zboczu
- 7 – port T0 na narastającym zboczu

TCNT0 – aktualna wartość zegara
TIMSK (bit TOIE) – włącza przerwanie
TIFR (bit TOV0) – flaga, która zostanie ustawiona na 1 po przepełnieniu licznika, można ją wyzerować zapisując do niej 1.

Przykład

Chcemy stworzyć funkcję wait(long t), która czeka 8*t taktów zegara głównego.

```
void wait(long t) {
      long cnt = 0;
      TCCR0 = 2;
```

```
	TCNT0 = 0;
	TIFR |= (1<<TOV0);
	while (cnt < t) {
		if (TIFR & (1<<TOV0)) {
			cnt += 256;
			TIFR |= (1<<TOV0);
		}
		cnt = (cnt & ~255) | TCNT0;
	}
}
```

Zmienna cnt zawiera nasz licznik 32-bitowy. Kończymy funkcję, gdy licznik osiągnie wartość t.

Najpierw ustalamy źródło zegara, zerujemy licznik, czyścimy flagę przepełnienia. Wykonujemy pętlę, gdy cnt jest mniejszy niż t. Gdy nastąpi przepełnienie zwiększamy cnt o 256. Młodsze 8-bitów licznika cnt pochodzi z licznika TCNT0.

10.2. Timer/Counter1

Jest to zaawansowany 16-bitowy licznik, timer, generator PWM. Posiada wiele trybów pracy. Dokładny opis wszystkich trybów i rejestrów znajduje się w dokumentacji. Tutaj zostaną podane tylko przykłady kilku zastosowań.

Przykłady

A. Chcemy stworzyć funkcję wait(long t), która czeka 64*t taktów zegara głównego.

```
void wait(long t) {
	long cnt = 0;
	TCCR1A = 0;
	TCCR1B = 3<<CS10;
	TCNT1 = 0;
	TIFR |= (1<<TOV1);
	while (cnt < t) {
		if (TIFR & (1<<TOV1)) {
			cnt += 65536;
			TIFR |= (1<<TOV1);
		}
		cnt = (cnt & ~65535) | TCNT1;
	}
}
```

Został tutaj użyty tryb „Normal”, w którym licznik zachowuje się identycznie jak licznik 8-bitowy (zobacz poprzedni przykład), z tym, że teraz mamy dostępne 16 bitów, czyli wartości w zakresie 0..65535. Ustawienie tego trybu i wybranie zegara głównego / 64 znajduje się w tych dwóch liniach:

```
	TCCR1A = 0;
	TCCR1B = 3<<CS10;
```

Kilka rejestrów w tym timerze (w tym TCNT1) różni się od większości pozostałych tym, że są 16bitowe.

B. Naszym celem jest wytwarzanie fali prostokątnej o zadanej częstotliwości, np. aby podłączyć głośnik i odtwarzać jakąś melodię.

Do tego przykładu zostanie wykorzystany tryb CTC. W tym trybie licznik liczy od 0 do wartości zadanej rejestrem OCR1A. Po dojściu do niego przełącza stan na wyjściu OC1A. Zmieniając wartość OCR1A można sterować częstotliwością sygnału, wypełnienie zawsze 50%.

```
#define sound(hz, ms) hsound((unsigned int)((long)(F_CPU)/2l/(long)(hz)), (unsigned int)
((long)(ms)*(long)(hz)*2l/1000l))

void hsound(unsigned int t, unsigned int n)
{
```

```
	TCCR1B = 0;		// wyłączamy timer
	TCNT1 = 0;		// zerujemy licznik
	TIFR |= (1<<OCF1A); // czyścimy flagę porównania z OCR1A
	OCR1A = t;		// ustawiamy ilość cykli zegara na jedną połówkę okresu
	TCCR1A = (1 << COM1A0);	// ustawiamy tryb pracy
	TCCR1B = (1 << WGM12) | (1 << CS10);	// tryb pracy i włączenie timera
	while (n > 0) {
		if (TIFR & (1<<OCF1A)) {	// gdy licznik doliczy do wartości zadanej
			n--;			// zmniejsz ilość, która pozostała
			TIFR |= (1<<OCF1A);	// wyczyść flagę
		}
	}
	TCCR1B = 0;		// wyłącz licznik
}
```

Funkcja hsound wytwarza dźwięk o okresie *2*t* cykli zegara o długości *n/2* okresów. Makro sound przelicza odpowiednie jednostki i wywołuje funkcję hsound.

C. Sterujmy dwoma silnikami za pomocą PWM'a.

Tutaj będzie potrzebny któryś z trybów PWM. Zastosujemy Fast PWM 10-bitowy, który przy zegarze 1MHz daje częstotliwość 977 Hz i sterowanie od 0 do 1023. Wyjście to OC1A, OC1B.

Funkcja inicjalizująca uruchamia PWM'a i ustawia wypełnienia na 0.

```
void pwmInit()
{
	TCCR1B = 0;

	TCNT1 = 0;
	OCR1A = 0;
	OCR1B = 0;

	TCCR1A = (2 << COM1A0) | (2 << COM1B0) | (3 << WGM10);
	TCCR1B = (1 << WGM12) | (1 << CS10);
}
```

Funkcja pwmSet zmienia wypełnienie sygnału wyjściowego. Parametry: 0 – wyjście w stanie niskim, 1023 – wyjście w stanie wysokim, wartości pośrednie – sygnał prostokątny o zadanym wypełnieniu, inne wartości – niedozwolone.

```
void pwmSet(int a, int b) {
	OCR1A = a;
	OCR1B = b;
}
```

11. AVR Libc

AVR Libc jest standardową biblioteką c dla mikrokontrolerów AVR. Podręcznik można znaleźć na:

http://www.nongnu.org/avr-libc/user-manual/

Oprócz standardowych funkcji c biblioteka zawiera funkcję do obsługi AVRów. Niektóre z nich zostaną tutaj omówione. Inne zostały omówione wcześniej.

11.1. Funkcje opóźniające

Funkcje opóźniające zadeklarowane są w <util/delay.h>:

```
_delay_ms(double ms)	// Czeka podaną ilość milisekund
_delay_us(double us)	// Czeka podaną ilość mikrosekund
```

11.2. Pamięć Flash

...

11.3. Stan uśpienia

...