Kurs programowania mikrokontrolerów XMEGA (8)

Użycie bloku DMA

Mikrokontrolery XMEGA mają tak dużo peryferiów, że rdzeń procesora potrzebuje dwóch dodatkowych układów wspomagających, by móc je efektywnie wykorzystać. Pierwszy z nich to system zdarzeń, który poznaliśmy w EP 2014/03. Służy on do przekazywania prostych sygnałów logicznych pomiędzy peryferiami. Drugi to układ DMA (Direct Memory Access), który służy do przesyłania danych bez udziału rdzenia procesora.

Ponieważ DMA nie występowało w starych układach ATtiny oraz ATmega i zapewne jest nowością dla wielu czytelników – omówię dokładnie jak wygląda przesyłanie danych w pamięci mikrokontrolera i w jaki sposób DMA poprawia efektywność tego procesu.

Organizacja pamięci XMEGA

Układy AVR mają dwie oddzielne przestrzenie adresowe – osobną dla pamięci programu (tzw. ROM, choć tak naprawdę jest to pamięć Flash wielokrotnego zapisu) oraz oddzielną dla pamięci RAM i układów peryferyjnych. Mamy zatem dwie niezależne magistrale adresowe i dwie magistrale danych (zobacz **rysunek 1**). Do pamięci programu podłączony jest jedynie dekoder instrukcji procesora i z tej pamięci procesor praktycznie non-stop pobiera instrukcje. Inaczej wygląda sytuacja z pamięcią danych – tutaj transfery zachodzą tylko wtedy, kiedy procesor chce uzyskać dostęp do jakiejś zmiennej lub rejestrów układów peryferyjnych. Są więc sytuacje, kiedy magistrala adresowa i magistrala danych pamięci RAM jest niewykorzystywana.

Jak wygląda operacja skopiowania komórki pamięci z jednego adresu pod inny adres? Każda komórka pamięci w procesorze ma swój unikalny adres, który w przypadku XMEGA może być 24-bitowy. Procesor na magistralę adresową wystawia adres komórki, która ma być odczytana i wysyła żądanie odczytu. Pamięć lub układ peryferyjny udostępnia żądany bajt informacji na magistrali danych. Procesor kopiuje ten bajt do jednego ze swoich rejestrów roboczych R0-R31. Następnie procesor na magistralę adresową wypisuje adres komórki, do której mają trafić dane. Na magistralę danych kopiuje wartość rejestru R0-R31, który przechowuje interesujące nas informacje, po czym wysyła sygnał zapisu. Adresy komórek źródłowych i docelowych również są przechowywane w rejestrach roboczych R0-R31.

Skomplikowane? Niekoniecznie. Jest to dość prosta operacja, ale ma dwie wady. Trwa dość długo i całkowicie pochłania rdzeń procesora, przez co nie może on wykonywać żadnych innych operacji. Kopiowanie dużych bloków pamięci może na długi czas zablokować procesor. W takiej sytuacji pomocny jest układ DMA.

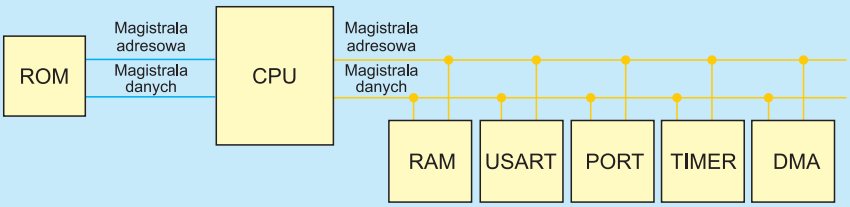
DMA jest układem bardzo prostym w działaniu. Jego zadaniem jest skopiowanie określonej liczby bajtów z jednego miejsca w drugie miejsce. W tym czasie procesor może wykonywać inne operacje całkowicie bez utraty funkcjonalności. Co się stanie w sytuacji, kiedy procesor będzie chciał uzyskać dostęp do pamięci, w czasie kiedy DMA transferuje dane? Wówczas praca DMA zostanie automatycznie zawieszona, a kiedy procesor zakończy swoje działanie, DMA będzie kontynuować pracę.

Do czego może się przydać DMA?

Oprócz zastosowań trywialnych, jak kopiowanie danych z jednej tablicy do drugiej, DMA bardzo dobrze współpracuje z innymi układami peryferyjnymi. DMA potrafi przesyłać dane z tablicy do układu interfejsowego USART czy SPI, dzięki czemu całkowicie sprzętowo i bardzo szybko można przesłać duży blok danych pomiędzy urządzeniami. W podobny sposób istnieje możliwość odbierania danych z układów transmisyjnych. DMA równie dobrze współpracuje z przetwornikiem analogowo-cyfrowym i cyfrowo-analogowym. Na przykład, pomiar przetwornikiem może być inicjowany timerem w zadanych odstępach czasowych, a po zakończeniu pomiaru, DMA może kopiować wyniki pomiaru z przetwornika do tablicy, którą procesor przetwarza w czasie rzeczywistym. Przypomina to układ akwizycji z oscyloskopu? A jakże! Równie łatwo można zrobić generator DDS. Zastosowanie układu DMA ogranicza jedynie wyobraźnia programisty, a żeby tę wyobraźnię rozbudzić, warto przeczytać książki Tomasza Francuza, w których opisuje najróżniejsze zastosowania DMA w praktycznych przykładach.

Konfigurowanie DMA

Mikrokontroler ATxmega128A3U wyposażony jest w cztery kanały DMA, pracujące niezależnie od siebie.



Rysunek 1. Organizacja pamięci w mikrokontrolerach XMEGA

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 76305, pass: 8741rnfv

Listing 1. Kod programu do pierwszego ćwiczenia

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

uint8_t source[10] = {10,11,12,13,14,15,16,17,18,19};
uint8_t dest[15];

int main(void) {

    // konfiguracja kontrolera DMA
    DMA.CTRL  =  DMA_ENABLE_bm|  //włączenie kontrolera
                 DMA_DBUFMODE_DISABLED_gc|  //bez podwójnego buforowania
                 DMA_PRIMODE_RR0123_gc;  //wszystkie kanały równy priorytet
    // konfiguracja kanału DMA
    DMA.CH0.SRCADDR0  =  (uint16_t)source & 0xFF;  //adres źródła
    DMA.CH0.SRCADDR1  =  (uint16_t)source >> 8;
    DMA.CH0.SRCADDR2  =   0;

    DMA.CH0.DESTADDR0  =  (uint16_t)&dest[12] & 0xFF;  //adres celu
    DMA.CH0.DESTADDR1  =  (uint16_t)&dest[12] >> 8;
    DMA.CH0.DESTADDR2  =   0;
    DMA.CH0.TRFCNT     =  sizeof(source);  //rozmiar tablicy
    DMA.CH0.ADDRCTRL   =  DMA_CH_SRCRELOAD_NONE_gc|  //przeładowanie adresu źródła po zakończeniu bloku
                          DMA_CH_SRCDIR_INC_gc|  //zwiększanie adresu źródła po każdym bajcie
                          DMA_CH_DESTRELOAD_NONE_gc|  //przeładowanie adresu celu po zakończeniu bloku
                          DMA_CH_DESTDIR_DEC_gc;  //zmniejszenie adresu celu po każdym bajcie
    DMA.CH0.CTRLA      =  DMA_CH_ENABLE_bm|  //włączenie kanału
                          DMA_CH_TRFREQ_bm|  //uruchomienie transmisji
                          DMA_CH_BURSTLEN_1BYTE_gc;  // burst = 1 bajt
    // pusta pętla główna
    while(1) {}
}
```

W jednej chwili może pracować tylko jeden kanał o najwyższym priorytecie, a praca pozostałych jest zawieszana. Każda operacja kopiowania danych przez DMA w dokumentacji procesora nazywana jest **transakcją**. Transakcję można podzielić na poszczególne **bloki**, a te dzielą się na transfery **burst**. Jest to spowodowane koniecznością przerwania pracy układu DMA, kiedy do pamięci dostęp chce uzyskać procesor.

Burst jest najbardziej elementarną częścią transferu i może mieć długość 1, 2, 4 lub 8 bajtów. Jest to fragment transmisji, której nie można przerwać – jeśli procesor będzie chciał uzyskać dostęp do pamięci podczas transmisji burst, będzie mógł to zrobić dopiero po jej zakończeniu. Nie ma sensu ustalać długości burst większej niż rzeczywiście potrzebna, ponieważ może to niepotrzebnie blokować procesor i obniżać jego wydajność, zamiast ją podwyższać. Powinniśmy wybrać taką długość transmisji burst, jaką maja typ kopiowanych danych. Tzn. dla danych tekstowych ASCII, przechowywanych w zmiennych 8-bitowych typu char lub uint8_t, powinniśmy wybrać burst o długości 1 bajta. W przypadku kopiowania danych z przetwornika cyfrowo-analogowego, przechowywanych w dwóch rejestrach 8-bitowych, powinniśmy wybrać burst 2-bajtowy.

Blok jest jednym burstem lub określoną ilością transmisji burst następujących po sobie. Transmisja bloku może zostać zainicjalizowana automatycznie po wystąpieniu odpowiedniego wyzwalacza, pochodzącego z układu USART, SPI lub przetwornika analogowego. W obrębie bloku można przesłać maksymalnie 65536 bajtów.

Transakcja jest zbiorem bloków, obojętnie czy przesyłanych bezpośrednio po sobie, czy z jakimiś przerwami. Istotne jest to, że ustawienia układu DMA podczas wykonywania transakcji są stałe i nie mogą się zmieniać. Transakcja może składać się z 256 bloków, co przy maksymalnym rozmiarze bloku oznacza, że DMA może przesłać w jednej transakcji nawet 16 MB.

Może się to wydawać trochę zagmatwane, jednak odrobina ćwiczeń i praktyki rozwieje wątpliwości. Praktyczne przykłady, które omówimy w tym kursie, zilustrują zastosowanie poszczególnych ustawień układu DMA.

Przesyłanie tablicy

W pierwszym przykładzie napiszemy bardzo prosty program, w którym będą dwie tablice, a korzystając z DMA, skopiujemy zawartość jednej tablicy do drugiej. Tablica źródłowa source[] będzie miała 10 elementów, a tablica docelowa dest[] będzie składać się z 15 elementów. Żeby przykład nie był zbyt trywialny, podczas kopiowania odwrócimy kolejność danych. Kod programu przedstawia **listing 1**, a jego działanie będziemy mogli zaobserwować przy pomocy debugera JTAG albo symulatora. Przy okazji poznamy kilka ciekawych opcji Atmel Studio, pozwalających na zaglądanie do wnętrza procesora podczas pracy, dzięki czemu można na własne oczy zobaczyć pracę programu, co pozwala łatwo i szybko znaleźć błędy. Jeśli nie masz JTAG – to jeszcze nie problem! Program możesz przetestować w symulatorze Atmel Studio. Mimo to, polecam zaopatrzenie są w programator JTAG, np. AVR Dragon, który jest warty swojej ceny. Czas zaoszczędzony na szukaniu błędów bardzo szybko rekompensuje cenę programatora JTAG.

Aby skorzystać z dobrodziejstw DMA, musimy najpierw ustawić jego kontroler, a dopiero potem poszczególne kanały. Ustawienie kontrolera jest bardzo proste i polega na wpisaniu do rejestru DMA.CTRL odpowiednich wartości:

- DMA_ENABLE_bm – włączenie kontrolera DMA.
- DMA_DBUFMODE_xxx_gc – grupa konfiguracyjna odpowiadająca za konfigurację podwójnego buforowania. Za xxx można wpisać DISABLED, CH01, CH23 lub CH01CH23. W naszych przykładach z podwójnego buforowania korzystać nie będziemy, więc wybieramy opcję DISABLED.
- DMA_PRIMODE_xxx_gc – grupa konfiguracyjna ustalająca priorytety kanałów. Domyślnie wszystkie kanały mają równy priorytet, a który z nich ma mieć pierwszeństwo w przypadku jednoczesnej pracy, określa algorytm Round Robin. Oznacza to, że ten, który ostatnio był używany, trafia na koniec kolejki i musi czekać aż pozostałe kanały zakończą pracę, po czym cykl się powtarza. Tryb taki uzyskujemy po wpisaniu RR0123 w miejsce xxx. Możliwe jest, by wybrane kanały o najniższych numerach miały wyższy priorytet. Po wpisaniu CH0RR123 kanał 0 będzie miał

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 76305, pass: 8741rnfv

priorytet najwyższy, natomiast 1, 2 i 3 będą działać na zasadzie Round Robin. Możliwe jest też ustawienie CH01RR23 lub CH0123, gdzie nie ma Round Robin, a priorytety kanałów ustawione są na sztywno.

Następnie przechodzimy do konfiguracji kanału 0 w rejestrach DMA.CH0, gdzie musimy ustalić adresy tablicy źródłowej i docelowej. Magistrala adresowa w XMEGA ma szerokość 24 bitów, zatem adresy musimy wpisywać do trzech rejestrów, przechowujących adres źródła danych: SRCADDR0, SRCADDR1, SRCADDR2 oraz do trzech rejestrów przechowujących adres docelowy: DESTADDR0, DESTADDR1, DESTADDR3. Sposób wpisywania adresów jest dość „fikuśny". Jeśli chcemy podać adres początku tablicy, wystarczy wpisać jej nazwę bez nawiasów klamrowych. W przypadku zwykłej zmiennej, rejestru lub innego elementu tablicy niż pierwszy, musimy posłużyć się operatorem pobrania adresu &. W obu przypadkach adres musimy rzutować na zmienną 16-bitową i na końcu wyciągnąć z niej młodszy i starszy bajt. Najlepiej będzie spojrzeć na kod programu na **listingu 1**, gdzie zostało to przedstawione. Dobrze jest po prostu zapamiętać pewien szablon kodu, w którym podajemy adresy celu i źródła dla DMA.

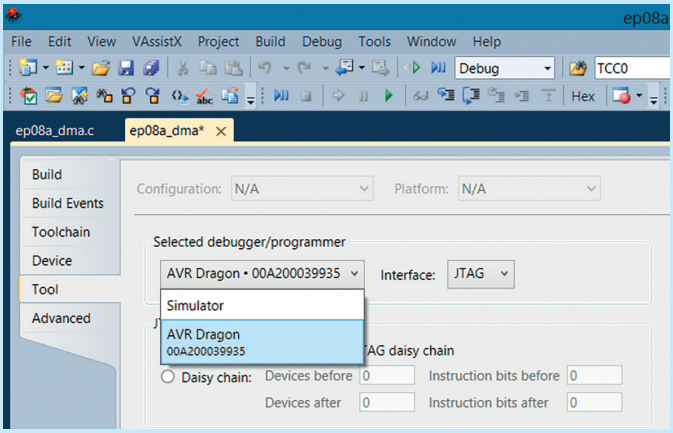
```
    DMA.CH0.SRCADDR0      =     (uint16_t)
source & 0xFF;          // adres źródła
    DMA.CH0.SRCADDR1      =     (uint16_t)
source >> 8;
    DMA.CH0.SRCADDR2      =     0;
    DMA.CH0.DESTADDR0     =
(uint16_t)&dest[12] & 0xFF;      // adres
celu
    DMA.CH0.DESTADDR1     =
(uint16_t)&dest[12] >> 8;
    DMA.CH0.DESTADDR2     =     0;
```

Kiedy chcemy podać początek tablicy, wystarczy wpisać jej nazwę bez żadnych nawiasów ani innych ozdobników. Chcemy jednak, by kolejność danych została odwrócona, więc musimy zapisywać tablicę od tyłu – w naszym przykładzie będzie to od 12 elementu, dlatego w kodzie programu do rejestrów DESTADDR wpisujemy adres &dest [12] (jest to de facto trzynasty element, ponieważ w C elementy numeruje się od zera, zatem nasza 15-elementowa tablica ma elementy o numerach 0-14).

Kolejnym krokiem jest określenie, ile bajtów zamierzamy przesłać i wpisać tę wartość do rejestru DMA.CH0. TRFCNT. Warto tutaj się posłużyć operatorem sizeof() i jako argument podać nazwę tablicy (uwaga – choć sizeof() wygląda jak funkcja, w rzeczywistości jest to operator działający na etapie kompilacji programu; użycie sizeof() nie jest możliwe w przypadku tablic o zmiennym rozmiarze z dynamiczną alokacją pamięci).

W rejestrze DMA.CH0.ADDRCTRL musimy ustalić, w jakim kierunku będą kopiowane dane. Możliwe są trzy opcje:

1. Dane kopiowane są zawsze z/do tej samej komórki pamięci. Ma to zastosowanie, kiedy DMA współpracuje z jakimś układem peryferyjnym – wpisujemy DMA_CH_SRCDIR_FIXED_gc.
2. Adres źródłowy/docelowy zwiększa się po każdym przesłanym bajcie. W ten sposób kopiujemy dane od początku do końca – DMA_CH_SRCDIR_INC_gc.
3. Adres źródłowy/dolecowy zmniejsza się po każdym bajcie. Przez to możemy tablicę zapisywać od końca do początku – DMA_CH_SRCDIR_DEC_gc.



Rysunek 2. Ustawienia programatora JTAG

Nic nie stoi na przeszkodzie, by tablicę źródłową odczytywać od początku, a docelową zapisywać od końca. W tym samym rejestrze określamy też, kiedy ma nastąpić przeładowanie rejestrów adresowych, tzn. przywrócenie wartości początkowej. Ponieważ w tym przykładzie interesuje nas pojedyncza transakcja, nie będziemy korzystać z możliwości przeładowania.

Ostatnim rejestrem jest DMA.CH0.CTRLA, do którego wpisujemy odpowiednio:

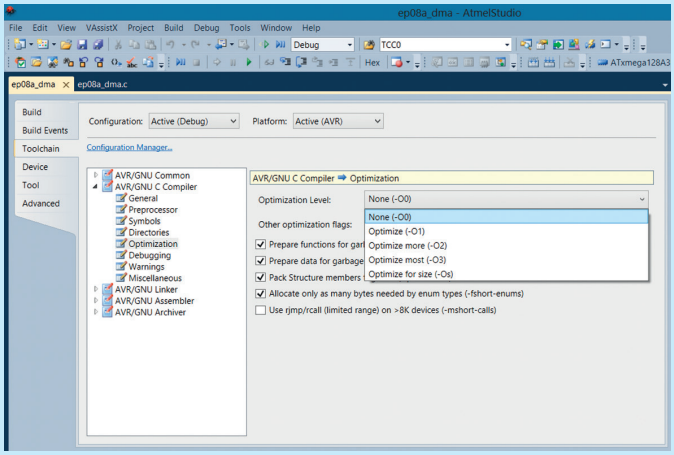
- DMA_CH_ENABLE_bm – uruchomienie kanału.
- DMA_CH_TRFREQ_bm – ustawienie tego bitu powoduje uaktywnienie transmisji, jeśli nie wybraliśmy wcześniej automatycznego wyzwalacza. W ten sposób można programowo sterować układem DMA, kiedy ma się rozpocząć kopiowanie.
- DMA_CH_BURSTLEN_xBYTE_gc – ustalamy długość transmisji burst, na 1, 2, 4 lub 8 bajtów. W przypadku, kiedy nasze tablice przechowują zmienne 8-bitowe, więc wybieramy burst o wielkości 1 bajtu.

To już wszystko! Dalej musi być tylko pusta pętla while(1), a cała transmisja zostanie zrealizowana sprzętowo.

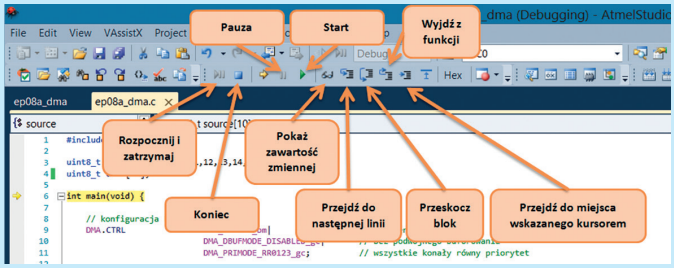
Przetestujemy działanie programu przy pomocy programatora JTAG lub poprzez symulator wbudowany w Atmel Studio. Przy próbie uruchomienia debugowania, po wciśnięciu klawisza F5, powinien pojawić się komunikat, że nie wybrano programatora. Jeśli takie okienko się nie pojawiło, wybieramy z menu Project → Properties i w zakładce Tools wybieramy Simulator lub posiadany programator JTAG (ja wybrałem AVR Dragon). W przypadku programatorów, trzeba jeszcze wybrać, poprzez który interfejs procesor ma być połączony. Wybieramy oczywiście JTAG. Przedstawiono to na **rysunku 2**.

Choć w przypadku tak prostego programu nie ma to znaczenia, to warto wyrobić sobie zwyczaj dostosowywania optymalizacji kodu do naszych wymagań. W tym samym oknie należy otworzyć zakładkę Toolchain, następnie z drzewka wybierz AVR/GNU C Compiler, a potem Optimalization. Domyślnie włączona jest opcja –O1, stanowiąca kompromis pomiędzy wielkością kodu wynikowego a szybkością działania programu. Kiedy zależy nam na oszczędzaniu miejsca warto wybrać opcję –Os. Optymalizator zastosuje różne sztuczki, aby kod wynikowy był jak najbardziej zwarty. Do debugowania przez JTAG warto jednak wyłączyć wszelkie optymalizacje, gdyż optymalizator potrafi pozmieniać kolejność wykonywania

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 76305, pass: 8741rnfv



Rysunek 3. Wybór poziomu optymalizacji kodu



Rysunek 4. Przyciski sterujące pracą krokową programu

linii, co może nas niepotrzebnie mylić, więc wybieramy opcję –O0. Kod programu będzie wtedy relatywnie duży. Oczywiście, jeśli zakończymy debugowanie i będziemy chcieli uzyskać końcową wersję programu, możemy wtedy zmienić poziom optymalizacji na inny. Właściwe ustawienia pokazano na **rysunku 3**.

Debugowaniem sterują przyciski z górnego paska narzędzi, opisana na **rysunku 4**. Warto nauczyć się ich skrótów klawiaturowych.

Aby zobaczyć na żywo, jak DMA kopiuje poszczególne komórki, wystartujmy program z natychmiastowym zatrzymaniem go w pierwszej linijce. Aby to zrobić, klikamy na przycisk Rozpocznij i zatrzymaj lub naciskamy Alt-F5. Aktualnie wykonywana linijka zostanie podświetlona na żółto, a po prawej stronie pojawią się dodatkowe okna:

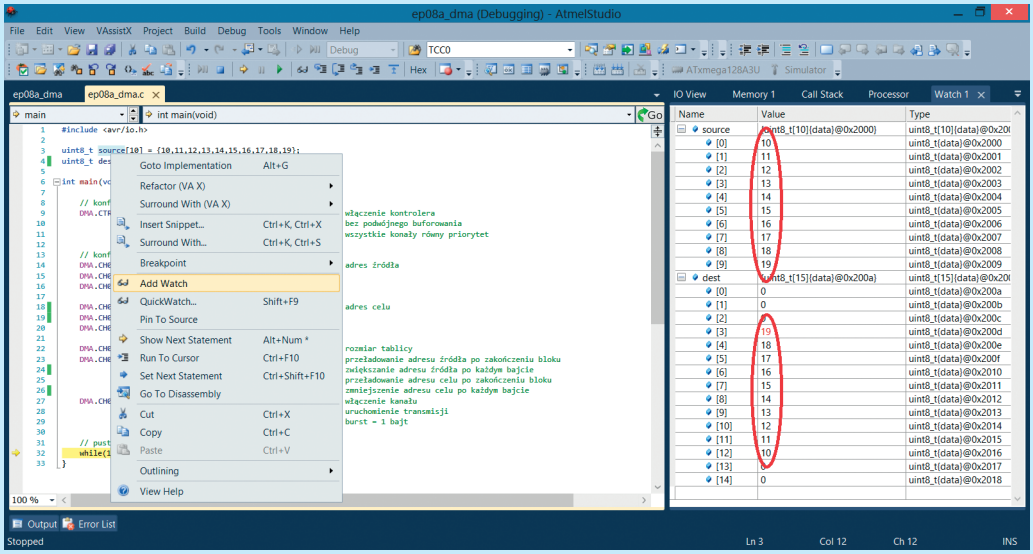
- **Processor** – stan najważniejszych rejestrów procesora, wskaźnik stosu, licznik programu, rejestr statusowy i rejestry robocze R0-R31,
- **IO View** – pozwala podglądać i modyfikować rejestry wszystkich układów peryferyjnych,
- **Call stack** – podgląd stosu i wywołania poszczególnych funkcji, wywołujących kolejne funkcje,
- **Memory** – podgląd pamięci RAM, Flash, EEPROM,
- **Watch** – podgląd wybranych zmiennych.

Oprócz tego, dostępnych jest jeszcze całe mnóstwo narzędzi ułatwiających debugowanie i monitorowanie pracy procesora – nie będę ich tu opisywał, ponieważ jest to temat na osobny odcinek (albo i dwa).

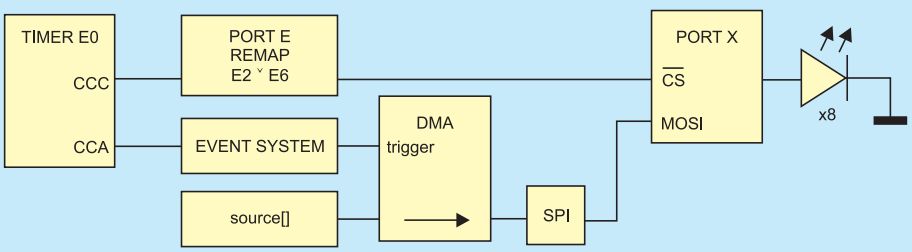
Aby widzieć zawartość tablic source[] oraz dest[], musimy kliknąć je prawym przyciskiem myszy, a następnie wybrać opcję Add to watch. Po prawej stronie pokażą nam się tabelę source[], wypełnioną liczbami od 10 do 19 oraz dest[], która jest wypełniona 0.

Wciskaj klawisz F11, aby przejść przez kolejne linie programu, aż do pustej pętli głównej. Możesz wtedy poćwiczyć korzystanie z IO View – obserwuj jak ustawiają się poszczególne bity w rejestrach kontrolera DMA.

Kiedy dojdziesz do pętli głównej, wróć do Watch i obserwuj tablicę dest[] wciskając klawisz F11. Kontroler DMA



Rysunek 5. Wynik działania pierwszego programu demonstrującego pracę DMA



Rysunek 6. Schemat połączeń między układami peryferyjnymi

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 76305, pass: 8741rnfv

zaczyna działać i rozpoczyna kopiowanie od elementu zerowego tablicy źródłowej, który trafia do elementu dwunastego. W ten sposób zapełnia się tabela aż do elementu trzeciego, kiedy to kopiowanie zostaje zakończone. Wynik programu przedstawiono na **rysunku 5**.

Przesyłanie tablic do peryferiów

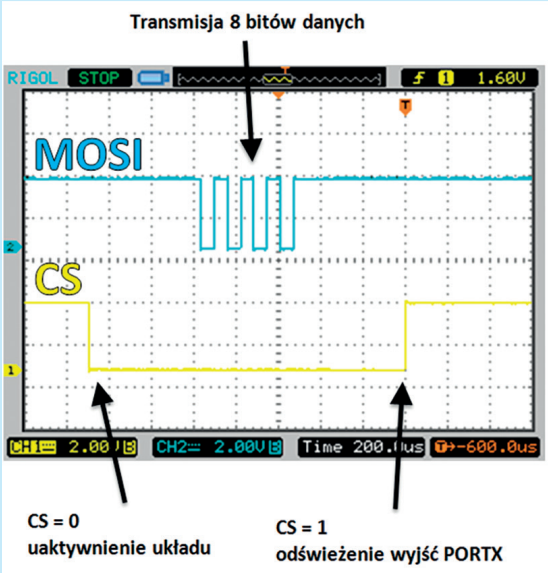
Poznaliśmy już elementarne podstawy działania DMA – czas najwyższy przejść do praktycznych zastosowań tego fantastycznego układu. Tym razem DMA będzie pobierać dane z tablicy i przesyłać je do diod LED portu X na płytce eXtrino XL, tworząc prostą animację. Rozwiązanie to będzie zrealizowane całkowicie sprzętowo – wystarczy raz skonfigurować poszczególne peryferia, puścić je w ruch, a potem wszystko będzie działo się automatycznie.

Schemat blokowy połączenia peryferiów pokazano na **rysunku 6**, natomiast kod programu opisywanego w tym ćwiczeniu zawiera **listing 2**. Dyrygentem naszej orkiestry będzie timer E0, wyznaczający cykl o okresie około 100 ms. Podczas tego cyklu procesor musi wykonać trzy ważne zadania, związanie z tym, że PORTX w płyt-

Listing 2. Kod programu do drugiego ćwiczenia

```
#include <avr/io.h>
#include „extrino_portx.h”

uint8_t source[] = {0b00000001,
                    0b00000011,
                    0b00000111,
                    0b00001111,
                    0b00011111,
                    0b00111111,
                    0b01111111,
                    0b11111111,
                    0b11111110,
                    0b11111100,
                    0b11111000,
                    0b11110000,
                    0b11100000,
                    0b11000000,
                    0b10000000,
                    0b01000000,
                    0b00100000,
                    0b00010000,
                    0b00001000,
                    0b00000100,
                    0b00000010,
                    0b00000001,
                    0b01010101,
                    0b10101010,
                    0b11111111,
                    0b00000000
                    };

int main(void) {

    // inicjalizacja PORTX (uwaga - przerwania wyłączone w pliku extrino_portx.h)
    PortxInit();

    // konfiguracja timera by zgłaszał zdarzenie co 1 sek i sterował pinem CS portu X
    TCE0.CTRLB          =    TC_WGMODE_SINGLESLOPE_gc|       // tryb normalny
                             TC0_CCCEN_bm;                   // włączenie wyjścia kanału output compare C
    TCE0.CTRLA          =    TC_CLKSEL_DIV1024_gc;           // ustawienie preskalera i uruchomienie
    TCE0.CCC            =    198;                            // wartość wyzwalająca kanał C
    TCE0.CCA            =    199;                            // wartość wyzwalająca kanał A
    TCE0.PER            =    200;                            // okres timera

    // konfiguracja CS PORTX
    PORTE.REMAP         =    PORT_TC0C_bm;                   // przeniesienie wyjścia kanału C TC0 z E2 na E6
    PORTE.DIRSET        =    PIN6_bm;                        // pin E6 jako wyjście

    // konfiguracja systemu zdarzeń
    EVSYS.CH0MUX        =    EVSYS_CHMUX_TCE0_CCA_gc;        // zdarzenie na CH0 wywołuje kanał A TE0

    // konfiguracja kontrolera DMA
    DMA.CTRL            =    DMA_ENABLE_bm|                  // włączenie kontrolera
                             DMA_DBUFMODE_DISABLED_gc|       // bez podwójnego buforowania
                             DMA_PRIMODE_RR0123_gc;          // wszystkie kanały równy priorytet

    // konfiguracja kanału DMA
    DMA.CH0.SRCADDR0    =    (uint16_t)source & 0xFF;       // adres źródła
    DMA.CH0.SRCADDR1    =    (uint16_t)source >> 8;
    DMA.CH0.SRCADDR2    =    0;

    DMA.CH0.DESTADDR0   =    (uint16_t)&SPIC.DATA & 0xFF;  // adres celu
    DMA.CH0.DESTADDR1   =    (uint16_t)&SPIC.DATA >> 8;
    DMA.CH0.DESTADDR2   =    0;

    DMA.CH0.TRFCNT      =    sizeof(source);                 // rozmiar bloku = rozmiar tablicy source
    DMA.CH0.REPCNT      =    0;                              // ile bloków, 0 oznacza wysyłanie
w nieskończoność
    DMA.CH0.TRIGSRC     =    DMA_CH_TRIGSRC_EVSYS_CH0_gc;    // kanał CH0 powoduje transfer
    DMA.CH0.ADDRCTRL    =    DMA_CH_SRCRELOAD_BLOCK_gc|      // przeładowanie adresu źródła po zakończeniu
bloku
                             DMA_CH_SRCDIR_INC_gc|           // zwiększanie adresu źródła po każdym bajcie
                             DMA_CH_DESTRELOAD_NONE_gc|      // przeładowanie adresu celu nigdy
                             DMA_CH_DESTDIR_FIXED_gc;        // stały adres docelowy
    DMA.CH0.CTRLA       =    DMA_CH_ENABLE_bm|               // włączenie kanału
                             DMA_CH_BURSTLEN_1BYTE_gc|       // burst = 1 bajt
                             DMA_CH_SINGLE_bm|               // pojedynczy burst po każdym zdarzeniu
                             DMA_CH_REPEAT_bm;               // powtarzanie

    // pusta pętla główna
    while(1) {}
}
```

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 76305, pass: 8741rnfv



Rysunek 7. Oscylogram przedstawiający przebiegi na liniach MOSI oraz CS

kach eXtrino XL jest sterowany przez SPI (co zostało omówione w EP 2014/07):

1. Ustawienie pinu CS portu X w stan niski, co uaktywnia układ slave SPI
2. Rozpoczęcie przesyłania bajtu danych,
3. Ustawienie pinu CS portu X w stan wysoki, co spowoduje przedstawienie przesłanego bajtu na diodach LED.

Żeby bardziej obrazowo przedstawić sposób, w jaki zachowują się sygnały na magistrali SPI, przedstawiam zdjęcie z oscyloskopu na **rysunku 7**.

Aby cykl pracy wynosił 100 ms, musimy ustalić preskaler timera na 1024 oraz ustawić rejestr PER na 200. Timery zostały dokładniej omówione w EP 2014/02.

Zacznijmy od omówienia sterowania pinem CS portu X, które najlepiej jest zrobić przy pomocy funkcji Capture/Compare (potocznie zwanej PWM, choć nie każde zastosowanie CC to jest PWM). Fizycznie CS na płytce eXtrino XL poprowadzony jest do pinu E6 procesora, ale timer E0 ma dostęp do pinów E0-E3, a timer E1 może sterować pinami E4-E5. Co tu robić? Na szczęście projektanci XMEGA przewidzieli taki przypadek i umożliwili remapowanie pinów w portach. Możemy wybrane wyjścia timera przenieść z pinów E0-E3 na E4-E7. Wystarczy zatem przenieść tylko E6, który jest powiązany z kanałem C funkcji Captue/Compare (stąd CCC). Remapowanie kanału C timera można włączyć wpisując PORT_TC0C_bm do rejestru PORTE.REMAP.

Nadanie rejestrowi TCE0.CCC wartości 198, kiedy PER ma wartość 200, umożliwi uzyskanie krótkotrwałego stanu niskiego, potrzebnego podczas przesyłania danych przez SPI. Przez większość czasu na pinie E6 będzie poziom wysoki, a w chwili kiedy licznik timera zrówna się z wartością 198 (=CCC), pin E6 zostanie wyzerowany, co będzie trwało aż do osiągnięcia przez timer wartości 200 (=PER).

W programie wykorzystujemy bibliotekę extrino_portx.h (dostępnej na płycie dołączonej do niniejszego numeru EP). Zawiera ona definicję PORTX_AUTOREFRESH, która może przyjmować wartość 1 lub 0, w zależności od tego czy chcemy, bo PORTX odświeżał się automatycznie z wykorzystaniem przerwań czy nie. Ponieważ odświeżaniem zajmuje się timer, musimy do wspomnianej definicji wpisać 0.

Jako drugie wyjście Capture/Compare timera wykorzystamy CCA (tym razem jest to zupełnie dowolne, można wybrać inny kanał). Wyjście to poprowadzimy do DMA przez kanał 0 systemu zdarzeń. Wyzwalaczem DMA mogą być kanały 0, 1 i 2. Po otrzymaniu sygnału wyzwalającego, DMA automatycznie skopiuje kolejną komórkę z tablicy source[] i przeniesie ją do interfejsu SPI, który natychmiast zacznie transmisję do portu X.

Układ DMA może być wyzwalany różnymi sygnałami i co ciekawe, ma on coś w rodzaju własnego systemu zdarzeń. Można więc bezpośrednio połączyć DMA do różnych peryferiów i uzyskać jeszcze większą prędkość kopiowania danych, ale trzeba uważać na pewną pułapkę. Dokładniej rzecz biorąc, transmisję DMA uaktywniać może flaga przerwania wybranego układu peryferyjnego, ale DMA nie zawsze może taką flagę wyzerować! Tak jest w przypadku timerów – gdybyśmy jako wyzwalacz wzięli flagę przerwania CCA, wówczas z wielkim skonsternowaniem byśmy stwierdzili, że po pierwszym wyzwoleniu DMA nie zatrzymuje się, lecz działa w nieskończoność. W takim przypadku powinniśmy odblokować przerwania, ponieważ flaga jest kasowana bezpośrednio po wejściu do procedury przerwania. Jednak, jeśli procesor miałby wchodzić do niej tylko po to, by zresetować flagę, to jest całkowicie bez sensu. Dlatego lepiej jest wykorzystać system zdarzeń, który rozwiązuje ten problem.

W kodzie programu na **listingu 2** powinniśmy zwrócić uwagę na różnice w ustawieniach względem pierwszego programu. Do rejestru DMA.CH0.REPCNT zostało wpisane zero. Oznacza to, że transakcja składa się z jednego bloku, który kopiowany będzie w nieskończoność. Do rejestru DMA.CH0.TRFCNT wpisujemy z ilu bajtów składa się blok i jest to oczywiście rozmiar tablicy źródłowej, pobrany operatorem sizeof(). Kolejna różnica jest w DMA.CH0.ADDRCTRL, gdzie wpisując DMA_CH_SRCRELOAD_BLOCK_gc ustaliliśmy, że adres tablicy źródłowej zostanie przywrócony do stanu początkowego, po zakończeniu przesyłania bloku. Ostatnie różnice dotyczą rejestru DMA.CH0.CTRLA, gdzie zniknęło polecenie uruchomienia transmisji, a pojawiły się dwa dodatkowe symbole:

- DMA_CH_SINGLE_bm – ten bit powoduje, że po wystąpieniu sygnału wyzwalającego, DMA wykona tylko jedną transmisję burst, po czym będzie oczekiwać na kolejny wyzwalacz.
- DMA_CH_REPEAT_bm – włącza powtarzanie transakcji tyle razy, ile jest wpisane do rejestru DMA.CH0.REPCNT. Zero jest wartością specjalną i oznacza kopiowanie w nieskończoną liczbę razy.

Na końcu programu mamy pustą pętle while(1). Kompilujemy program, ładujemy go do procesora i obserwujemy, jak kolejne elementy tablicy source[] pojawiają się na diodowym wyświetlaczu portu X.

Cały program to jedynie konfiguracja kilkunastu rejestrów, a potem wszystko dzieje się całkowicie sprzętowo. Czy XMEGA to wciąż zwykły mikrokontroler czy może specyficzne FPGA z różnymi peryferiami, które możemy sobie łączyć jak chcemy?

Dominik Leon Bieczyński
www.leon-instruments.pl

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 76305, pass: 8741rnfv