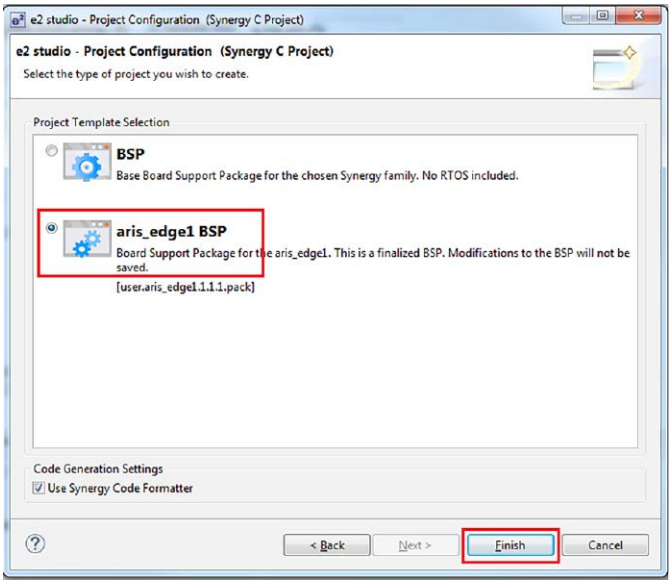
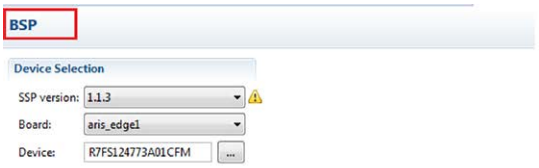
Synergy RTOS

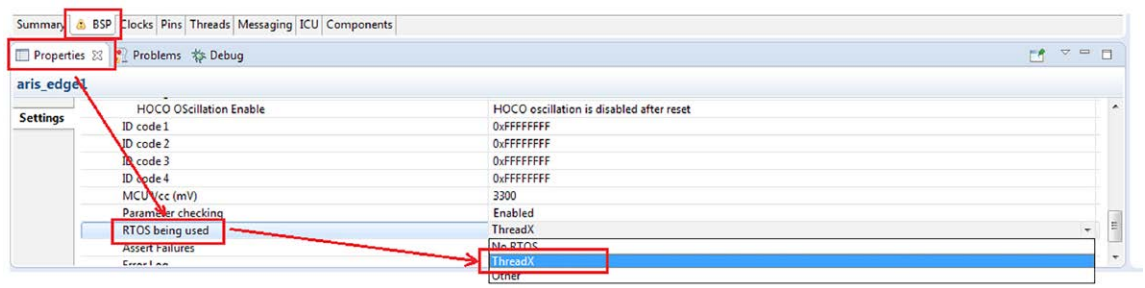
Układy wbudowane często wykonują skomplikowane zadania. Jeżeli stopień skomplikowania jest bardzo duży, to programistom jest łatwiej podzielić wykonywane zadanie na małe wątki i uruchomić pod systemem czasu rzeczywistego RTOS. Dla procesorów Synergy firma Renesas dostarcza RTOS dystrybuowany pod nazwą ThreadX. Pokażę na prostym przykładzie, w jaki sposób wykonać w środowisku e2studio projekt wspierający programowanie pod kontrolą ThreadX. Do przeprowadzania testów zostanie użyty moduł ewaluacyjny ARIS EDGE firmy ARRIS.

Nowy projekt tworzymy standardowo: *Fil* → *New* → *Synergy Project*. Po nadaniu nazwy projektu (Aris_RTOS) i o wyborze typu procesora, wersji biblioteki SSP, kompilatora GCC ARM itp. w dwóch pierwszych oknach Project Configuration wybieramy szkielet i zaznaczamy opcję aris_edge1 BSP, jak na **rysunku 1**. Ten szkielet projektu został wcześniej zdefiniowany przeze mnie. Można również zaznaczyć uniwersalny szkielet BSP. Praca pod kontrolą ThreadX wymaga dodania niezbędnych systemowych plików źródłowych. Można to zrobić ręcznie, ale dużo lepszym pomysłem będzie zlecenie tej czynności konfiguratorowi wbudowanemu w e2studio. W tym celu otwieramy perspektywę Synergy Configurator i klikamy na zakładkę BSP.

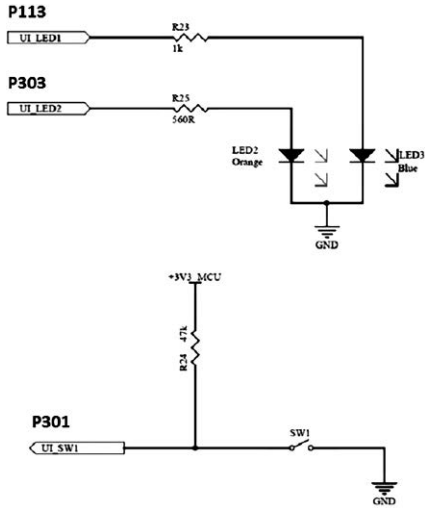


Rysunek 1. Wybór szkieletu projektu





Rysunek 2. Konfigurowanie zakładki BSP – praca z RTOS ThreadX



Rysunek 3. Podłączenie diod LED i styku SW1 do linii portów

W oknie właściwości (Properties) klikamy na RTOS being used i z rozwijanej listy wybieramy ThreadX, tak jak to zostało pokazane na **rysunku 2**. Teraz, kiedy konfigurator już „wie", że ma konfigurować projekt do pracy z ThreadX, można dodawać wątki zadania do wykonania. Każdy wątek to funkcja zawierająca pętlę nieskończoną *while(1)* {}. Funkcje wątków mogą zależnie od systemu RTOS mieć jakieś argumenty. W konfiguratorze e2studio wątki definiuje się bardzo prosto. Otwieramy zakładkę *Threads* i w oknie *Threads* klikamy na ikonkę dodawania (z zielonym plusem). Po dodaniu wątku w oknie właściwości zmieniamy jego nazwę adekwatnie do wykonywanego zadania. Ja zmieniłem nazwę na **LED Thread**, ponieważ wątek będzie miał za zadanie sterowanie diodą LED. Zmieniłem też nazwę symbolu na **led_thread**. Ta nazwa pojawi się potem w kodzie i warto również powiązać ją z wykonywanym zadaniem.

Ponieważ naszym zadaniem jest pokazanie konfiguracji projektu do pracy pod systemem RTOS, to przykładowa aplikacja będzie nieskomplikowana. Program będzie cyklicznie gasił i zapalał niebieską diodę LED3 po każdym naciśnięciu przycisku SW1 na module ewaluacyjnym. Podłączenie diod LED i styku SW1 do linii portów zostało pokazane na **rysunku 3**. Dioda LED3 jest dołączona do linii portu P113. Żeby ta linia mogła sterować diodą LED3, trzeba ją skonfigurować jako wyjściowy port GPIO. Można to zrobić z poziomu konfiguratora e2studio po wybraniu zakładki Pins (**rysunek 4**). Styk SW1 jest połączony z linią przerwania zewnętrznego IRQ6 (linia portu P301). Ta linia musi być skonfigurowana w zakładce Pins, tak jak to zostało pokazane na **rysunku 5**.

Każde naciśnięcie SW1 ma powodować zgłoszenie przerwania. Każde przerwanie trzeba skonfigurować i obsłużyć przez napisanie procedury obsługi. Konfigurację przerwania, a przerwania ze-

wnętrznego w szczególności można wykonać za pomocą konfiguratora. Najpierw dodajemy do wątku moduł drivera obsługującego przerwanie zewnętrzne, jak na **rysunku 6**.

Konfiguracja przerwania jest wykonywana we właściwościach drivera:

- Priorytet 2.
- Kanał 6 (IRQ6).
- Wyzwalanie opadającym zboczem.
- Włączenie cyfrowego filtra drgań styku.
- Nazwa procedury obsługi external_irq6_callback – włączenie mechanizmu callback.

Parametry konfiguracji właściwości zostały pokazane na **rysunku 7**. Jeżeli w zakładce Callback okna Properties drivera IRQ zamiast opcji NULL wpiszemy jakąś swoją nazwę, to konfigurator zdefiniuje prototyp funkcji o tej nazwie wywoływanej w momencie zgłoszenia przerwania. Argumentem tej funkcji jest wskaźnik na strukturę zawierającą między innymi informacje o zdarzeniach dotyczących zmiany stanów na linii P301. Użytkownikowi pozostaje tylko napisać tę swoją funkcję, umieścić ją w programie źródłowym i testować tam zdarzenie przyciśnięcia SW1. Ta funkcja musi być formalnie zdefiniowana zgodnie z wymaganiami biblioteki SSP, tak jak to zostało pokazane na **listingu 1**. Nazwa procedury: irq6_callbacki, została nadana w oknie Callback zakładki properties drivera przerwania IRQ6 (rys. 7).

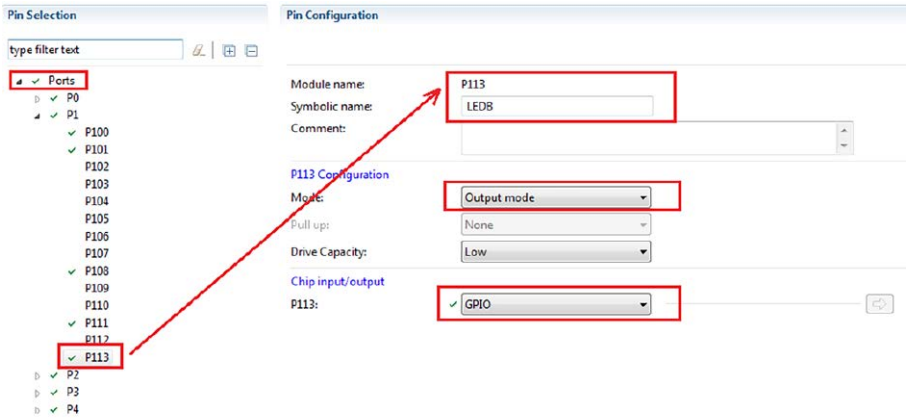
Jak pamiętamy, nasz program ma cyklicznie zapalać i gasić diodę LED po każdym naciśnięciu przycisku SW1. Do tego celu wykorzystamy zdefiniowany już mechanizm zgłaszania przerwania od zmiany stanu SW1 oraz semafor. Semafor dodajemy z poziomu konfiguratora, jak pokazano na **rysunku 8**. W oknie właściwości nadajemy swoją nazwę semafora – ja nazwałem go sw1_semaphore. Teraz projekt jest skonfigurowany i po kliknięciu na Generate Project Content zostaną wygenerowane odpowiednie pliki. W katalogu src zostały umieszczone pliki źródłowe *hal_entry.c* i *led_thread_entry.c* przeznaczone do edytowania przez użytkownika (**rysunek 9**). Oczywiście, konfigurator dodaje również pliki jądra RTOS, ale nimi nie będziemy się zajmować.

Na **listingu 2** pokazano plik *Led_thread_entry.c*. Ten plik trzeba teraz zmodyfikować, aby zmieniał stan diod, czekając na aktywowanie semafora. Umieścimy tam też procedurę obsługi przerwania, która aktywuje semafor po naciśnięciu przycisku SW3. Pokazano to na **listingu 3**.

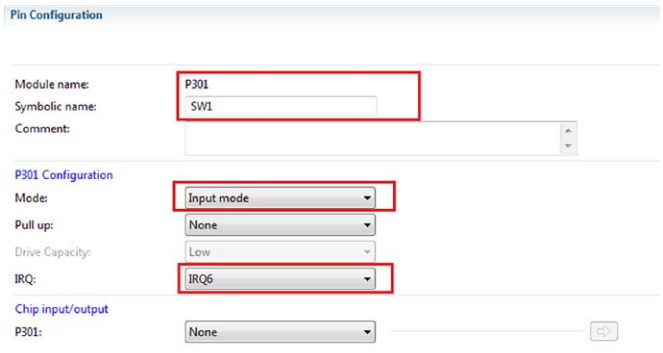
Działanie programu jest bardzo proste. O stanie diody LED (zapalona lub zgaszona, decyduje wartość wpisana do zmiennej level (zainicjowana na wartość 1). Funkcja biblioteki warstwy HAL *g_ioport.p_api –> pinWrite* zmienia stan linii portu w zależności od wartości zmiennej *level*. Potem zmienna *level* zmienia swój stan na przeciwny i program jest zatrzymywany na semaforze sw1_semaphore do czasu jego zwolnienia. Zwolnienie semafora

Listing 1. Funkcja callback

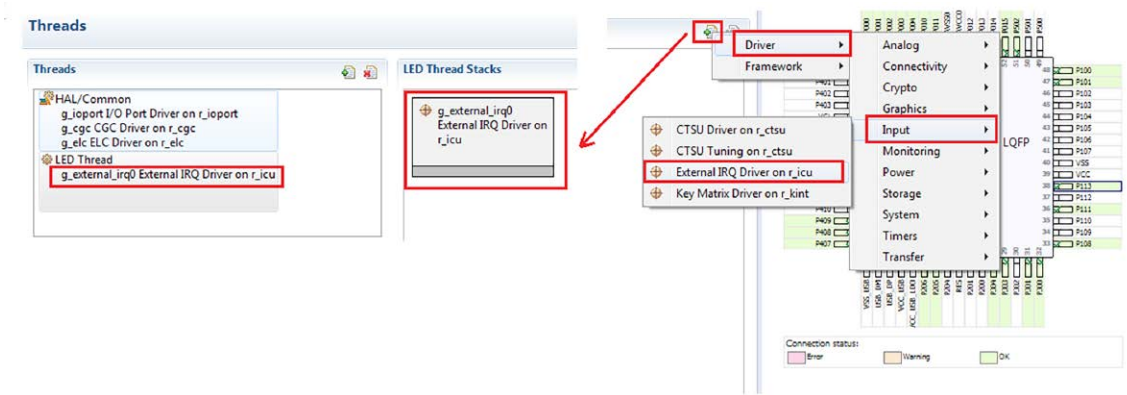
```
//*********************************************************
//prototyp funkcji calback dla przerwania zewnętrznego IRQ6
//*********************************************************
void irq6_callback(external_irq_callback_args_t * p_args)
{
    /* Post to the semaphore to indicate SW1 has been pressed. */
    tx_semaphore_put(&sw1_semaphore);//zwolnij  semafor sw1_semaphore
}
```



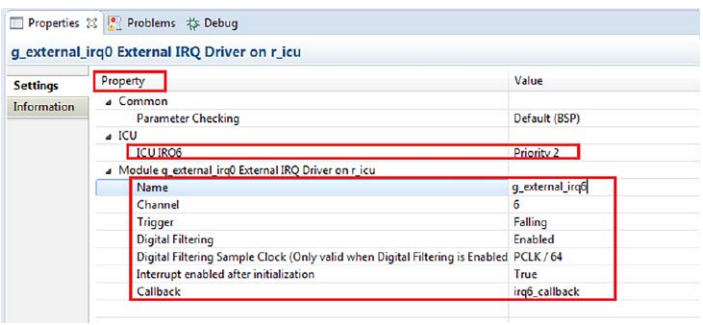
Rysunek 4. Konfiguracja linii P113 sterującej diodą LED3



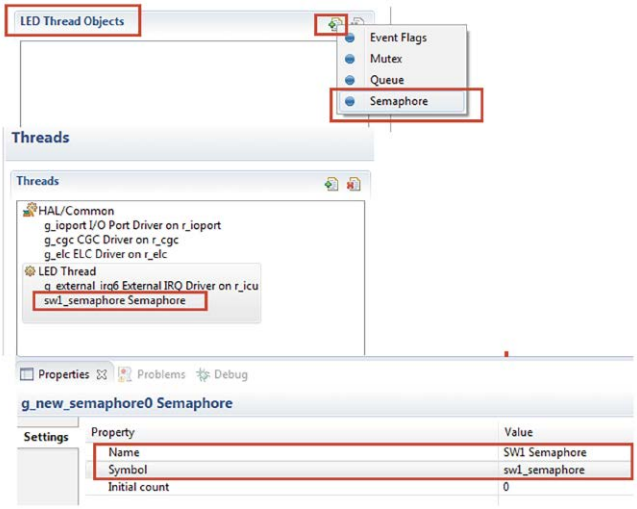
Rysunek 5. Konfiguracja linii P301 skojarzona z przerwaniem IRQ6



Rysunek 6. Dodanie drivera obsługi przerwania zewnętrznego



Rysunek 7. Konfiguracja przerwania IRQ6



Rysunek 8. Dodanie semafora sw1_semaphore

jest wykonywane w procedurze przerwania zewnętrznego IRQ6 zgłaszanego po naciśnięciu przycisku SW1. W pętli nieskończonej ponownie jest wpisywana wartość zmiennej level do rejestrów portu P113 i stan diody LED zmienia się na przeciwny. Potem program zatrzymuje się na semaforze i cały cykl jest wykonywany ponownie. Pozostaje teraz skompilowanie projektu, zapisanie kodu wynikowego do pamięci Flash mikrokontrolera i sprowadzenie poprawności działania.

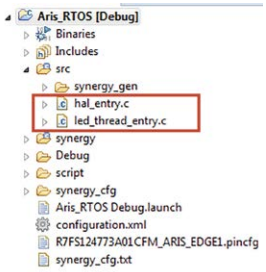
Pokazaliśmy tu konfigurowanie projektu do pracy z systemem RTOS ThreadX. Przy takim postepowaniu konfigurator pakietu e2studio sam zadba o konfigurację RTOS oraz o dołączenie bibliotek potrzebnych do pracy ThreadX. Użytkownik może również z poziomu konfiguratora dodawać swoje wątki oraz semafory, kolejki i inne. Zależnie od potrzeb można dodawać i konfigurować drivery oraz realizować obsługę przerwań z wykorzystaniem mechanizmu callback. Sam przykład cyklicznego zaświecania i gaszenia diody LED został całkowicie podporządkowany pokazaniu konfiguracji projektu z ThreadX i z tego powodu nie jest przewodnikiem po programowaniu w środowisku RTOS.

Tomasz Jabłoński, EP

Listing 2. Plik led_thread_entry.c

```
#include „led_thread.h”

void led_thread_entry(void);
/* LED Thread entry function */
void led_thread_entry(void)
{
    /* TODO: add your own code here */
    while (1)
    {
        tx_thread_sleep (1);
    }
}
```



Rysunek 9. Pliki projektu wygenerowane przez konfigurator

Listing 3. Plik led_thread_entry.c po modyfikacji

```
#include „led_thread.h”

void led_thread_entry(void);
void irq6_callback_internal(external_irq_callback_args_t * p_args);
/* LED Thread entry function */
void led_thread_entry(void)
{
    /* TODO: add your own code here */
    ioport_level_t level = IOPORT_LEVEL_HIGH;//inicjalizacja zmiennej stanu diody LED
    //konfiguruj przerwanie zewnętrzne od zmiany stanu na SW1
    g_external_irq6.p_api->open(g_external_irq6.p_ctrl, g_external_irq6.p_cfg);
    while (1)
    {
        g_ioport.p_api->pinWrite (IOPORT_PORT_01_PIN_13, level);//wystawienie stanu na linię P113
        level=1-level;//zamiana stanu didot LED na przeciwny
        tx_semaphore_get(&sw1_semaphore, TX_WAIT_FOREVER);//czekanie na zwolnienie semafora
        tx_thread_sleep (1);
    }
}

//obsługa przerwania zgłaszana po naciśnieciu SW1
void irq6_callback(external_irq_callback_args_t * p_args)
{
    /* Post to the semaphore to indicate SW1 has been pressed. */
    tx_semaphore_put(&sw1_semaphore);//zwolnij  semafor
}
```