NIOS II na maXimatorze, czyli mikroprocesor w układzie FPGA (4)

Nowości i powrót do świata timerów i przerwań

Docierają do nas kolejne nowinki od Intel FPGA – została wydana aktualizacja środowiska Quartus do wersji 17.1, w której będą prowadzone kolejne części niniejszego kursu. Ponadto kontynuować będziemy zgłębianie w praktyczny sposób tematyki timerów oraz innych przerwań w naszym systemie.

Firma Intel FPGA przygotowała w ostatnich tygodniach aktualizację środowiska programistycznego Quartus do wersji 17.1. Aby zbytnio nie zanudzać Was analizą zmian, powiem w „telegraficznym skrócie" o najważniejszych z nich:

1. Zmieniono nazwę narzędzia Qsys, na Platform Designer. Taka mała, kosmetyczna zmiana, ale mogąca wprowadzić nieco zamieszania, jeśli chcielibyśmy znaleźć interesującą nas pozycję.
2. Zaktualizowano środowisko programistyczne Eclipse dołączone do pakietu do nowszej wersji (Mars).
3. Naprawiono kilka błędów.

Myślę, że to też dobre miejsce na pewną uwagę – po aktualizacji nowe Eclipse będzie chciało utworzyć nowe środowisko pracy (workspace). Nic nie stoi jednak na przeszkodzie, aby wskazać to, którego używaliśmy poprzednio – program poprosi nas o zgodę na aktualizację „bazy danych", wraz z ostrzeżeniem, że może nie być możliwości powrotu do korzystania ze starszej wersji.

W zasadzie z większości funkcji korzystamy dokładnie tak jak w poprzedniej wersji – więc do dzieła!

Poprzednie części kursu i dodatkowe materiały dostępne są na stronie www.media.avt.pl

Listing 1. Odmierzanie czasu za pomocą przerwania Timera 0

```
void timer0Interrupt(void* context){
    //Zadanie 2
    static uint16_t msCounter = 0;
    static uint16_t secCounter = 0;
    if(msCounter < 999){
        msCounter++;
    }else{
        msCounter = 0;
        secCounter++;
        intDisplayDec(secCounter);
    }
    IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER0_BASE, 0);
    refreshDisplay();
}
```

Sprawdzamy zadanie domowe

Na początek, wzorem poprzednich spotkań, rzućmy okiem na zadania domowe. Pierwsze z nich wymagało napisania zaledwie 2 linijek w funkcji *main* zaraz przed uruchomieniem timera (pamiętajmy o tym, że zmiana zawartości rejestru *PERIOD* powoduje zatrzymanie timera):

```
IOWR_ALTERA_AVALON_TIMER_PERIODH(TIMER0_BASE,
(50000000UL-1UL)>>16);
IOWR_ALTERA_AVALON_TIMER_PERIODL(TIMER0_BASE,
(50000000UL-1UL));
```

Zapisujemy za ich pomocą rejestr odpowiadający za okres, z którym jest generowane przerwanie. Od pożądanej wartości odejmujemy 1, ponieważ timer zlicza od 0, zaś dopisek UL powoduje, że kompilator będzie traktował liczbę jako typ *unsigned long*, dzięki czemu na pewno nie nastąpi jej „przycięcie" do mniejszej liczby bitów. Ponieważ zapis prowadzimy po 16 bitów (a timer nasz jest 32-bitowy) operacja podzielona jest na zapis starszej i młodszej części bitów, stąd też wynika przesunięcie bitowe przy zapisie rejestru *PERIODH*. Po dodaniu tych instrukcji domyślny (ustawiony w *Platform Designer* <dawny *Qsys*>) okres timera zostanie zastąpiony wartością 49 999 999, zatem przerwanie zostanie wywołane co 50 000 000 cykli zegara, czyli co... 1 sekundę. Idealna prędkość dla pokazania jak działa multipleksowanie!

Kolejne zadanie wymagało od nas odmierzania czasu. Myślę, kod z **listingu 1** nie wymaga żadnych dodatkowych komentarzy.

Ostatnie zadanie dotyczące wyświetlania liczb w różne sposoby także nie powinno stanowić większego problemu, a przykładowe rozwiązania znajdziecie wśród plików towarzyszących tej części kursu.

Przerwania z zewnątrz

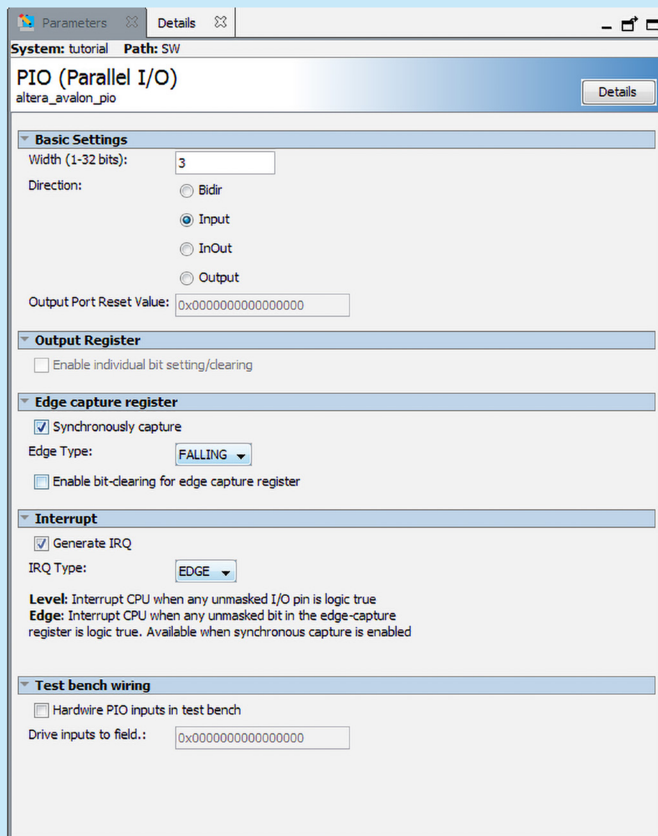
Ostatnio zajmowaliśmy się przerwaniami generowanymi przez timer, czyli układ całkowicie wewnątrz naszego procesora. Często jednak zachodzi potrzeba, aby wygenerować przerwanie w wypadku jakiegoś zdarzenia zewnętrznego, wymagającego pilnej obsługi, np. pochodzącego z zegara czasu rzeczywistego (takiego jak PCF8583 czy DS1307), czy w wypadku konieczności precyzyjnego pomiaru czasu wystąpienia jakiegoś zdarzenia (budując przykładowo licznik rowerowy chcielibyśmy, aby bardzo precyzyjnie mierzyć okres obrotu koła, podobnie taka funkcjonalność przydaje się np. w wypadku dekodowania sygnału z kodowaniem Manchester).

W czasie naszego spotkania dotyczącego linii GPIO wspominałem o tym, jak można ustawić różne tryby generowania przerwań przez porty wejściowe. Czas, abyśmy przypomnieli sobie gdzie należy poklikać, aby wszystko „migało prawidłowo".

Na początek trochę grzebania w sprzęcie

Otwieramy nasz (możecie wykorzystać swój, do czego gorąco zachęcam, ja jednak będę operował nazwami, które zastosowałem w swoim dziele sztuki ;)) projekt w Platform Designer, a następnie otwieramy właściwości (*Parameters*) komponentu PIO, który kiedyś zdefiniowaliśmy do obsługi przycisków (u mnie nosi on nazwę SW).

Na początek zaznaczamy opcje *Synchronously capture*, a następnie wybieramy *FALLING*. Dzięki temu dodana zostanie funkcja



Rysunek 1. Ustawienia komponentu PIO do generowania przerwań po wciśnięciu przycisków

wykrywania zboczy opadających. Potem zaznaczamy *Generate IRQ*, i wybieramy *EDGE*. Dzięki temu system przerwań będzie aktywowany po wykryciu zbocza (które wcześniej określiliśmy jako opadające).

Po tej operacji pojawi się dodatkowy port – *irq*. Musimy podłączyć go do odpowiadającego portu w naszym rdzeniu, dokładnie tak samo jak w wypadku timera. Dla porządku możemy wybrać *System → Assign Interrupt Numbers*, choć zaraz po podłączeniu portu systemu przerwań powinien automatycznie zostać nadany niekolidujący z niczym numer.

Na koniec zapisujemy nasz system, generujemy pliki HDL, i dokonujemy syntezy w Quartusie, po czym wgrywamy nową konfigurację do naszego układu FPGA. Nic prostszego, prawda? Zatem czas uruchomić środowisko programistyczne i skorzystać z właśnie utworzonego doskonałego (czy aby na pewno?) narzędzia do wykrywania wciśnięcia przycisków (**rysunek 1**).

A potem programujemy!

Bazując na naszym poprzednim programie (i przywracając go do stanu bez wykonanych zadań domowych związanych z odmierzaniem czasu oraz spowalnianiem multipleksowania), rozpocznijmy pracę od napisania funkcji, która będzie obsługiwała nasze przerwanie generowane poprzez wciśnięcie przycisku.

Omówienia wymaga tu chyba tylko linijka zapisująca do rejestru komponentu *PIO* wartość 0. Powoduje ona wykasowanie wszystkich bitów w rejestrze wykrywania zboczy (w rejestrze takim pojawia się

W naszym systemie cały port PIO wywołuje jedno przerwanie, nie ma osobnych przerwań dla każdego z pinów. Podobne rozwiązania można spotkać w procesorach AVR (przerwania typu PCINT), czy ARM, gdzie czasem istnieje tylko jedno przerwanie na dany układ peryferyjny i dopiero sprawdzenie odpowiednich rejestrów daje nam wiadomość, jakie było dokładne źródło zdarzenia.

Poprzednie części kursu i dodatkowe materiały dostępne są na na stronie www.media.avt.pl

1 na pozycji odpowiadającej pinowi, na którym wyryto zbocze). Jako, że niezerowa wartość tego rejestru powoduje przerwanie, wykasowanie go zapewni „potwierdzenie" obsłużenia przerwania (podobnie jak w wypadku timera, pamiętacie?). Co więcej, jeśli chcielibyśmy wiedzieć który pin wywołał to konkretne przerwanie, moglibyśmy odpowiednią instrukcją odczytać wartość tego rejestru.

Następnie dopisujemy w funkcji *main*, po inicjalizacji timera, następujące linijki kodu:

```
alt_ic_isr_register(SW_IRQ_INTERRUPT_CONTROLLER_ID,
SW_IRQ, SWInterrupt, NULL, NULL);
IOWR_ALTERA_AVALON_PIO_IRQ_MASK(SW_BASE, 0b111);
intDisplayDec(0);
```

Do pierwszej z nich komentarz jest chyba w 200% zbędny, bo o tym już się uczyliśmy. Druga linijka ustawia rejestr „maskujący", który pozwala na filtrowanie które piny powodować będą przerwanie – przecież nie zawsze chcemy aby zbocze opadające na jakimkolwiek pinie danego portu generowało przerwanie – czasem wystarczy tylko jeden pin. Tu ustawiamy wszystkie bity na 1, aby generować przerwania za pomocą dowolnego z przycisków – potem możecie poeksperymentować i sprawić aby tylko wybrane z nich generowały przerwanie.

Po skompilowaniu i wgraniu programu licznik będzie zliczał ilość przerwań, czyli... ilość kliknięć przyciskami. (Kody programów, na poszczególnych etapach naszego spotkania znajdują się w głównym folderze projektu i noszą nazwy main_XX_(...).c – jeśli potrzebujecie przywołać dany kod wystarczy zawartość wybranego pliku skopiować do pliku main.c w folderze *\Projekt\qsys\software\Tutorial04* i skompilować program).

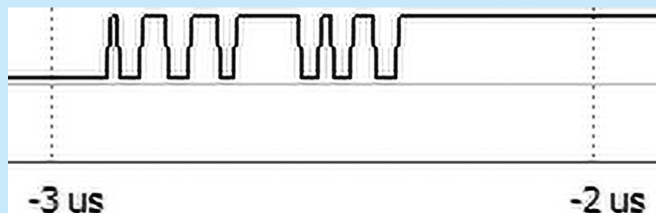
Czas na testy, które mam nadzieję, zakończą się połowicznym sukcesem. Dlaczego połowicznym? Gdyż prawdopodobnie niektóre przyciski, przynajmniej raz na jakiś czas będą generowały zliczenia w dziwny sposób – np. zliczone zostanie puszczenie przycisku (mimo iż to zbocze narastające!), lub jedno wciśnięcie przycisku zostanie zliczone jako kilka. Co zrobiliśmy źle?

W zasadzie nic – udało nam się jedynie zademonstrować zjawisko zwane potocznie „drganiem styków", którym zajmiemy się już wkrótce. Na czym ono polega? To nic innego jak fakt, że zbocze generowane przez przycisk jest finalnie widziane przez układ cyfrowy jako szereg przejść pomiędzy stanem wysokim a niskim. Jeśli układ taki działa dostatecznie szybko (a nasz tak działa) to w zboczu, które z punktu widzenia człowieka jest idealne, zauważy on wiele zboczy, zarówno narastających jak i opadających. Czas zająć się tym problemem i jednocześnie stwierdzić, że... wykorzystanie przerwań zewnętrznych do obsługi przycisków to nie najlepszy pomysł.

Drgania styków – jak sobie z nimi poradzić

Skoro ustaliliśmy już, że przerwanie generowane przez przycisk nie jest dobrym sposobem na jego obsługę, wypada zadać sobie pytanie – jak to robić?

Najlepiej wykorzystać do tego celu... przerwanie naszego timera! Jeśli w każdym przerwaniu sprawdzać będziemy stan przycisku (a więc u nas co 1 ms), a następnie założymy, że np. brak zmian stanu danego wejścia układu przez 50 ms (50 kolejnych przerwań) oznacza brak drgań styków, to wtedy możemy zliczyć dane kliknięcie. Jak to zaimplementować? Deklarację zmiennej *counter* przenosimy przed definicję funkcji przerwania, którą następnie modyfikujemy w sposób przedstawiony na **listingu 3**. Na początek może to wyglądać nieco skomplikowanie, ale powolutku przeanalizujmy ten kod.



Rysunek 2. Efekt „drgania styków" – układ cyfrowy wykrywa wiele przełączeń, zanim nastąpi ustabilizowanie się stanu

Listing 2. Funkcja generująca przerwanie po naciśnięciu przycisku

```
void timer0Interrupt(void* context){
    IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER0_BASE, 0);
    refreshDisplay();
}

void SWInterrupt(void* context){
    static uint16_t counter = 0;
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(SW_BASE, 0);
    counter++;
    intDisplayDec(counter);
}
```

Listing 3. Funkcja eliminująca drgania styków

```
volatile uint16_t counter = 0;

void timer0Interrupt(void* context){
    IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER0_BASE, 0);
    refreshDisplay();
    static uint16_t msCounter = 0;
    uint16_t state = IORD_ALTERA_AVALON_PIO_DATA(SW_BASE);
    if(state != 0b111){
        if(msCounter < 50){
            msCounter++;
        }else if(msCounter == 50){
            counter++;
            intDisplayDec(counter);
            msCounter++;
        }
    }else{
        msCounter = 0;
    }
}
```

Zaczynamy od odczytania stanu przycisków i sprawdzamy, czy którykolwiek z nich jest wciśnięty (czy którekolwiek wejście ma stan inny niż 1). Jeśli nie to resetujemy licznik milisekund (bo, albo nie jest wciśnięty żaden przycisk, albo odczytaliśmy stan wysoki z powodu „drgań styków"). W przeciwnym wypadku, jeśli jeszcze nie odmierzono 50 ms, licznik zwiększamy.

W momencie przekroczenia przez licznik wartości 49 (warunek < 50), przechodzimy do sprawdzenia, czy jego wartość wynosi dokładnie 50, jeśli tak, wykonujemy zadanie (u nas zwiększenie wartości licznika i wyświetlenie jej na wyświetlaczu) i zwiększamy kolejny raz licznik milisekund (do wartości 51). Ostatnie porównanie i zwiększenie licznika służy temu, aby zadanie wykonać tylko raz (dopiero zwolnienie przycisku spowoduje ustawienie licznika milisekund na zero).

Proste rozwiązanie a cieszy oko! Już nie ma zwiększania licznika przy puszczaniu przycisku, ani liczenia jednego kliknięcia za kilka. Sukces! I to z wykorzystaniem dalej jedynego timera w naszym systemie!

Rozwiązanie „na sterydach"

Jakie rozwiązanie byłoby „fajniejsze" od tego, które właśnie opracowaliśmy? Na przykład takie, które umożliwia wykrywanie przytrzymania przycisku, powtarzanie jakiejś akcji przy trzymaniu przycisku i proste definiowanie oraz zmienianie działań podejmowanych

Rysunek 3. Uproszczona ilustracja funkcji eliminującej drgania styków

Poprzednie części kursu i dodatkowe materiały dostępne są na stronie www.media.avt.pl

w każdym z tych wypadków. Wykresy pokazana na **rysunkach 4** i **5** obrazują różne przypadki działania takiego koncepcyjnego systemu. Poszedłem jeszcze nieco dalej i dodałem opcję przekazywania do funkcji parametrów (podobnie jak umożliwia to nam biblioteka do obsługi przerwań w ekosystemie NIOS II). W kodzie przemyciłem także parę ciekawych konstrukcji programistycznych dostępnych w języku C:

1. Wskaźniki na funkcje.
2. Struktury.
3. Definiowanie typów.
4. Typy wyliczeniowe.
5. Skończone maszyny stanów.

Wszystkie te rozwiązania umieszczone zostały w plikach *KBD.c* oraz *KBD.h*, ale o tym będzie już Wasze zadanie.

Podsumowanie i zadania

W czasie naszego spotkania udało nam się zapoznać obsługą przerwań zewnętrznych oraz z efektem „drgań styków". Potem jednak szybko poradziliśmy sobie z tym problemem poradziliśmy wykorzystując do tego timer. Ważną nauczką z naszych ostatnich zmagań powinien być też fakt możliwości wykorzystania jednego timera do wielu zadań – często w projektach zachodzi konieczność odmierzania różnych czasów, obsługi wielu przycisków, generowania opóźnień pomiędzy różnymi zadaniami – w znakomitej większości takich przypadków nie potrzeba nam więcej niż jednego licznika!

Jakie jednak będzie zadanie tym razem? Jako iż nie jest to kurs poświęcony językowi C, który to służy nam jedynie za narzędzie w „fpg-owym" warsztacie, ale z drugiej strony im lepiej dialekt ten znamy, tym łatwiej sobie poradzimy w różnych sytuacjach. Tym razem zachęcam w ramach pracy domowej do dokładnej analizy wspomnianego „zaawansowanego" kodu obsługi przycisków (czy klawiatury), oraz zapoznania się z pięcioma zagadnieniami wypunktowanymi kilka linijek wcześniej.

Ponadto, macie już pewną wiedzę, która może zostać wykorzystana do stworzenia własnych projektów, czemu by np. nie wykonać minutnika, licznika rowerowego, czy innego ciekawego projektu?

Pozostaje mi życzyć tylko powodzenia w zgłębianiu wiedzy na temat języka i zaprosić na kolejne spotkanie, w którym zajmiemy się obsługą transmisji szeregowej (UART). Dlatego warto wyposażyć się w przejściówkę USB ⟷ UART, lub inne urządzenie pozwalające na komunikację po tym interfejsie, najlepiej w standardzie napięciowym 3,3 V (choć ze standardem 5 V także sobie poradzimy).

Małe post scriptum, czyli drobne uzupełnienie

Jest jeszcze kilka drobnych, acz ważnych spraw związanych z timerami i przerwaniami, o których muszę wspomnieć.

W naszym obecnym projekcie czas mierzymy z dokładnością do 1 ms. Jeśli potrzeba by było większej dokładności to… no właśnie, co zrobić? Pierwsza (niezbyt dobra) opcja to zwiększenie częstotliwości występowania przerwań. Ale uwaga! Jeśli przesadzimy z tym parametrem, to procesor może nie zdążyć z wykonaniem wszystkich instrukcji przerwania, zanim zgłoszone zostanie następne, lub pomiędzy zakończeniem obsługi jednego przerwania a zgłoszeniem kolejnego będzie bardzo mało czasu. Wtedy procesor będzie non-stop spędzał czas w przerwaniu, nie obsługując nawet innych przerwań, ani nie wracając do pętli głównej (u nas na razie jest pusta, ale wkrótce to się zmieni).

Druga opcja to z jednej strony liczyć tak jak obecnie milisekundy, ale w celu dokładniejszego pomiaru czasu odczytywać wartość licznika momencie jakiegoś zdarzenia. W tym jednak miejscu, przynajmniej w mojej głowie, zapala się pomarańczowa lampka dotycząca dostępu atomowego do tych danych. Kluczowa część programu mogłaby wyglądać tak:

```
IOWR_ALTERA_AVALON_TIMER_SNAPH(TIMER0_BASE, 0);
uint16_t copyMs = counterMs;
```

Pierwsze (zgodnie z tym, czego dowiedzieliśmy się już wcześniej) instrukcja powodująca skopiowanie aktualnego stanu licznika do rejestrów *SNAPH/L*, a potem odczyt licznika milisekund, zwiększanego w przerwaniu timera. Co jednak, jeśli między tymi instrukcjami wystąpi przerwanie? Musimy temu zapobiec – najlepiej wyłączając na krótką chwilę przerwania.

Jak wyłączyć przerwania, czyli o dostępie atomowym raz jeszcze

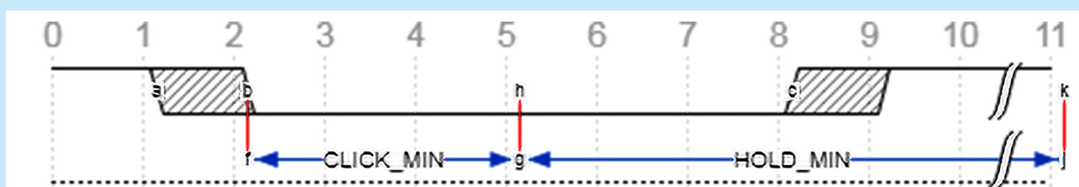
Aby w naszym systemie wyłączyć, a potem włączyć przerwania musimy wykonać następujące zadania:

```
alt_irq_context context = alt_irq_disable_all();
alt_irq_enable_all(context);
```

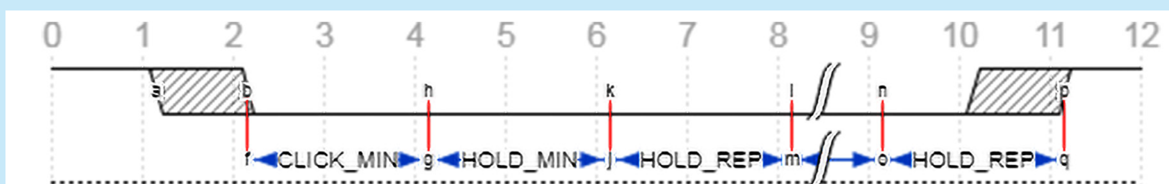
Po co jednak ta zmienna *context*? Przechowuje ona informację o tym, czy przerwania nie były już wyłączone w momencie wywołania instrukcji ich wyłączenia. Przeanalizujmy potencjalną sytuację, w której korzystamy z biblioteki tworzonej przez inną osobę, która też korzysta z wyłączania i włączania przerwań, np. wg takiego schematu:

1. Wyłączenie przerwań.
2. Wywołanie funkcji bibliotecznej:
   a. wyłączenie przerwań,
   b. instrukcje z biblioteki,
   c. włączenie przerwań.
3. Nasze instrukcje.
4. Włączenie przerwań.

Jeśli w tej sytuacji funkcja biblioteczna wykonywałaby bezwzględne włączenie przerwań, nasze instrukcje wykonywałyby się



Rysunek4. Jeśli zwolnienie przycisku nastąpiło po odmierzeniu czasu CLICK_MIN, ale przed odmierzeniem czasu HOLD_MIN, to zdarzenie takie interpretujemy jako kliknięcie i w momencie oznaczonym liczbą 8 nastąpi wykonanie odpowiedniej akcji (kliknięcia)



Rysunek 5. Jeśli przycisk był trzymany dostatecznie długo, po odmierzeniu czasu HOLD_MIN (w chwili czasu oznaczonej 6) następuje wykonanie akcji przytrzymania, które jest powtarzana dopóki przycisk jest wciśnięty, czyli w 8, 9, ale już nie w momencie 11

Poprzednie części kursu i dodatkowe materiały dostępne są na na stronie www.media.avt.pl

przy włączonych przerwaniach. Natomiast, jeśli skorzystamy z niejako narzuconej przez producenta formy realizacji tego zadania, będziemy w 100% bezpieczni. Ostatecznie nasz program może wyglądać tak, jako pokazano na **listingu 4**.

Listing 4. Pomiar czasu z użyciem funkcji dostarczanej przez producenta

```
volatile uint16_t counterMs = 0;
void timer0Interrupt(void* context){
    IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER0_BASE, 0);
    refreshDisplay();
    counterMs++;

}
// w pętli while
if(!(IORD_ALTERA_AVALON_PIO_DATA(SW_BASE) & (1<<0))){
    alt_irq_context context = alt_irq_disable_all();
    IOWR_ALTERA_AVALON_TIMER_SNAPH(TIMER0_BASE, 0);
    uint16_t copyMs = counterMs;
    // (*)
    alt_irq_enable_all(context);
    uint32_t timerValue = IORD_ALTERA_AVALON_TIMER_SNAPH(TIMER0_BASE);
    timerValue <<= 16;
    timerValue |= IORD_ALTERA_AVALON_TIMER_SNAPL(TIMER0_BASE);
    printf("T = %u.%03lu ms\r\n", copyMs, 999-timerValue/50);
    intDisplayDec(copyMs);
    while(!(IORD_ALTERA_AVALON_PIO_DATA(SW_BASE) & (1<<0)));
}
```

Wszystko super (rzecz jasna nie mamy eliminacji drgań styków, ale nie o to nam chodzi), mam nadzieję, że każdy wie, iż zapis 999-timerValue/50 wynika z faktu, że licznik zlicza w dół, oraz że aby otrzymać mikrosekundy musimy podzielić jego wartość przez 50 (ponieważ zegar ma częstotliwość 50 MHz). Uważniejsi obserwatorzy jednak zaraz postawią mi zarzut – przecież timer cały czas działa, po wyłączeniu przerwań mógł się on przepełnić i wtedy po pierwsze zgubimy jedną milisekundę, a po drugie wartość odczytana z rejestru *SNAP* będzie przekłamana.

Pierwszy zarzut na szczęście nie sprawdzi się – nawet, jeśli nastąpi zgłoszenie przerwania (które nie zostanie obsłużone z powodu wyłączenia przerwań), to zostanie ono „zapamiętane" i zaraz po włączeniu przerwań nastąpi jego obsługa (pomijam niedopuszczalny fakt, gdyby przerwania były wyłączone na zbyt długi okres czasu i zgłoszone zostałyby 2 lub więcej przerwań.).

Drugi zarzut jest jednak w 100% słuszny, pomimo iż prawdopodobieństwo takiej sytuacji nie jest wielkie. Można by temu zaradzić zatrzymując timer na czas odczytu (jednak wtedy ze 100% pewnością wprowadzać będziemy błąd w naszym pomiarze czasu), albo sprawdzając, czy w momencie zatrzymania przerwań nie doszło do wyzerowania się licznika. Jeśli tak to cały odczyt może być obarczony błędem nawet 1 ms (w przeciwnym wypadku błąd to co najwyżej czas wykonania instrukcji, od momentu wykrycia stanu niskiego na pinie, do momentu zatrzaśnięcia wartości rejestrów *SNAPH/L*). Aby dodać takie sprawdzenie wystarczy w miejscu oznaczonym gwiazdką w komentarzu dodać poniższą instrukcję odczytu rejestru statusu, oraz zmodyfikować funkcję wyświetlającą czas w konsoli Eclipse (korzystamy z JATG UART).

```
uint8_t valid = !(IORD_ALTERA_AVALON_TIMER_
STATUS(TIMER0_BASE) & ALTERA_AVALON_TIMER_STATUS_
TO_MSK);
// ...
printf("T = %u.%03lu ms %c \r\n", copyMs,
999-timerValue/50, valid?' ':'*');
```

Oczywiście podejście zależy od tego, jaki czas chcemy mierzyć. Czasem zasadne może być użycie osobnego timera do takich celów i uruchamianie go a potem zatrzymywanie, w celu odmierzenia czasu między jakimiś zdarzeniami – wszystko zależy od konkretnego problemu, jaki chcemy rozwiązać, a także od tego, jakiej dokładności oczekujemy.

Piotr Rzeszut, AGH

REKLAMA

ELEKTRONIKA PRAKTYCZNA na facebook

https://www.facebook.com/ElektronikaPraktyczna

Poprzednie części kursu i dodatkowe materiały dostępne są na stronie www.media.avt.pl