SPRAWOZDANIE

SYSTEMY WBUDOWANE

Obsługa układów peryferyjnych. Timery.

System przerwań układu SAM7.

IMIĘ I NAZWISKO: Jacek Wójcik

NUMER ĆWICZENIA: 10

Grupa laboratoryjna: 10

Data wykonania ćwiczenia: 22.12.2021

Spis treści

2. Zadanie 1 - kod

2.1 PIO_library.h

```
#include <targets\AT91SAM7.h> // Dołączam bibliotekę dla zestawu w laboratorium

/*  Funkcja do realizowania opóźnień
*   ms - liczba milisekund, jakie trzeba odczekać
*   Nic nie zwraca
*/
void time_delay(unsigned int ms);

/*  Funkcja do włączania i wyłączania zegara dla peryferiów
*   pio_pcer - włączenie (1) lub wyłączenie (0) zegara
*   a_b - kontroler peryferiów: PIOA (0) lub PIOB(1)
*   Nic nie zwraca
*/
void PIO_clock_enable(unsigned int pio_pcer, unsigned int a_b);

/*  Funkcja do włączania i wyłączania obsługi pinów w kontrolerze PIOB
*   pin_number - numer pinu do włączenia w kontrolerze PIOB
*   enable - włączenie (1) lub wyłączenie (0) danego pinu
*   Nic nie zwraca
*/
void PIOB_enable(unsigned int pin_number, unsigned int enable);

/*  Funkcja do ustawiania pinów w kontrolerze PIOB jako wejście lub wyjście
*   pin_number - numer pinu do skonfigurowania w kontrolerze PIOB
*   enable - ustawianie pinu jako wyjście (1) lub wejście (0)
*   Nic nie zwraca
*/
void PIOB_output_enable(unsigned int pin_number, unsigned int enable);

/*  Funkcja do ustawiania stanu pinów w kontrolerze PIOB
*   pin_number - numer pinu do skonfigurowania w kontrolerze PIOB
*   enable - ustawianie pinu w stan wysoki (1) lub niski (1)
*   Nic nie zwraca
*/
void PIOB_output_state(unsigned int pin_number, unsigned int enable);

/*  Funkcja do negowania stanu pinów w kontrolerze PIOB
*   pin_number - numer pinu do zanegowania
*   Nic nie zwraca
*/
void PIOB_output_negate(unsigned int pin_number);

/*  Funkcja do pobierania stanu pinu z kontrolera B
*   SW_number - numer pinu, którego stan będę pobierał
*   Zwraca liczbę oznaczającą stan wysoki (1) lub stan niski (0)
*/
unsigned int SW_odczyt(unsigned int SW_number);
```

```
/*  Funkcja do debouncingu przycisków
*   SW_number - numer pinu, którego stan będę sprawdzał
*   Nic nie zwraca
*/
void SW_czytaj(unsigned int SW_number);
```

2.2 PIO_library.c

```
#include "PIO_library.h"  // Dołączam moją bibliotekę

/*  Funkcja do realizowania opóźnień
*   ms - liczba milisekund, jakie trzeba odczekać
*   Nic nie zwraca
*/
void time_delay(unsigned int ms)
{
	// Pierwsza pętla odliczająca milisekundy
	for( volatile int aa=0;aa<=ms;aa++)
	{
	// Druga pętla odliczająca jedną milisekundę
	for( volatile int bb=0;bb<=3000;bb++)
	{
		__asm__("NOP");
	}
	}
}

/*  Funkcja do włączania i wyłączania zegara dla peryferiów
*   pio_pcer - włączenie (1) lub wyłączenie (0) zegara
*   a_b - kontroler peryferiów: PIOA (0) lub PIOB(1)
*   Nic nie zwraca
*/
void PIO_clock_enable(unsigned int pio_pcer, unsigned int a_b)
{
	if(pio_pcer==1)
	{
	// Włączam kontroler ustawiając bit nr. 2 dla PIOA lub 3 dla PIOB
	PMC_PCER = 1 << a_b + 2;
	}

	if(pio_pcer==0)
	{
	// Wyłączam kontroler ustawiając bit nr. 2 dla PIOA lub 3 dla PIOB
	PMC_PCDR = 1 << a_b + 2;
	}
}

/*  Funkcja do włączania i wyłączania obsługi pinów w kontrolerze PIOB
*   pin_number - numer pinu do włączenia w kontrolerze PIOB
*   enable - włączenie (1) lub wyłączenie (0) danego pinu
*   Nic nie zwraca
*/
void PIOB_enable(unsigned int pin_number, unsigned int enable)
{
	if(enable == 1)
	{
	// Włączam obsługę pinu ustawiając bit odpowiadający danemu pinowi
	PIOB_PER = 1 << pin_number;
```

```
	}

	if(enable == 0)
	{
	// Wyłączam obsługę pinu ustawiając bit odpowiadający danemu pinowi
	PIOB_PDR = 1 << pin_number;
	}
}

/*  Funkcja do ustawiania pinów w kontrolerze PIOB jako wejście lub wyjście
*   pin_number - numer pinu do skonfigurowania w kontrolerze PIOB
*   enable - ustawianie pinu jako wyjście (1) lub wejście (0)
*   Nic nie zwraca
*/
void PIOB_output_enable(unsigned int pin_number, unsigned int enable)
{
	if(enable == 0)
	{
	// Ustawiam pin jako wejście
	PIOB_ODR = 1 << pin_number;
	}

	if(enable == 1)
	{
	// Ustawiam pin jako wyjście
	PIOB_OER = 1 << pin_number;
	}
}

/*  Funkcja do ustawiania stanu pinów w kontrolerze PIOB
*   pin_number - numer pinu do skonfigurowania w kontrolerze PIOB
*   enable - ustawianie pinu w stan wysoki (1) lub niski (1)
*   Nic nie zwraca
*/
void PIOB_output_state(unsigned int pin_number, unsigned int enable)
{
	if(enable == 0)
	{
	// Ustawiam pin w stan niski
	PIOB_CODR = 1 << pin_number;
	}

	if(enable == 1)
	{
	// Ustawiam pin w stan wysoki
	PIOB_SODR = 1 << pin_number;
	}
}

/*  Funkcja do negowania stanu pinów w kontrolerze PIOB
*   pin_number - numer pinu do zanegowania
*   Nic nie zwraca
```

```
*/
void PIOB_output_negate(unsigned int pin_number)
{
	// Oznaczam piny, których stan będę zmieniał
	PIOB_OWER = 1 << pin_number;
	// Neguję stan pinu
	PIOB_ODSR ^= 1 << pin_number;
}

/*  Funkcja do pobierania stanu pinu z kontrolera B
*   SW_number - numer pinu, którego stan będę pobierał
*   Zwraca liczbę oznaczającą stan wysoki (1) lub stan niski (0)
*/
unsigned int SW_odczyt(unsigned int SW_number)
{
	// Zwracam wartość bitu, który odpowiada danemu pinowi
	return PIOB_PDSR & 1 << SW_number;
}

/*  Funkcja do debouncingu przycisków
*   SW_number - numer pinu, którego stan będę sprawdzał
*   Nic nie zwraca
*/
void SW_czytaj(unsigned int SW_number)
{
	time_delay(75); // Czekam, aż stan przycisku się ustabilizuje
	// Czekam tak długo, jak przycisk będzie wciśnięty
	while((PIOB_PDSR & 1 << SW_number) == 0);
}
```

2.3 main.c

```
#include "PIO_library.h" // Dołączam moją bibliotekę
#include <ctl_api.h>     // Dołączam bibliotekę konieczną do przerwań

#define LCD_BACKLIGHT 20 // Podstawiam 20 za LCD_BACKLIGHT - dla czytelności
#define PIOB 1           // Podstawiam 1 za PIOB - dla czytelności
#define ENABLE 1         // Podstawiam 1 za ENABLE - dla czytelności
#define DISABLE 0        // Podstawiam 0 za DISABLE - dla czytelności
#define TIMER_COUNTER_0 12  // Podstawiam 12 za TIMER_COUNTER_0 - dla czytelności
#define INT_PRIORITY 1   // Podstawiam 1 za INT_PRIORITY- dla czytelności

// Procedura obsługi przerwania
void timer_int() {
        // Kasuję flagę przerwania poprzez pusty odczyt rejestru statusu
        TC0_SR;
        // Funkcją z mojej biblioteki zmieniam stan podświetlenia ekranu
        PIOB_output_negate(LCD_BACKLIGHT);
}

// Główna funkcja
int main() {
        PIO_clock_enable(ENABLE,PIOB);           // Włączam zegar peryferiów dla kontrolera PIOB
        PIOB_enable(LCD_BACKLIGHT, ENABLE);      // Włączam pin LCD_BACKLIGHT
        PIOB_output_enable(LCD_BACKLIGHT, ENABLE);  // Ustawiam pin LCD_BACKLIGHT jako wyjście

        PMC_PCER |= PMC_PCER_TC0;        // Włączam zegar peryferiów dla timera 0

        TC0_CCR = TC0_CCR_CLKDIS;        // Wyłączam Timer 0
        TC0_IDR = 0xFF;                  // Wyłączam wszystkie źródła przerwania dla Timera 0
        // Wykonuję pusty odczyt rejestru stanu Timera 0 w celu usunięcia flagi przerwania
        TC0_SR;
        // Wybieram preskaler 1/1024 (1 << 2),
        // ustawiam reset licznika po osiągnięciu wartości rejestru RC (1 << 14)
        TC0_CMR = 1<<2 | 1<<14;
        TC0_RC = 9375;                   // Ustawiam wartość RC na czas 200ms
        // Ustawiam źródło przerwania jako porównanie wartości licznika z rejestrem RC
        TC0_IER |= 1<<4;
        TC0_CCR = TC0_CCR_CLKEN | TC0_CCR_SWTRG; // Włączam zegar i resetuję licznik

        ctl_global_interrupts_disable();  // Globalnie wyłączam przerwania
        // Konfiguruję przerwanie od timera, czyli ustawiam wektor przerwania jako Timer 0,
        // z priorytetem 1, z stałym wyzwalaczem przerwania, z metodą do obsługi przerwania jako
        // "timer_int", i bez zwracania wskaźnika do funkcji, która poprzednio obsługiwała to
        // przerwanie (przesyłam 0).
        ctl_set_isr(TIMER_COUNTER_0,INT_PRIORITY,CTL_ISR_TRIGGER_FIXED,timer_int,0);
        ctl_unmask_isr(TIMER_COUNTER_0);   // Włączam programowe przerwania od timera
        ctl_global_interrupts_enable();   // Globalnie włączam przerwania

        while(1); //Nieskończona pętla główna
}
```

3. Zadanie 1 - opis

To zadanie polegało na zaprogramowaniu przerwania od Timera 0 tak, żeby co 200 milisekund podświetlenie ekranu LCD zmieniało swój stan (włączało się lub wyłączało).
Do wykonania tego zadania potrzebowałem dwóch bibliotek:

- PIO_library - moja własna biblioteka do obsługi portów mikrokontrolera, dołączająca również bibliotekę “<targets\AT91SAM7.h>” z podstawowymi makrami mikrokontrolera pozwalającymi na obsługę timerów. Opis tej biblioteki zawarłem w pliku nagłówkowym oraz w poprzednim laboratorium.
- ctl_api - biblioteka potrzebna do obsługi przerwań.

Na początek wykonałem standardową inicjalizację kontrolera PIOB i pinu 20, służącego do włączania i wyłączania podświetlenia. Następnie przeszedłem do inicjalizacji Timera 0, na co składały się następujące kroki:

1. Wyłączenie Timera 0 przy użyciu rejestru TC0_CCR
2. Wyłączenie wszystkich źródeł przerwania dla timera poprzez rejestr TC0_IDR
3. Wyczyszczenie rejestru stanu timera poprzez odczytanie rejestru TC0_SR
4. Ustawienie za pomocą rejestru TC0_CMR preskalera na 1/1024 i resetu licznika na moment, gdy wartość licznika osiągnie wartość z rejestru TC0_RC.
5. Ustawienie wartości rejestru TC0_RC na 93075, co przy taktowaniu 48MHz i preskalerze 1/1024 odpowiada czasowi 200ms. Sposób obliczeń prezentowałem w poprzednich sprawozdaniach.
6. Ustawiam źródło przerwania jako porównanie wartości licznika z rejestrem TC0_RC
7. Włączam zegar i jednocześnie resetuję licznik za pomocą rejestru TC0_CCR

Gdy udało mi się już zainicjalizować Timer 0, zostało mi wykonać obsługę przerwania. W tym celu napisałem procedurę “**timer_int()**”, w której kasuję flagę przerwania poprzez odczyt rejestru stanu TC0_SR i zmieniam stan podświetlenia ekranu funkcją z mojej biblioteki.

Żeby podłączyć moją procedurę do przerwania od Timera 0 muszę najpierw globalnie wyłączyć obsługę przerwań, żeby nie doszło do przerwania przed zakończeniem konfiguracji, następnie metodą “**ctl_set_isr**” łączę wektor przerwania “**TIMER_COUNTER_0**” z moją procedurą “**timer_int()**”. W tej samej funkcji nadaję temu przerwaniu priorytet 1, oznaczam wyzwalacz tego przerwania jako stały, co oznacza brak możliwości określenia momentu wyzwolenia przerwania, i nie chcę otrzymać wskaźnika do poprzedniej procedury, jaka była podłączona do tego przerwania. Po podłączeniu procedury wystarczy włączyć przerwanie dla Timera 0 i włączyć globalną obsługę przerwań.

4. Zadanie 2 - kod

```
#include "PIO_library.h" // Dołączam moją bibliotekę
#include <ctl_api.h>     // Dołączam bibliotekę konieczną do przerwań

#define LCD_BACKLIGHT 20 // Podstawiam 20 za LCD_BACKLIGHT - dla czytelności
#define BUTTON_SW1 24    // Podstawiam 24 za BUTTON_SW1 - dla czytelności
#define BUTTON_SW2 25    // Podstawiam 25 za BUTTON_SW2 - dla czytelności
#define PIOB 1           // Podstawiam 1 za PIOB - dla czytelności
#define ENABLE 1         // Podstawiam 1 za ENABLE - dla czytelności
#define DISABLE 0        // Podstawiam 0 za DISABLE - dla czytelności
#define PIO_CONTROLLER_B 3 // Podstawiam 3 za PIO_CONTROLLER_B - dla czytelności
#define INT_PRIORITY 1   // Podstawiam 1 za INT_PRIORITY - dla czytelności

// Procedura obsługi przerwania
void button_int() {
  // Zapamiętuję stan rejestru statusu przerwań w kontrolerze PIOB
  int interrupt_status = PIOB_ISR;
  if(interrupt_status & (1 << BUTTON_SW1)) { // Jeżeli wcisnąłem przycisk SW1
        PIOB_output_state(LCD_BACKLIGHT, ENABLE);   // Włącz podświetlenie
  }
  else if(interrupt_status & (1 << BUTTON_SW2)) {  // Jeżeli wcisnąłem przycisk SW2
        PIOB_output_state(LCD_BACKLIGHT, DISABLE);  // Wyłącz podświetlenie
  }
}
// Główna funkcja
int main(void) {
  PIO_clock_enable(ENABLE,PIOB);                 // Włączam zegar peryferiów dla kontrolera PIOB
  PIOB_enable(LCD_BACKLIGHT, ENABLE);            // Włączam pin LCD_BACKLIGHT
  PIOB_output_enable(LCD_BACKLIGHT, ENABLE);  // Ustawiam pin LCD_BACKLIGHT jako wyjście
  PIOB_IER = 1<<BUTTON_SW1 | 1<<BUTTON_SW2;   // Włączam przerwania dla przycisków SW1 i SW2

  ctl_global_interrupts_disable();  // Globalnie wyłączam przerwania
  // Konfiguruję przerwanie od kontrolera PIOB, czyli ustawiam wektor przerwania jako PIOB,
  // z priorytetem 1, z stałym wektorem przerwania, z metodą do obsługi przerwania jako
  // "button_int", i bez zwracania wskaźnika do funkcji, która poprzednio obsługiwała to
  // przerwanie (przesyłam 0).
  ctl_set_isr(PIO_CONTROLLER_B,INT_PRIORITY,CTL_ISR_TRIGGER_FIXED,button_int,0);
  ctl_unmask_isr(PIO_CONTROLLER_B); // Włączam programowe przerwania od timera
  ctl_global_interrupts_enable();   // Globalnie włączam przerwania

  while(1); //Nieskończona pętla główna
}
```

5. Zadanie 2 - opis

To zadanie polegało na zaprogramowaniu obsługi przerwań w kontrolerze PIOB w taki sposób, aby naciśnięcie przycisku SW_1 włączało podświetlenie ekranu LCD, a naciśnięcie przycisku SW_2 wyłączało podświetlenie ekranu LCD.

Do tego zadania wykorzystałem identyczne biblioteki jak do zadania 1. Również tak samo jak w zadaniu 1 na początek zainicjalizowałem kontroler PIOB i pin 20, służący do kontroli podświetlenia. Następnie włączyłem generowanie przerwania przez kontroler PIOB, gdy ten wychwyci zmianę na linii wejścia/wyjścia.

Następnym krokiem było napisanie procedury obsługującej przerwanie o nazwie “**button_int()**”. W tej procedurze, najpierw pobieram wartość rejestru statusu przerwań PIOB_ISR do zmiennej “**interrupt_status**”. Musiałem zapamiętać wartość tego rejestru w innej zmiennej, ponieważ odczytanie jego wartości usuwa jego treść. Następnie za pomocą instrukcji if() sprawdziłem który przycisk został naciśnięty i włączyłem lub wyłączyłem podświetlenie ekranu.

Ostatnim krokiem było podłączenie procedury do obsługi przerwania. Wykorzystałem do tego kod podłączenia procedury z zadania 1, zamieniając jedynie wektor przerwania na kontroler PIOB i procedurę obsługi przerwania na “**button_int()**”. Priorytet, typ wyzwalacza i żądanie zwrócenia wskaźnika pozostawiłem tak samo jak w zadaniu 1. Oczywiście również wyłączyłem obsługę przerwań przed podłączeniem, a po podłączeniu odmaskowałem odpowiedni wektor przerwania i ponownie włączyłem obsługę przerwań.

Dodatkowo, do tego zadania w pliku z poleceniem zostało zadane pytanie:

Jak rozpoznać która linia PIOB spowodowała procedurę obsługi przerwania od PIOB?

Żeby odpowiedzieć na to pytanie, należy zgodnie ze wskazówką udzieloną pod tym pytaniem udać się na stronę 224 dokumentacji mikrokontrolera. Znajduje się tam rozdział “Input Change Interrupt”, który opisuje w jaki sposób można zaprogramować kontroler PIOB tak, żeby generował przerwanie dla każdej zmiany stanu na linii kontrolera.

W tym celu należy najpierw włączyć zegar kontrolera PIOB, ponieważ wykrywanie zmiany stanu odbywa się poprzez porównanie dwóch kolejnych próbek stanu wejścia linii. Następnie należy odpowiednie piny odmaskować w rejestrze PIOB_IMR (*Interrupt Mask Register*, rejestr maski przerwań) za pomocą rejestru PIOB_IER (*Interrupt Enable Register*, rejestr włączania przerwań).

Gdy te rzeczy są już zrobione, należy podłączyć wektor przerwań dla kontrolera PIOB z własną procedurą. Wartość pinu, który wywołał przerwanie, można odczytać za pomocą rejestru PIOB_ISR (*Interrupt Status Register*, rejestr statusu przerwania).

Ważne jest, żeby pamiętać, że za każdym razem, gdy odczytujemy wartość rejestru PIOB_ISR, jego wartość jest automatycznie usuwana, a zatem należy na początku procedury obsługi przerwania zapisać jego wartość do zmiennej i więcej nie odczytywać jego wartości, ponieważ będzie to wartość 0 (pusty rejestr).

8. Wnioski

Z tych laboratoriów wyciągnąłem następujące wnioski:

1) Dzięki bibliotece “**ctl_api.h**” jestem w stanie sterować obsługą przerwań w dużo łatwiejszy sposób niż na AVR:
    a) Funkcje “**ctl_global_interrupts_disable()**” i “**ctl_global_interrupts_enable()**” pozwalają mi w czytelny sposób sterować tym, czy globalne przerwania są włączone.
    b) Funkcje “**ctl_unmask_isr()**” i “**ctl_mask_isr()**” pozwala mi w łatwy sposób włączać i wyłączać wektory przerwań podczas działania programu.
    c) Funkcja “**ctl_set_isr()**” pozwala mi na przypisanie do obsługi przerwania procedury o dowolnej nazwie, co zwiększa czytelność kodu.
    d) Dodatkowo mogę określić priorytet przerwania, co pozwala mi np. określić, że przerwanie o większym priorytecie może zostać wykonane podczas obsługiwania przerwania o niższym priorytecie.
2) Odczyt rejestru statusowego przerwań dla danego wektora w przerwaniu automatycznie czyści wartość tego rejestru, co powoduje konieczność zapisywania wartości tego rejestru w osobnej zmiennej.
3) Jednak z drugiej strony podczas procedury obsługi przerwania należy zawsze czyścić wartość rejestru statusu przerwania dla danego wektora, ponieważ w innym przypadku może się okazać że program wpadnie w nieskończoną pętlę przerwań, ponieważ to samo przerwanie będzie wykonywało się cały czas.
4) Inną różnicą pomiędzy AVR a ARM jest sposób działania przerwań na pinach.

    W AVR dla każdego pinu pisze się osobną procedurę obsługi przerwania (a raczej implementuje gotową funkcję z biblioteki, taką jak np. ISR (INT0_vect) dla przerwania INT0. Z jednej strony od razu uruchamia się obsługa przerwania dla danego pinu, ale z drugiej na Atmedze32 mamy tylko 3 takie piny, co jest bardzo małą ilością.

    W ARM każdy kontroler PIO ma jeden wektor przerwania dla całej linii, co z jednej strony pozwala nam na użycie wszystkich 32 pinów, jednak mając informacje z rejestru PIO_ISR musimy samodzielnie znaleźć pin, który wywołał dane przerwanie.
5) Ciekawą rzeczą jest fakt, że nie da się ustawić preskalera na wartość MCK/1 - najmniejszy możliwy to MCK/2, co oznacza że przy częstotliwości 48MHz możemy za pomocą timera zliczać z dokładnością do 0,04 mikrosekundy.