Mikrokontrolery Xmega (1)

Co trzeba wiedzieć, by zacząć

Mikrokontrolery AVR firmy Atmel zdobyły są bardzo popularne w Polsce. Dotychczas producent oferował nam dwie rodziny: ATtiny oraz ATmega, które różniły się możliwościami i ceną, choć sposób ich programowania był identyczny. Wprowadzając najnowszą rodzinę, Xmega, producent dokonał istotnych zmian w budowie mikrokontrolera oraz w sposobie pisania programów.

Prezentowany cykl artykułów ma na celu pokazanie różnic pomiędzy tradycyjnymi ATmega a nowymi Xmega oraz sposobu ich użycia je w rozwiązaniach praktycznych. Mam nadzieję, że zachęcę czytelników do używania tych fantastycznych procesorów.

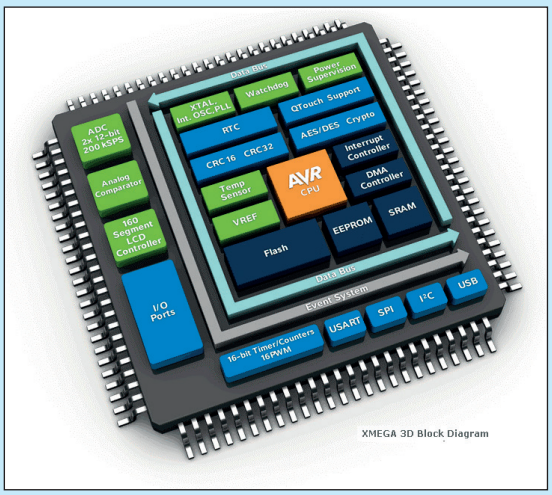
Podstawowe informacje – zegar i zasilanie

Napięcie zasilania wynosi od 1,6 V do 3,6 V. Częstotliwość taktowania rdzenia mikrokontrolera wynosi maksymalnie 32 MHz już przy napięciu zasilania 2,7 V. Przypomnijmy, że ATmega8 mogła „rozpędzić się" tylko do 16 MHz i potrzebowała do tego wyższego napięcia. Obniżenie napięcia zasilania pozwala na znaczące zmniejszenie poboru prądu przy zachowaniu dużej wydajności obliczeniowej. Na porty procesora można podać napięcie nie większe niż napięcie zasilania, więc należy uważać przy dołączaniu układów zasilanych napięciem 5 V.

Po włączeniu zasilania mikrokontroler uruchamia się korzystając z wbudowanego generatora RC o częstotliwości 2 MHz. Xmega ma dużo bardziej rozbudowany system dystrybucji sygnałów zegarowych niż tradycyjne AVR-y. Najważniejszą różnicą jest to, że już możemy zapomnieć o fusebitach! Cały system zegarowy możemy konfigurować w dowolnym momencie z poziomu programu. Na wypadek błędnej konfiguracji lub awarii, procesor jest w stanie wykryć nieprawidłowy sygnał zegarowy i przełączyć się samoczynnie na wbudowany generator 2 MHz, więc można eksperymentować bez obawy o jego zablokowanie. Użytkownik ma do dyspozycji następujące generatory sygnału zegarowego:

- wbudowany RC generujący przebiegi o częstotliwości: 32 kHz, 2 MHz lub 32 MHz,
- z zewnętrznym oscylatorem kwarcowym o częstotliwości od 32 kHz do 16 MHz.

Oprócz tego, mamy do dyspozycji układ PLL podwyższający częstotliwość źródła oraz szereg różnych preskalerów. Co ciekawe, niektóre peryferia mogą pracować z częstotliwością wyższą niż rdzeń procesora (32 MHz). Nic nie stoi na przeszkodzie, by procesor pracował z częstotliwością nawet 1 kHz, aby zmniejszyć pobór mocy, a potem przełączyć go na 32 MHz, gdy użytkownik będzie potrzebował szybkiego przetwarzania programu.

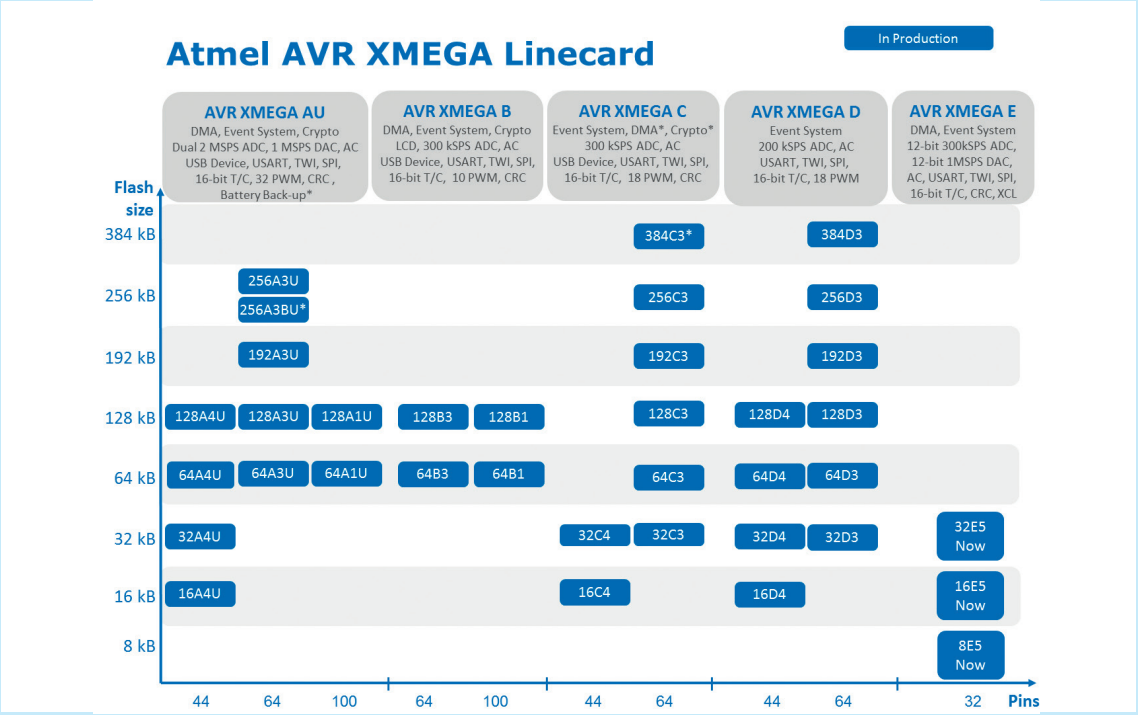


Dodatkowymi dwoma „cukiereczkami" poprawiającymi wydajność procesora jest system obsługi zdarzeń oraz DMA. Układy te pozwalają przetwarzać dane oraz przenosić je między peryferiami lub pamięcią bez zaangażowania rdzenia. Taka możliwość świetnie poprawia np. wydajność układów pomiarowych – przetwornik ADC może ładować dane prosto do pamięci, podczas gdy rdzeń może je analizować w tym samym czasie.

Dokumentacja

Przyzwyczailiśmy się, że Atmel publikuje jedną dokumentację, w której znajdziemy wszystko na temat jednego lub kilku podobnych mikrokontrolerów. W przypadku Xmega jest inaczej.

Poszukajmy na stronie Atmela dokumentacji mikrokontrolera ATxmega128A3U. Znajdziemy dwa dokumenty: *Atmel AVR XMEGA AU Manual* opisujący poszczególnych bloki funkcjonalne bez koncentrowania się na konkretnym modelu procesora oraz *ATxmega64A3U/128A3U/192A3U/256A3U Complete* zawierający sprecyzowane informacje specyficzne dla konkretnego typu mikrokontrolera. Dlaczego te informacje zostały rozdzielone? Mi-



Rysunek 1. Porównanie poszczególnych przedstawicieli rodziny XMEGA ze względu na pojemność pamięci, liczbę wyprowadzeń i układy peryferyjne

krokontrolery Xmega zaprojektowano z myślą o łatwym przenoszeniu programów. Dzięki temu oprogramowanie z ATxmega128A3U możemy łatwo przenieść na ATxmega16A4U, gdyż bloki funkcjonalne w nim zawarte są identyczne. Różnią się tylko ich liczbą, pojemnością pamięci oraz liczbą wyprowadzeń. Oba mikrokontrolery należą do rodziny „A" z rozszerzeniem USB, czyli „AU". Dlatego informacje o blokach funkcjonalnych, wspólne dla wszystkich mikrokontrolerów z tej rodziny, umieszczono w osobnej dokumentacji.

Warto zwrócić uwagę, że bloki funkcjonalne są wielokrotnie powielone oraz dostępne na wielu różnych portach procesora. Już nie ma problemów znanych z ATmega, kiedy np. wyjścia PWM wyprowadzone są na te same nóżki co jedyny interfejs SPI – w przypadku XMEGA po prostu wystarczy użyć SPI w jednym porcie, a PWM w drugim. Niektóre XMEGA mają nawet 8 interfejsów USART, 4×SPI, 4×I²C, 2 przetworniki A/C, 2×C/A oraz 8 timerów 16-bitowych! Różnice pomiędzy poszczególnymi modelami Xmega pokazano na **rysunku 1**.

Polecam też dokumenty z serii AVR1000, a przede wszystkim:

- AVR1000: Getting Started Writing C-code for XMEGA,
- AVR1001: Getting Started With the XMEGA Event System,
- AVR1003: Using the XMEGA Clock System,
- AVR1005: Getting started with XMEGA,
- AVR1305: XMEGA Interrupts,
- AVR1306: Using the XMEGA Timer/Counter,
- AVR1307: Using the XMEGA USART,
- AVR1308: Using the XMEGA TWI,
- AVR1309: Using the XMEGA SPI.

Tych poradników jest cała masa, a wyżej wymienione, to tylko wierzchołek góry lodowej. Zachęcam do poszperania na stronie Atmela, bo można tam znaleźć wiele przykładów i gotowych rozwiązań.

Skąd czerpać wiedzę?

W polskim Internecie pojawiły się już dwa cykle artykułów na temat mikrokontrolerów Xmega. To najlepszy sposób, by poznać podstawy ich działania i najszybciej nauczyć się je wykorzystywać. Zapraszam na kurs prowadzony na stronie *Leon Instruments* (*leon-instruments.blogspot.com*) oraz na świetnym blogu *Mikrokontrolery Jak Zacząć?* (*mikrokontrolery.blogspot.com*). Godna polecenia jest również najnowsza książka Tomasza Francuza pt. *AVR. Praktyczne projekty*. Autor jest znawcą tematu, a także świetnym pisarzem – w książce znajdują się podstawowe informacje zrozumiałe dla początkujących oraz nieco bardziej skomplikowane rzeczy dla osób chcących zagłębić się w szczegóły. Niezmiernie przydane są również krótkie i czytelnie opisane fragmenty kodu, pozwalające od razu przetestować opisywane peryferia na jakimś zestawie testowym.

Zestawy rozwojowe

Atmel przygotował kilka zestawów testowych o nazwie XMEGA XPLAINED. Dostępne są płytki z procesorami należącymi do różnych rodzin. W Internecie znajdziemy również sporo materiałów szkoleniowych przygotowanych przez producenta. Zestawy XPLAINED mają niewątpliwie największą wartość edukacyjną, jednak wadą jest ich stosunkowo wysoka cena.

Dobrym rozwiązaniem, będącym złotym środkiem pomiędzy kosztami i możliwościami jest płytka rozwojowa X3-DIL64 z procesorem ATxmega128A3U. Zaprojektowano ją tak, by wyglądem przypominała układ scalony w obudowie DIL64 w celu łatwego budowania prototypów przy użyciu tanich i popularnych płytek stykowych. Wszystkie wyprowadzenia procesora są dostępne dla użytkownika, a ponadto dostępny jest kompletny układ zasilający z USB, złącze do karty SD oraz do programatora PDI. Ważne! Użytkownik nie musi kupować programatora PDI, bo procesor ma fabrycznie wgrany bootloader FLIP, dzięki któremu można przesyłać programy przez

zwyczajny kabel USB! Warto też dodać, że przykłady opisane w książce „AVR. Praktyczne projekty" praktycznie bez zmian da się uruchomić na zestawie X3-DIL64, dzięki czemu stanowi on idealną pomoc dla osób rozpoczynających programowanie Xmega.

Programator

Mikrokontrolery Xmega nie posiadają interfejsu do programowania pamięci, takiego jak w ATmega i ATtiny, więc niestety może nas czekać zmiana programatora. Polecam następujące trzy rozwiązania.

FLIP – darmowy program do ściągnięcia ze strony producenta (*http://www.atmel.com/tools/FLIP.aspx*) umożliwiający wgrywanie programów za pomocą zwyczajnego kabla USB. Niestety, nie integruje się z Atmel AVR Studio i jest mało wygodny przy dużych projektach. Jednak na początek jest idealny, jako rozwiązanie najtańsze, bo kabelek USB ma każdy. Aby FLIP działał, procesor musi mieć wgrany bootloader FLIP – warto dodać, że procesor na płytce X3-DIL64 ma FLIP wgrany fabrycznie, a przycisk uruchamiający tryb programowania FLIP jest zamontowany na płytce.

AVR ISP mkII – programator, który interfejsy ISP, PDI i TPI, a więc umożliwiający zaprogramowanie wszystkich współczesnych mikrokontrolerów AVR. Znakomicie integruje się z Atmel Studio. Wystarczy nacisnąć F5, by program został przesłany do procesora. Na stronie AVT, Kamami i allegro jest dostępna cała masa różnorodnych klonów mkII za bardzo atrakcyjną cenę. Polecam ten programator wszystkim hobbystom.

AVR Dragon – hit od wielu lat, gdyż jako jedyny za rozsądną cenę udostępnia interfejs JTAG. Dzięki niemu można zajrzeć do środka procesora i zobaczyć, co się z nim dzieje. Jest to nieoceniona pomoc przy większych projektach, gdyż pozwala szybko i skutecznie znajdywać błędy w programach. AVR Dragon polecam dla profesjonalistów i bardziej zaawansowanych hobbystów.

Zaczynamy! – nowe metody konfigurowania rejestrów

Rejestry konfiguracyjne są pomostem, pomiędzy sprzętem a programem. Dla programu są to zmienne, posiadające adres w pamięci i jakąś wartość. Natomiast dla sprzętu zmiana zawartości rejestru oznacza zmianę jakiegoś napięcia, włączenie jakiegoś układu czy przeprowadzenie jakiejś operacji.

Porty są najprostszym układem peryferyjnym każdego mikrokontrolera. Mimo tego w Xmega do obsługi portu mamy aż… 21 rejestrów na każdy port! Wszystkich rejestrów konfiguracyjnych w procesorze może być kilkaset lub nawet ponad tysiąc! W tym artykule pokażę jak to „ogarnąć" i nie zwariować. Choć początek tej części artykułu może wydawać się trochę mętny – proszę się nie zniechęcać, bo w dalszej części zamieściłem proste przykłady praktyczne.

Wartości można wpisywać do rejestrów w sposób znany ze starszych mikrokontrolerów AVR, czyli NAZWA_REJESTRU = (1<<BIT1) | (1<<BIT2);. Jednak mając do dyspozycji kilkaset rejestrów, taki sposób jest niewygodny. Inżynierowie Atmela wpadli na pomysł, by do konfiguracji rejestrów wykorzystać struktury, przez co kod wygląda nico inaczej NAZWA_PERYFERIUM.NAZWA_REJESTRU = ...;. Tak jak napisałem, układy peryferyjne mikrokontrolerów Xmega są wielokrotnie powielone, a różnią się jedynie adresami rejestrów w pamięci oraz nazwą peryferium (PORTA, PORTB, PORTC…). Poza tym, wszystko jest identyczne. Można zatem wpisać jakąś wartość do rejestrów portów w ten sposób:

```
PORTA.DIR = ...;
PORTB.DIR = ...;
PORTC.DIR = ...;
PORTA.OUT = ...;
PORTB.OUT = ...;
PORTC.OUT = ...;
```

Pisanie kodu programu przy użyciu struktur niesie ze sobą bardzo ważną zaletę – raz napisany kod dla jakiegoś peryferium może być użyty do obsługi wszystkich jego kopii, zwanych instancjami, więc jeśli mamy do dyspozycji 8 interfejsów USART, to w przypadku starych AVR-ów funkcje obsługujące USART należałoby skopiować osiem razy i pozmieniać w nich nazwy rejestrów. W przypadku Xmega wystarczy napisać funkcję raz, a jako argument podać, jaki konkretnie układ peryferyjny nas interesuje. Niżej zamieszczono przykład, w jaki sposób można sterować różnymi portami przy pomocy jednej funkcji.

```
void UstawPort(PORT_t *nazwaportu) {
  nazwaportu->DIR = ...;
  nazwaportu->OUT = ...;
}
```

Jako argument funkcja przyjmuje nazwę portu. Sposób jej użycia wygląda następująco:

```
UstawPort(&PORTA);
UstawPort(&PORTB);
UstawPort(&PORTC);
```

Dzięki zastosowaniu funkcji operującej na strukturach, można znacząco zmniejszyć rozmiar programu. Choć w wypadku portów, sposób ten może wydawać się trochę bez sensu, to zapewniam, że przy bardziej skomplikowanych peryferiach taki sposób zdecydowanie przyspiesza pisanie programu.

Do rejestrów można wpisywać wartości heksadecymalne lub binarne, jednak jest to proszenie się o błędy i marnowanie czasu. Takich metod lepiej nie stosować!

Dopuszczalny jest sposób znany ze starych AVR-ów, wykorzystujący operator przesunięcia bitowego << oraz predefiniowane symbole z końcówką *_bp*, czyli *bit position*, określające numer bitu w rejestrze. Linijka programu konfigurująca kierunek pracy bitów 0…1 portu A może wyglądać następująco:

```
PORTA.DIR = (1<<PIN1_bp) | (1<<PIN0_bp);
```

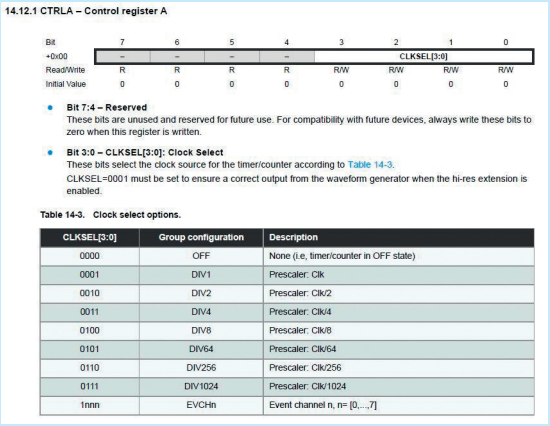
Dostępna jest nowa metoda, wykorzystująca predefiniowane symbole z końcówką *_bm*, czyli *bit mask*. Dzięki wyeliminowaniu „znaczków-krzaczków" zapis staje się bardziej czytelny:

```
PORTA.DIR = PIN1_bm | PIN0_bm
```

Bardziej skomplikowane peryferia mogą mieć kilka bitów odpowiedzialnych za realizację jakiegoś procesu. Dobrym przykładem jest tu źródło taktowania timera, wybierane za pomocą 4 bitów. Stosujemy w takim wypadku symbole z końcówką *_gc* (*group configuration*). Aby zilustrować przykład, zobaczmy możliwości ustawienia źródła taktowania i preskalera w timerze na **rysunku 2**. Zatem, by ustawić preskaler timera TCC0 na wartość 64, musimy do rejestru CTRLA wpisać odpowiednią grupę konfiguracyjną CLKSEL. Wszystko wyjaśnia przykład:

```
TCC0.CTRLA = TC_CLKSEL_DIV64_gc;
```

Atmel AVR Studio posiada bardzo przydatną funkcję przewidywania, co programista zamierza wpisać, przez co program podpowiada, jakie są dostępne możliwości, przed-



14.12.1 CTRLA – Control register A

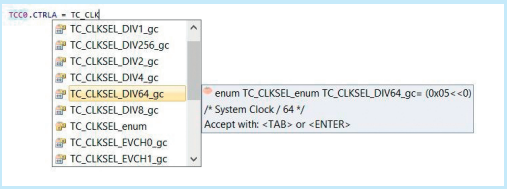
Bit	7	6	5	4	3	2	1	0
+0x00	–	–	–	–	CLKSEL[3:0]			
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

- **Bit 7:4 – Reserved**
  These bits are unused and reserved for future use. For compatibility with future devices, always write these bits to zero when this register is written.
- **Bit 3:0 – CLKSEL[3:0]: Clock Select**
  These bits select the clock source for the timer/counter according to Table 14-3.
  CLKSEL=0001 must be set to ensure a correct output from the waveform generator when the hi-res extension is enabled.

Table 14-3. Clock select options.

CLKSEL[3:0]	Group configuration	Description
0000	OFF	None (i.e, timer/counter in OFF state)
0001	DIV1	Prescaler: Clk
0010	DIV2	Prescaler: Clk/2
0011	DIV4	Prescaler: Clk/4
0100	DIV8	Prescaler: Clk/8
0101	DIV64	Prescaler: Clk/64
0110	DIV256	Prescaler: Clk/256
0111	DIV1024	Prescaler: Clk/1024
1nnn	EVCHn	Event channel n, n= [0,...,7]

Rysunek 2. Fragment dokumentacji ATxmega



Rysunek 3. Atmel AVR Studio przewiduje, co chcemy wpisać i wyświetla dostępne możliwości

stawioną na **rysunku 3**. Gdyby w rejestrze CTRLA było więcej bitów do skonfigurowania, poszczególne symbole *_gc*, *_bm* możemy oddzielić operatorem |. Dla zwiększenia czytelności kodu, można instrukcje podzielić na kilka linijek oraz opatrzyć je stosownym komentarzem.

```
PERYFERIUM.REJESTR = CONFIG1_gc |  // komentarz
                     CONFIG2_gc |  // komentarz
                     CONFIG3_bm |  // komentarz
                     CONFIG4_bm ;  // komentarz
```

Nic nie stoi na przeszkodzie, by predefiniowane symbole były argumentami funkcji. Na przykład, można napisać funkcję konfigurującą jakiś układ peryferyjny i wywoływać ją w ten sposób:

TimerInit(&TCCO, TC_CLKSEL_DIV64_gc, inne argumenty...);

TimerInit(&TCC1, TC_CLKSEL_DIV2_gc, inne argumenty...);

TimerInit(&TCD0, TC_CLKSEL_EVCH0_gc, inne argumenty...);

TimerInit(&TCD1, TC_CLKSEL_OFF_gc, inne argumenty...);

W ten sposób przy pomocy jednej funkcji TimerInit skonfigurowaliśmy cztery timery o nazwach TCC0, TCC1, TCD0, TCD1.

Elektroniczne Hello World, czyli mruganie diodą

Nadszedł czas, aby napisać wreszcie nasz pierwszy program! Port jest podstawowym peryferium, pozwalającym mikrokontrolerowi porozumieć się z innymi urządzeniami. W znanych i lubiany procesorach ATtiny i ATmega, każdy port miał trzy rejestry PIN, PORT oraz DDR. W mikrokontrolerach Xmega każdy port ma aż 21 rejestrów, ale bez obawy! Obsługa portów w Xmega jest łatwiejsza niż w ATmega!

Wszystkie porty mają swoją unikalną nazwę: PORTA, PORTB, PORTC… i tak dalej, aż do końca alfabetu! W obrębie każdego z nich znajduje się szereg rejestrów, a najważniejsze to:

DIR – rejestr ten decyduje czy dana nóżka ma być wejściem czy wyjściem. Wpisanie jedynki powoduje skonfigurowanie pinu jako wyjścia, a zero oznacza wejście. Dla przykładu, poniższa instrukcja PORTA.DIR = PIN3_bm | PIN6_bm; ustawi pin 3 oraz 6 jako wyjście, a pozostałe piny będą wejściami.

OUT – jest to rejestr wyjściowy. Wpisanie jedynki powoduje pojawienie się poziomu wysokiego na odpowiadającej nóżce portu, a wpisanie zera oznacza poziom niski.

IN – rejestr wejściowy, służący do odczytywania obecnego stanu pinów. Poniżej jest przykład instrukcji warunkowej, sprawdzającej czy na pinie E5 jest stan wysoki if(PORTE.IN & PIN5_bm);.

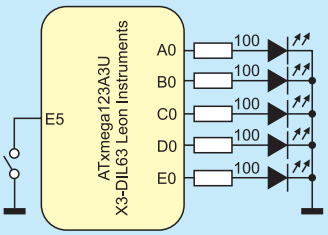
W mikrokontrolerach ATmega oraz ATtiny, aby ustawić lub wyzerować stan pojedynczego pinu w porcie, należało posłużyć się maskami bitowymi oraz operatorem |=. Dla przykładu, aby ustawić wyprowadzenie A1 oraz wyzerować A2, należało wydać następujące polecenia:

```
PORTA |= (1<<PA1);  // ustawienie pinu A1=1
PORTA &= ~(1<<PA2);  // ustawienie pinu A2=0
```

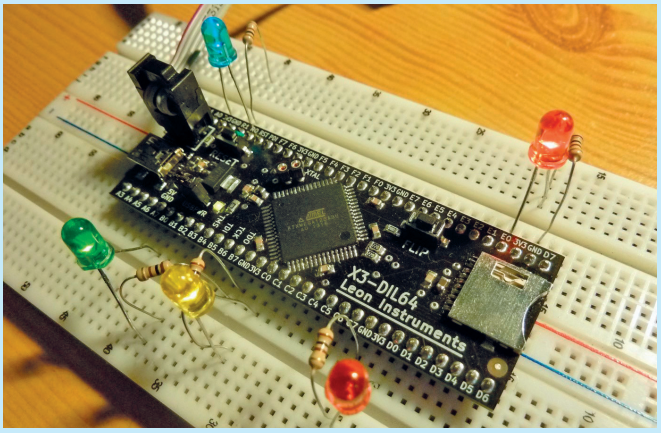
Nie jest to ani wygodne, ani szybkie w działaniu. Na szczęście projektanci procesorów Xmega wymyślili dużo lepszy i łatwiejszy sposób dostępu do wyprowadzeń. Podane instrukcje można zastąpić, wykorzystując rejestry *OUTSET* oraz *OUTCLR*:

```
PORTA.OUTSET = PIN1_bm;  // ustawienie pinu A1=1
PORTA.OUTCLR = PIN2_bm;  // ustawienie pinu A2=0
```

Istnieje też rejestr *OUTTGL* służący do zamiany stanu bitów portu, wskazanych w tym rejestrze. Mamy do dyspozycji także rejestry: *DIRSET*, *DIRCLR* i *DIRTGL*, które ustawiają, zerują lub zmieniają stan bitów odpowiedzialnych to czy wskazana nóżka ma być wejściem czy wyjściem. Ważne są rejestry *PINxCTRL* pozwalające na skonfigurowanie bardziej zaawansowanych opcji poszczególnych pinów. Co ważne, każdy pin ma osobny rejestr kontrolny. Wpisując do niego odpowiednie wartości, możemy włączać rezystory pull-up, pull-down, keeper. Za pomocą tych rejestrów konfi-



Rysunek 4. Schemat połączenia diod do płytki testowej. Przycisk połączony z pinem E5 już jest przylutowany na płytce X3-DIL64



Fotografia 5. Zdjęcie ilustrujące przykładowy sposób dołączenia diod LED

Listing 1. Konfigurowanie kierunku przepływu danych
```
PORTA.DIR = PIN0_bm;  // bit mask
PORTB.DIR = (1<<PORT0);  // po nazwie
PORTC.DIR = 0b00000001;  // wartość binarna
PORTD.DIR = 0x01;  // wartość szesnastkowa
PORTE.DIR = 1;  // wartość dziesiętna
```

Listing 2. Fragment pętli programu głównego
```
_delay_ms(500);  // czekanie 500ms (1)
PORTA_OUT |= (1<<PIN0_bp);  // ustawienie bitu po staremu (2a)
PORTB.OUTSET =  PIN0_bm;  // ustawienie bitu po nowemu (3a)
_delay_ms(500);  // czekanie 500ms (1)
PORTA_OUT &= ~(1<<PIN0_bp);  // zerowanie bitu po staremu (2b)
PORTB.OUTCLR =  PIN0_bm;  // zerowanie bitu po nowemu (3b)
```

Listing 3. Migotanie diod zależne od stanu przycisku
```
if(!(PORTE.IN & PIN5_bm))
{ // jeżeli przycisk wciśnięty
  toggle(&PORTD);
} else {  // jeżeli przycisk zwolniony
  toggle(&PORTE);
}
```

Listing 4. Przykładowy program dla Xmega
```
#define  F_CPU   2000000UL
#include <avr/io.h>
#include <util/delay.h>

// zamiana stanu pinu 0 wskazanego portu na przeciwny
void toggle(PORT_t *io)
{
  io->OUTTGL = PIN0_bm;
}

// różne sposoby na ustawienie pinu 0 każdego portu jako wyjście
int main(void)
{
  PORTA.DIR = PIN0_bm;  // bit mask
  PORTB.DIR = (1<<PORT0);  // po nazwie
  PORTC.DIR = 0b00000001;  // wartość binarna
  PORTD.DIR = 0x01;  // wartość szesnatkowa
  PORTE.DIR = 1;  // wartość dziesiętna
  // pin E5 jako wejście z podciągnięciem do zasilania
  PORTE.DIRCLR = PIN5_bm;
  PORTE.PIN5CTRL = PORT_OPC_PULLUP_gc;
  while(1)
  {
    _delay_ms(500);  // czekanie 500ms
    PORTA_OUT    |=  (1<<PIN0_bp);  // ustawienie bitu po staremu
    PORTB.OUTSET =   PIN0_bm;  // ustawienie bitu po nowemu
    _delay_ms(500);  // czekanie 500ms
    PORTA_OUT    &= ~(1<<PIN0_bp);  // zerowanie bitu po staremu
    PORTB.OUTCLR =   PIN0_bm;  // zerowanie bitu po nowemu
    toggle(&PORTC);
    if(!(PORTE.IN & PIN5_bm))  // jeżeli przycisk wciśnięty
      toggle(&PORTD);
    else  // jeżeli przycisk zwolniony
      toggle(&PORTE);
  }
}
```

guruje się także przerwania, szybkość narastania zbocza (*slew rate*) oraz kilka innych funkcji.

Do płytki X3-DIL podłączymy kilka diod LED, które będą mrugać z częstotliwością zależną od tego, czy jest wciśnięty przycisk E5 zamontowany na płytce (ten sam, który wykorzystuje się do programowania przez FLIP). Schemat układu pokazano na **rysunku 4**, a zdjęcie przedstawiono na **fotografii 5**.

W pierwszej kolejności musimy skonfigurować kierunek przepływu sygnałów przez piny w rejestrach *DIR* należących do odpowiednich portów (**listing 1**). Najpierw skonfigurujmy wyjścia A0, B0, C0, D0 oraz E0, do których dołączymy diody. Można to zrobić na różne sposoby – polecam sposób pierwszy z wymienionych. Wklepywanie wartości w kodzie szesnastkowym jest wysoce niewskazane, w szczególności przy konfigurowaniu bardziej skomplikowanych peryferiów. Dalej skonfigurujemy przycisk. Jest on przylutowany do nóżki E5 i zwiera ją do masy, kiedy jest wciśnięty. Kiedy jest zwolniony, pin E5 powinien mieć stan wysoki logiczny wymuszony rezystorem pull-up. W pierwszej linijce kodu zerujemy bit 5 w rejestrze DIR, poprzez wpisanie wartości *PIN5_bm* do rejestru *DIRCLR*. Następnie musimy włączyć rezystor podciągający za pomocą rejestru *PIN5CTRL*. Należy na to zwrócić szczególną uwagę, gdyż sposób włączania pull-upów w Xmega różni się od ATmega i ATtiny.

```
PORTE.DIRCLR = PIN5_bm;
PORTE.PIN5CTRL = PORT_OPC_PULLUP_gc;
```

Następnie przechodzimy do pętli głównej *while(1)*, która wykonuje się w nieskończoność. Mruganie diodami również zrealizujemy na kilka sposobów. Przeanalizujmy program z **listingu 2**.

Polecenie _delay_ms(500) powoduje czekanie przez pół sekundy. Następnie ustawiamy pin A0 oraz B0 w stan wysoki. Widać wyraźnie, że nowy sposób sterowania pinami, opisany w linijce 3a jest zdecydowanie bardziej zwięzły i czytelny. Co ważniejsze, jest również szybciej wykonywany i zajmuje mniej miejsca w pamięci, jako że stosujemy operator = zamiast |=. W dalszej części kodu zerujemy piny A0 oraz B0 i sposobem typowym dla ATmega oraz z XMEGA.

Możemy zrealizować mruganie diodą jeszcze inaczej. Można wywołać np. funkcję *toggle(&PORTC);*. Jej definicja wygląda następująco:

```
// zamiana stanu pinu 0 wskazanego portu na przeciwny
void toggle(PORT_t *io)
{
 io->OUTTGL = PIN0_bm;
}
```

Przyjmuje ona jako argument nazwę portu i zamienia stan ostatniego bitu na przeciwny za pomocą wpisania wartości *PIN0_bm* do rejestru *OUTTGL* wskazanego portu. Funkcji tej można użyć w następnych przykładach.

Niech dwie kolejne diody mrugają z różną częstotliwością, w zależności czy przycisk jest wciśnięty czy nie. Przykład realizacji takiego programu pokazano na **listingu 3**.

Instrukcja *PORTE.IN & PIN5_bm* sprawdza czy obecna jest jedynka logiczna na piątej pozycji w rejestrze *IN*. Jeśli tak, to instrukcja zwraca wartość prawdziwą. Jednak pamiętajmy, że pin E5 ma włączony rezystor pull-up, więc stanem domyślnym jest logiczna jedynka, a wciśnięcie przycisku powoduje zwarcie pinu do masy i tym samym pojawienie się zera. Dlatego wyrażenie to zostało zanegowane za pomocą operatora negacji !. Następnie wywołujemy znaną już funkcję *toggle*, która za argument przyjmuje PORTE, kiedy przycisk jest wciśnięty lub PORTD, kiedy przycisk jest zwolniony. Kod całego programu zamieszczono na **listingu 4**.

Podsumowanie

Jednak to nie wszystkie możliwości portów, a jedynie wierzchołek góry lodowej. Oprócz tego, porty w Xmega mają jeszcze inne możliwości, takie jak:

- kontrola szybkości narastania zbocza (slew rate),
- remapowanie pinów,
- konfiguracje pull-up, pull-down, keeper, wired or, wired and,
- zgłaszanie przerwań,
- generowanie zdarzeń,
- porty wirtualne.

Funkcje te są opisane w książce Tomasza Francuza *AVR. Praktyczne przykłady* i można je wygodnie przetestować korzystając z płytki rozwojowej X3-DIL64 Leon Instruments.

Dominik Leon Bieczyński
http://leon-instruments.blogspot.com