ISIX-RTOS v3 – System operacyjny dla mikrokontrolerów rodziny Cortex-M0/M3/M4/M7 (3)

API systemowe w przykładach

FTP WSZYSTKIE POPRZEDNIE ODCINKI KURSU SĄ DOSTĘPNE NA STRONIE WWW.MEDIA.AVT.PL

W poprzedniej części artykułu skonfigurowaliśmy podstawowe środowisko IDE oraz przygotowaliśmy potrzebne narzędzia niezbędne do pracy. W dzisiejszej części skupimy się na zaprezentowaniu w przykładach najczęściej używanej części API jądra systemowego ISIX, do których zaliczyć możemy: tworzenie wątków, zasady synchronizacji, przekazywania danych pomiędzy wątkami.

Do realizacji przykładów z bieżącego odcinka potrzebować będziemy:

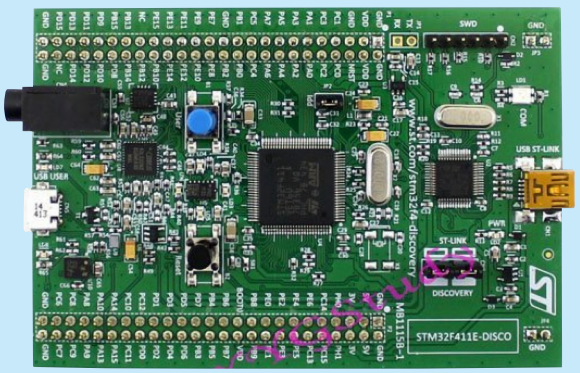
- Zestawu STM32F411E-DISCO – **fotografia 1**,
- Minimodułu KAMOD-LED8 zawierającego 8 diod LED (do przykładu 5) – **fotografia 2**,
- Minimodułu WSR-04499, zawierającego 8 przycisków tact switch (do przykładu 5) – **fotografia 3**,
- Zestawu przewodów połączeniowych (do przykładu 5) – **fotografia 4**,
- Dowolnej przejściówki Serial 3,3 V ↔ USB (do przykładu 2).

Aby skompilować przykłady, należy je wcześniej pobrać z repozytorium GIT. W tym celu uruchamiamy ikonkę GIT BASH, a następnie w wierszu polecenia wydajemy polecenie:

```
git clone -b pub/ep0419 --recursive https://www.
boff.pl/cgit/public/isixsamples
```

Gdy pobieranie się skończy przykłady możemy skompilować za pomocą następującej sekwencji poleceń wydanych w wierszu polecenia:

```
        waf configure -v --board=stm32f411e_disco
--crystal-hz=8000000 –debug
        waf
```



Fotografia 1.

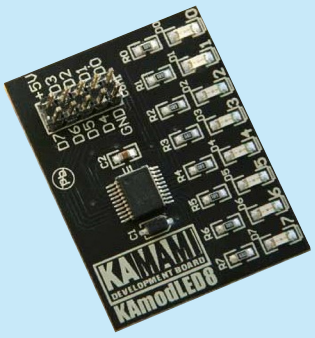
lub bezpośrednio z poziomu *Visual Studio Code*, tak jak opisano w poprzedniej części kursu.

Aby zaprogramować zestaw ewaluacyjny jednym z wybranych przykładów należy wywołać polecenie:

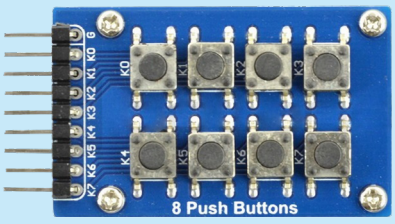
```
waf program
```

Po wykonaniu polecania system budowania zapyta o numer przykładu, który chcemy załadować (**rysunek 5**).

Opcja 0 oznacza zakończenie działania bez programowania, natomiast pozostałe opcje umożliwiają wybranie jednego z przykładów i zaprogramowanie płytki ewaluacyjnej.



Fotografia 2.



Fotografia 3.

Fotografia 4.

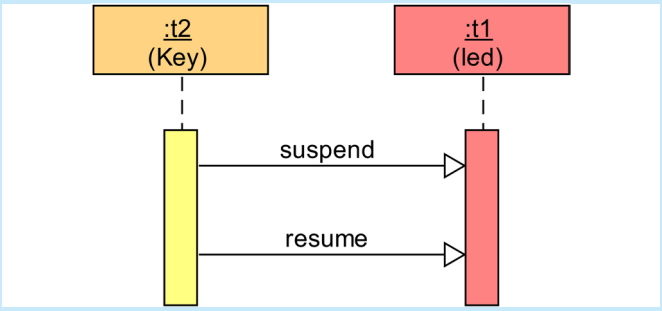
Przykład 1. Tworzenie, wybudzanie oraz usypianie wątków

Jedną z podstawowych czynności, wykonywanych przy pisaniu oprogramowania wykorzystującego systemy operacyjne, jest

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

```
Waf: Entering directory `/home/lucck/worksrc/internal/isixsamples/build'
Please select the application:
[0].    Abort
[1].    blink
[2].    mutexes
[3].    suspend resume
[4].    semaphores
[5].    fifos
[6].    events
```

Rysunek 5.



Rysunek 6.

tworzenie wątków oraz zarządzanie nimi. W pierwszym przykładzie pokażemy, w jaki sposób tworzyć nowe wątki z wykorzystaniem API C++11 oraz jak wybudzać oraz usypiać wątek z poziomu innego wątku. W tym celu utworzymy dwa wątki: wątek **t1**, którego zadaniem będzie błyskanie z częstotliwością 2 Hz diodą LD3 oraz wątek **t2**, który po każdym wciśnięciu przycisku USER będzie naprzemiennie usypiał oraz wybudzał watek **t1** (**rysunek 6**). Wynikiem działania powyższej aplikacji będzie naprzemienne wznawianie oraz wstrzymywanie błyskania diodą LED po każdym naciśnięciu klawisza USER.

Listing 1.
```
auto main() -> int
{
  // Create task for blinking
  static auto t1 = isix::thread_create_and_run(ISIX_MIN_STACK_SIZE,
      isix::get_min_priority(), isix_task_flag_suspended, blink_task );
  // Create task 2
  static auto t2 = isix::thread_create_and_run(ISIX_MIN_STACK_SIZE,
     isix::get_min_priority(),0, keyb_scan_task, std::ref(t1));
  isix::start_scheduler();
  return 0;
}
```

Listing 2.
```
    auto blink_task() -> void {
        // Configure gpio as output
        periph::gpio::setup( led_0,
            periph::gpio::mode::out{
                periph::gpio::outtype::pushpull,
                periph::gpio::speed::low
        } );
        //! Blinking loop
        for(;;) {
            periph::gpio::toggle(led_0);
            isix::wait_ms(500);
        }
    }
```

Listing 3.
```
    auto keyb_scan_task( isix::thread& task_ctl ) -> void {
        // Configure gpio as input
        periph::gpio::setup(key_0,
            periph::gpio::mode::in{periph::gpio::pulltype::down}
        );
        // Suspend resume flag
        bool resume {};
        for(auto pstate=false;;) {
            //Get key
            const auto val = periph::gpio::get(key_0);
            // On rising edge change state
            if(val && !pstate) {
                resume = !resume;
                //Suspend or resume task
                if(resume) task_ctl.resume();
                else task_ctl.suspend();
            }
            //Save previous state
            pstate = val;
            isix::wait_ms(10);
        }
    }
```

Program rozpoczyna działanie od wykonania funkcji *main* (**listing 1**), której jedynym zadaniem jest utworzenie nowych wątków oraz uruchomienie algorytmu szeregującego systemu ISIX.

Tworzenie wątku umożliwia funkcja *thread_create_and_run*, która jako argumenty przyjmuje odpowiednio: rozmiar stosu, priorytet, flagi oraz adres funkcji, która będzie stanowić kod wątku. Po wskazaniu funkcji wątku przekazujemy listę argumentów, które będą przesłane do funkcji stanowiącej kod wątku. Funkcja zwraca obiekt *isix::thread*, który zawiera metody umożliwiające sterowanie pracą wątku.

Jako pierwszy tworzymy wątek o identyfikatorze **t1**, którego kod stanowi funkcja *blink_task*, realizująca zadanie błyskania diodą LED. Przekazanie flagi *isix_task_flag_suspended* spowoduje utworzenie wątku w stanie uśpionym, a zatem po utworzeniu wątek przejdzie natychmiast w stan uśpienia i nie będzie zaszeregowany do wykonania. W następnej kolejności tworzony jest wątek o identyfikatorze **t2,** który będzie wykonywał kod funkcji *keyb_scan_task*, realizujący zadanie odczytywania stanu klawisza. Do funkcji przekazana została referencja do obiektu **t1,** umożliwiając późniejsze kontrolowanie stanu wątku błyskającego diodą. Jak wcześniej wspomniano, za błyskanie diodą LED odpowiedzialna jest funkcja *blink_task* (**listing 2**).

Na początku port **led_0** *(PD13)* konfigurowany jest w kierunku Wyjścia, a następnie funkcja wchodzi w pętlę nieskończoną, która składa się z dwóch instrukcji: zmiany stanu portu GPIO (led_0) na przeciwny oraz uśpieniu wątku na 500 ms za pomocą wywołania systemowego *isix::wait_ms*.

Za realizację odczytu kodu klawisza w wątku **t2** odpowiedzialna jest funkcja *keyb_scan_task* (**listing 3**).

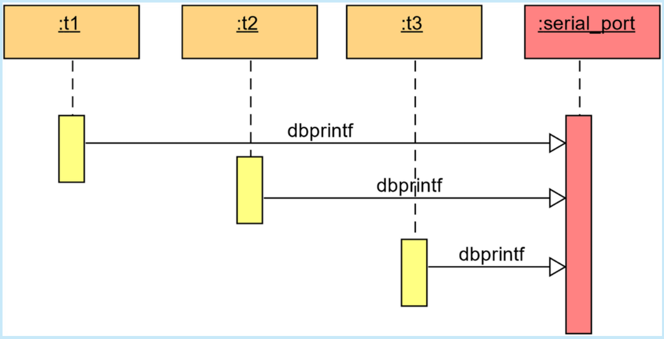
Funkcja przyjmuje jeden argument *task_ctl*, do którego podczas tworzenia wątku przekazywana jest referencja do obiektu *t1: isix_thread*. *Klasa isix::thread* umożliwia sterowanie pracą wątku, którego jest właścicielem. Na początku funkcji linia GPIO **key_0** (*PA0*) konfigurowana jest w kierunku wejścia z rezystorem pull-down. Następnie funkcja wchodzi w pętlę nieskończoną, gdzie co 10 ms sprawdzany jest stan klawisza *USER0*. Jeśli zostanie wykryte zbocze narastające, wówczas stan zmiennej *resume* zmieniany jest na przeciwny. Jeśli zmienna resume przyjmie wartość logiczną TRUE, wywoływana jest metoda *resume()*, na referencji do obiektu t1 (*isix::thread& task_ctl*), co spowoduje uruchomienie wątku **t1** odpowiedzialnego za mruganie diodą LED. Jeśli zmienna *resume* przyjmie wartość logiczną FALSE, wywoływana jest metoda *suspend()*, a w konsekwencji uśpienie wspomnianego wcześniej wątku, a zatem zatrzymanie mrugania diody LED.

Wznawianie oraz wstrzymywanie wątku jest najprostszym sposobem interakcji pomiędzy wątkami, co realizują metody *suspend/resume* klasy *isix::thread*.

Kod źródłowy przykładu możemy znaleźć w pliku: *stm32f411e_disco/e2_task_suspend/src/task_suspend.cpp*. Aby uruchomić powyższy przykład, należy wpisać polecenie waf program, a następnie wybrać opcję **[3].** Po załadowaniu przykładu, po każdorazowym wciśnięciu przycisku **USER,** możemy zaobserwować cykliczne wznawianie lub zatrzymywanie błyskania diody LED.

Przykład 2. Użycie i zastosowanie MUTEX-ów

Tworząc oprogramowanie wielowątkowe, często istnieje konieczność, aby w danym momencie tylko jeden wątek miał dostęp do danego zasobu. Typowym przykładem jest wysyłanie komunikatów do portu szeregowego. Jeśli dwa wątki w tym samym czasie rozpoczęłyby pisanie do portu szeregowego, wówczas komunikaty wysłane na port szeregowy uległyby wzajemnemu wymieszaniu. Aby zapobiec temu zjawisku, należy wykorzystać mechanizmy synchronizacji międzywątkowej, które w jednym czasie umożliwią zajęcie danego zasobu (portu szeregowego) tylko przez jeden wątek. Najczęściej używanym mechanizmem synchronizacyjnym jest mechanizm wzajemnego wykluczenia **MUTEX**



Rysunek 7.

Listing 4.

```
namespace {
    /* Initialize the debug USART */
    auto usart_protected_init() -> void {
        static isix::mutex m_mtx;
        dblog_init_locked(
                [](int ch, void*) {
                    return periph::drivers::uart_early::putc(ch);
                },
                nullptr,
                []() { m_mtx.lock(); },
                []() { m_mtx.unlock(); },
                periph::drivers::uart_early::open, „serial0”, 115200
        );
    }
}


auto main() -> int
{
    static constexpr auto tsk_stk_size = 1024;
    static constexpr auto tsk_prio = 1;
            isix::wait_ms( 500 );
    usart_protected_init();
    dbprintf(„<<<< Example 1. Mutexes >>>>”);
    // Create task as a lambda function
    auto task_fn =
        []( int tsk_no ) {
            for(;;) {
                dbprintf(„Hello from task %i”,tsk_no);
                isix::wait_ms(500);
            }
    };
    // Create task 1
    static auto t1 = isix::thread_create_and_run(tsk_stk_size,tsk_prio,0,task_fn,1);
    // Create task 2
    static auto t2 = isix::thread_create_and_run(tsk_stk_size,tsk_prio,0,task_fn,2);
    // Create task 3
    static auto t3 = isix::thread_create_and_run(tsk_stk_size,tsk_prio,0,task_fn,3);
    isix::start_scheduler();
    return 0;
}
```

(*MUTal Exclusion*). Mutex ma dwa stany: otwarty (niezajęty przez żaden wątek) lub zajęty (przez jeden wątek). Mutex nie może być w posiadaniu przez więcej niż jeden wątek naraz. Wątek próbujący zająć mutex będący w posiadaniu innego wątku jest zawieszany do czasu zwolnienia wątku przez proces, który go wcześniej zajął. W systemie **isix** mutex jest reprezentowany przez klasę *isix::mutex*, a jego dwie najważniejsze metody to metoda *lock()*, która zajmuje mutex oraz metoda *unlock()*, która zwalnia mutex. Pokażemy teraz na przykładzie wspomnianego mechanizmu pisania do portu szeregowego zasadę działania tego mechanizmu. Stworzymy trzy wątki, które w tym samym czasie będą pisały komunikaty tekstowe do portu szeregowego, i sprawdzimy, jaki będzie rezultat, jeśli funkcja logująca *dbprintf()* będzie zabezpieczona mutexem oraz przy braku takiego zabezpieczenia (**rysunek 7**).

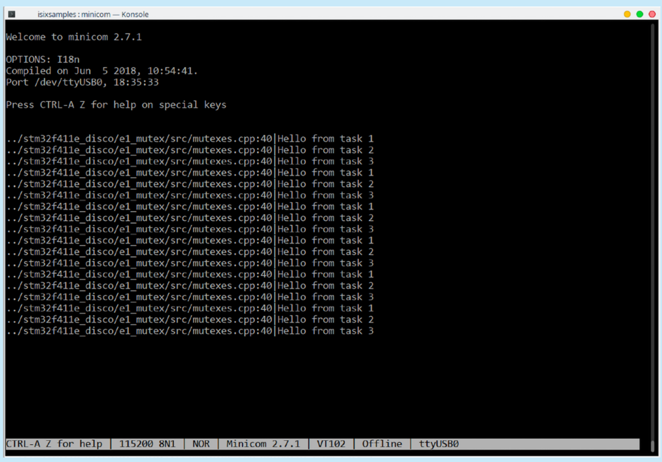
Kod programu jest bardzo prosty i stanowi drobną modyfikacje przykładu z poprzedniego odcinka kursu (**listing 4**).

Spójrzmy na funkcję *usart_protected_init*, która jest odpowiedzialna za mechanizm logowania systemu ISIX. Na początku tworzony jest mutex o nazwie *m_mtx* oraz wywoływana jest funkcja *dblog_init_locked*, która jako pierwszy argument przyjmuje adres funkcji wysyłającej znak na port szeregowy, jako drugi argument funkcję odpowiedzialną za blokowanie, a jako trzeci argument funkcję odpowiedzialną za odblokowanie funkcji na czas wysyłania komunikatu. W naszym przypadku jako funkcja wysyłająca dane na port szeregowy wykorzystana jest funkcja *usart_early::putc* z biblioteki urządzeń peryferyjnych, która jest opakowana w funkcji lambda. Podobnie w funkcji odpowiedzialnej za blokowanie tworzona jest funkcja lambda, która wywołuje metodę *m_mtx.lock()*, przejmującą mutex, natomiast w funkcji lambda odpowiedzialnej za odblokowanie wywoływana jest metoda *m_mtx.unlock()* zwalniająca mutex. Obie funkcje są wołane odpowiednio przy wejściu oraz przy wyjściu z funkcji *dbprinf()*, co powoduje iż w danym czasie tylko jeden watek może pisać do portu szeregowego.

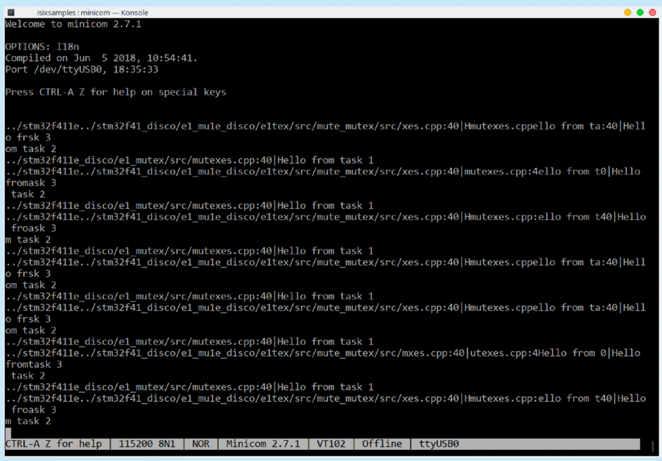
Aby zademonstrować działanie mechanizmu MUTEX, w funkcji głównej main tworzona jest funkcja lambda, która w nieskończonej pętli wysyła komunikat *Hello from task* <NR>, gdzie <NR> jest numerem danego wątku. Następnie wspomniana funkcja lambda wykorzystana jest do utworzenia trzech wątków (*isix::thread_create_and_run)*, które wypisują jednocześnie ten sam komunikat.

Spróbujmy uruchomić zatem wspomniany przykład, wpisując polecenia **waf program**, a następnie wybierając opcję [2]. Po załadowaniu przykładu, jeśli do linii portu *PA2* podłączymy przejściówkę USART ↔ USB i włączymy program terminalowy, powinniśmy zobaczyć następujące komunikaty (**rysunek 8**) wypisywane przez poszczególne wątki.

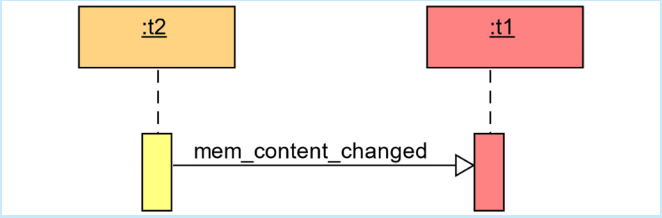
Jak możemy zauważyć, dzięki zastosowaniu mechanizmu MUTEX, pomimo że wszystkie wątki piszą do tego samego portu szeregowego, jednocześnie wszystkie komunikaty są wyświetlane poprawnie. Spróbujmy teraz wyłączyć mechanizm wzajemnego wykluczenia, wprowadzając komentarze w liniach 21 oraz 22 tak jak poniżej:



Rysunek 8.



Rysunek 9.



Rysunek 10.

```
[]() { /*m_mtx.lock(); */ },
[]() { /*m_mtx.unlock(); */},
```

Następnie spróbujmy skompilować ponownie program (polecenie **waf**) oraz zaprogramować płytkę ewaluacyjną (**waf program**, opcja [2]). Po załadowaniu programu na ekranie terminalu ujrzymy efekt działania programu pokazany na **rysunku 9**.

Jak możemy łatwo zauważyć, komunikaty wysłane przez poszczególne wątki uległy wzajemnemu pomieszaniu, ponieważ po wyłączeniu mechanizmu mutex wszystkie trzy wątki piszą dane do portu szeregowego w tym samym czasie.

Przykład 3. Komunikacja z wykorzystaniem wspólnej pamięci oraz semaforów

W poprzednich przykładach nauczyliśmy się tworzyć wątki oraz chronić dostęp do zasobów. Do omówienia pozostały nam jeszcze sposoby synchronizacji i przesyłania wiadomości pomiędzy procesami/wątkami. Najprostszym sposobem na przekazywanie wiadomości pomiędzy dwoma procesami jest wykorzystanie pamięci dzielonej, do której dostęp mają wątki komunikujące się ze sobą. W przypadku systemów z pamięcią wirtualną pamięć dzielona musi być utworzona za pomocą specjalnego wywołania systemowego, natomiast w przypadku systemów przeznaczonych dla układów pozbawionych MMU cała dostępna pamięć jest wspólna i widoczna zarówno dla procesów, jak i jądra. Przekazywanie danych za pomocą pamięci współdzielonej wymaga dodatkowego mechanizmu informującego o tym, że dane są gotowe do odczytania. W przypadku braku takiego mechanizmu wątek oczekujący na dane musiałby cyklicznie sprawdzać czy nowe dane nie nadeszły, co w systemie wielowątkowym byłoby dużym marnotrawstwem mocy obliczeniowej jednostki centralnej. Najprostszym sposobem na synchronizację danych jest wykorzystanie do tego celu semaforów binarnych. Wątek oczekujący na dane czeka na semaforze, natomiast wątek, który przekazuje dane po wypełnieniu pamięci dzielonej danymi podnosi semafor. Wątek oczekujący w wyniku podniesienia semafora zostaje wybudzony i może w tym czasie odczytać z pamięci dzielonej dane przekazane z drugiego wątku. Demonstrację wykorzystania semaforów i pamięci dzielonej przedstawimy na prostym przykładzie dwóch wątków, gdzie pierwszy wątek **t2** będzie odczytywał stan klawisza USER i w wyniku wciśnięcia będzie zmieniał stan zmiennej typu *bool* na przeciwny. Natomiast drugi wątek **t1** będzie odczytywał stan zmiennej i na tej podstawie włączał lub wyłączał diodę LED LD3 (**rysunek 10**).

Wynikiem działania powyższego przykładu będzie zmiana stanu diody LED na przeciwny w wyniku wciśnięcia przycisku USER zestawu ewaluacyjnego. Program rozpoczyna działanie od funkcji *main* (**listing 5**).

Na początku jest tworzony semafor, którego wartość początkowa została ustalona na 0, natomiast wartość maksymalna na 1, co oznacza, że mamy do czynienia z semaforem binarnym. Następnie tworzymy zmienną współdzieloną *val* oraz tworzymy dwa wątki: **t1**, odpowiedzialny za sterowanie diody LED LD3 oraz **t2** odpowiedzialny za odczytywanie stanu klawiatury. Do obu wątków jako argumenty przekazujemy referencję do zmiennej współdzielonej oraz referencję do wcześniej utworzonego semafora. Kod wątku odpowiedzialny za sterowanie diodą LED pokazano na **listingu 6**.

Listing 5.
```
// Start main function
auto main() -> int
{
    // Create the samaphore and val
    static isix::semaphore sem { 0, 1 };
    static bool val {};
    // Create task for blinking
    static auto t1 = isix::thread_create_and_run(1024,
        isix::get_min_priority(),0,blink_
task,std::ref(sem),std::ref(val));
    // Create task 2
    static auto t2 = isix::thread_create_and_run(1024,
        isix::get_min_priority(),0,keyb_scan_
task,std::ref(sem),std::ref(val));
    isix::start_scheduler();
    return 0;
}
```

Listing 6.
```
    auto blink_task(isix::semaphore& sem,bool& state) -> void {

        //! Blinking loop
        for(;;) {
            // Wait for the semaphore read the variable
            sem.wait(ISIX_TIME_INFINITE);
            periph::gpio::set(led_0,state);
        }
    }
}
```

Argumenty przekazane do funkcji wątku to referencje do semafora oraz zmiennej współdzielonej typu *bool*. Kod funkcji składa się z pętli nieskończonej, w której wywoływana jest metoda *wait()* na rzecz semafora, powodując uśpienie procesu. Jeśli drugi wątek zasygnalizuje gotowość do odebrania danych poprzez podniesienie semafora, działanie wątku zostanie wznowione, w wyniku czego odczyta on dane ze zmiennej *state* i ustawi stan diody LED zgodnie ze stanem tej zmiennej. Kod funkcji odpowiedzialny za odczyt stanu klawiatury realizowany jest przez funkcję *keyb_scan_task* (**listing 7**).

Funkcja stanowiąca wątek podobnie jak poprzednio przyjmuje referencję do tego samego semafora oraz zmiennej typu *bool*. W pętli nieskończonej cyklicznie odczytujemy stan klawisza USER (PA0) co 0,01 sekundy. Jeśli zostanie wykryte zbocze narastające, stan zmiennej współdzielonej jest zamieniany na przeciwny oraz wywoływana jest metoda *signal()* na rzecz semafora informująca o nadejściu nowych danych, co spowoduje wybudzenie poprzednio opisanego wątku czekającego na semaforze w metodzie *wait()*. Semafory systemu ISIX dopuszczają również sygnalizację semaforów z kontekstu przerwania (metoda *signal_isr*), zatem mogą być również wykorzystane do komunikacji pomiędzy procedurami obsługi przerwań a wątkami.

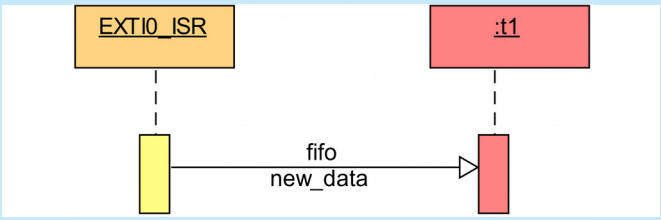
Pełny kod źródłowy przykładu dostępny jest w pliku: `stm32f411e_disco/e3_semaphore/src/semaphore.cpp`.

Aby sprawdzić działanie przykładu, należy wydać polecenie **waf program**, wybierając opcję [4]. Teraz wciskając przycisk USER, możemy zaobserwować, jak po każdym wciśnięciu przycisku dioda LD3 zmienia stan na przeciwny.

Przykład 4. Przekazywanie danych za pomocą kolejek FIFO

System ISIX zapewnia wygodniejszy sposób na przekazywanie danych pomiędzy wątkami lub wątkami a przerwaniami niż

Listing 7.
```
    // Task for button scanning and semaphore signaling
    auto keyb_scan_task(isix::semaphore& sem,bool& state) -> void {
        // Suspend resume flag
        for(auto pstate=false;;) {
            //Get key
            const auto val = periph::gpio::get(key_0);
            // On rising edge change state
            if(val && !pstate) {
                state = !state;
                // Semaphore signal
                sem.signal();
            }
            //Save previous state
            pstate = val;
            isix::wait_ms(10);
        }
    }
```



Rysunek 11.

mechanizm przedstawiony poprzednio. Do przekazywania danych możemy wykorzystać mechanizm w postaci kolejek FIFO. Kolejkę FIFO możemy stworzyć, korzystając z klasy *isix::fifo<T>*, gdzie w konstruktorze należy podać maksymalną liczbę elementów jaką kolejka może przechowywać, natomiast jako parametr wzorca **T** należy podać typ elementów, z jakich będzie się składać kolejka. Kolejka zapewnia automatyczne uśpienie procesu odczytującego, gdy nie ma w niej żadnych danych oraz uśpienie procesu zapisującego, gdy kolejka jest pełna i nie ma miejsca na nowe dane. Aby nieco urozmaicić przykłady, pokażemy teraz w jaki sposób przekazywać dane pomiędzy watkami a przerwaniami z wykorzystaniem mechanizmu kolejek. Działanie powyższego przykładu z punktu widzenia użytkownika będzie takie same jak poprzednio, jednak do odczytywania stanu klawisza USER wykorzystamy przerwania zgłaszane przez kontroler EXTI linia #0 (PA0). Procedura obsługi przerwania będzie wpisywać dane do kolejki FIFO, natomiast jedyny wątek w tym przykładzie będzie odczytywał dane z kolejki oraz odpowiednio ustawiał stan diody LD3 (**rysunek 11**).

Za inicjalizację systemu przerwań oraz konfigurację kontrolera EXTI odpowiedzialna jest funkcja *interrupt_input_config* (**listing 8**).

Na początku włączany jest sygnał zegarowy dla kontrolera EXTI, który znajduje się w domenie SYSCFG. Następnie do linii #0 kontrolera EXTI podłączany jest port A oraz włączane jest przerwanie z kanału #0. W kolejnych krokach kanał #0 ustawiany jest tak, aby reagował na zbocze narastające. Ostatnią czynnością jest odblokowanie przerwania od linii EXTI0 w kontrolerze NVIC oraz ustawienie priorytetu przerwania EXTI0 na najniższy możliwy. Od tego momentu każde wciśnięcie przycisku USER spowoduje wywołanie procedury obsługi przerwania (**listing 9**).

Procedura ta na początku zeruje flagę kanału #0 w kontrolerze EXTI, potwierdzając przyjęcie przerwania, a następnie sprawdza czas, jaki upłynął od przyjęcia ostatniego przerwania. Jeśli czas ten jest krótszy niż 10 ms, wciśnięcie klawisza jest ignorowane, co pozwala na eliminację drgań zestyków. Natomiast jeśli od poprzedniego wciśnięcia klawisza minęło więcej niż 10 ms, wówczas stan zmiennej state zmieniany jest na przeciwny i do kolejki za pomocą metody *push_isr* wpisywany jest aktualny stan tej zmiennej.

Za odczyt kolejki FIFO odpowiedzialny jest wątek **t1**, którego kod przedstawiono na **listingu 10**.

Funkcja wątku jako argument przyjmuje referencję do kolejki FIFO, przekazanej podczas jego utworzenia. W pętli nieskończonej wywoływana jest metoda *pop()* na rzecz kolejki FIFO. W przypadku, gdy są dostępne dane do odczytu, metoda zdejmuje daną z kolejki i przekazuje ją do zmiennej *state*. W przypadku, gdy kolejka jest pusta, metoda usypia wątek, do czasu gdy nadejdą nowe dane. Po odczytaniu danych z kolejki ustawiamy odpowiednio diodę LD3, stosownie do stanu elementu zdjętego z kolejki.

Kod źródłowy dotyczący tego przykładu możemy odnaleźć w pliku: `stm32f411e_disco/e4_fifo_from_irq/src/fifoirq.cpp`. Aby przetestować działanie powyższego przykładu, należy wydać polecenie **waf program**, a następnie wybrać opcję [5].

Przykład 5. Zaawansowana synchronizacja za pomocą mechanizmu zdarzeń bitowych (events)

W prezentowanych do tej pory programach przedstawiliśmy jedynie proste przykłady, gdzie jeden wątek przesyłał dane, natomiast drugi oczekiwał na nie. W danym momencie wątek mógł czekać tylko na jednym elemencie synchronizacyjnym. W wielu bardziej zaawansowanych przypadkach istnieje konieczność równoczesnego oczekiwania na kilka zdarzeń jednocześnie lub na wystąpienie odpowiedniej kombinacji zdarzeń pochodzących od kilku wątków. W takim przypadku z pomocą przychodzą nam zdarzenia bitowe *isix::events*. Zdarzenia bitowe szczegółowo zostały omówione w pierwszej części kursu. Dla przypomnienia podam jedynie, że zdarzenia zawierają 31-bitową maskę bitową, gdzie każdy wątek może czekać na ustawienie wszystkich wymaganych bitów lub jeden z wybranych. Umożliwia to realizację wielu skomplikowanych przypadków synchronizacyjnych, które w systemach zgodnych ze standardem POSIX realizowane są zazwyczaj z wykorzystaniem wywołań systemowych *poll* oraz *epoll*. Typowym przykładem zastosowaniem tego mechanizmu jest oczekiwanie w jednym wątku na dane z kilku kanałów komunikacyjnych, np. gdy wątek musi oczekiwać na nadejście danych z portu szeregowego i klawiatury. Wystąpienie dowolnego z wybranych warunków powinno powodować wybudzenie oczekującego wątku.

Działanie kolejnego przykładu będzie nieco bardziej skomplikowane. Utworzymy 5 wątków scantsk[0..3] odpowiedzialnych za

Listing 8.
```
auto interrupt_input_config() -> void {
        //Configure EXTI controller
        LL_APB2_GRP1_EnableClock(LL_APB2_GRP1_PERIPH_SYSCFG);
        LL_SYSCFG_SetEXTISource(LL_SYSCFG_EXTI_PORTA, LL_SYSCFG_EXTI_LINE0);
        LL_EXTI_EnableIT_0_31(LL_EXTI_LINE_0);
        LL_EXTI_EnableRisingTrig_0_31(LL_EXTI_LINE_0);
        //Nvic enable irq
        isix::set_irq_priority(EXTI0_IRQn, {1, 7});
        isix::request_irq(EXTI0_IRQn);
}
```

Listing 9.
```
void exti0_isr_vector() {
        //Debounce time and debounce timer
        static constexpr auto debounce_ms {10};
        static ostick_t last_irq_tick {};
        //Current state on or off
        static bool state {};
        //If exti0 clear flag and check usec timer
        if(LL_EXTI_IsActiveFlag_0_31(LL_EXTI_LINE_0)) {
                LL_EXTI_ClearFlag_0_31(LL_EXTI_LINE_0);
                //If debounce time  change state
                if(isix::timer_elapsed(last_irq_tick,debounce_ms)) {
                        state = !state;
                        fifo.push_isr(state);
                }
                last_irq_tick = isix::get_jiffies();
        }
}
```

Listing 10.
```
auto blink_task(isix::fifo<bool>& fifo) -> void {
        //! Blinking loop
        for(;;) {
                // Wait for the semaphore read the variable
                bool state {};
                fifo.pop(state,ISIX_TIME_INFINITE);
                periph::gpio::set(led_0,state);
        }
}
```

Tabela 1.

Wątek/ Klawisz	Ustawia bit	Wątek/Dioda	Czeka na bity
K0 (PC0)	0	LED0 (PD0)	0
K1 (PC1)	1	LED1 (PD1)	1
K2 (PC2)	2	LED2 (PD2)	2
K3 (PC3)	3	LED3 (PD3)	3 oraz 4
K4 (PC4)	4		

sprawdzanie stanu klawiszy **K0-K4** (każdy klawisz osobny wątek) oraz 4 wątki ledtasks[0..3], odpowiedzialne za sterowanie diodami LED **D0-D3** (każda dioda to osobny wątek). Wciskanie klawiszy **K0-K2** spowoduje odpowiednio naprzemienne włączanie diod **D0-D2**, natomiast do zmiany stanu diody **D3** wymagane będzie wciśnięcie w dowolnej kolejności klawiszy **K3** oraz **K4.** Zadanie można zrealizować za pomocą odpowiedniej liczby semaforów, jednak dużo wygodniejsze w tym przypadku będzie skorzystanie z jednego obiektu zdarzeń bitowych i*six::events*. Każdy wątek odpowiedzialny za poszczególne klawisze w wyniku jego wciśnięcia ustawiał będzie odpowiedni bit w polu bitowym *isix::events*. Natomiast wątki odpowiedzialne za sterowanie diodami LED będą oczekiwały na ustawienie odpowiednich bitów lub w przypadku diody **D3** kombinacji bitów zgodnie z **tabelą 1**.

Za odczyt stanu przycisków poprzez poszczególne wątki scantsk[0..3] odpowiada funkcja *key_scan_task* (**listing 11**).

Jako argumenty funkcja przyjmuje referencję do obiektu zdarzeń bitowych oraz numer porządkowy przycisku stanowiący także numer bitu w zdarzeniu bitowym, który funkcja będzie ustawiać. Po wykryciu zbocza opadającego na wybranej linii wywoływana jest metoda *set()* z klasy event, która ustawia wybrany bit w polu bitowym.

Za sterowanie pracą poszczególnych diod LED odpowiada funkcja *blink_task*, która jest wykorzystana przez 4 wątki ledtasks[0..3], odpowiedzialne za sterowanie diodami LED (**listing 12**).

Funkcja jako argumenty (przekazane przy tworzeniu wątków) przyjmuje odpowiednio referencję do obiektu zdarzeń bitowych, numer porządkowy diody LED, którą ma sterować oraz maskę bitową, na które funkcja ma czekać. W pętli nieskończonej wywoływana jest metoda *wait()* z klasy event, do której jako pierwszy argument przekazujemy bity, na które funkcja ma czekać. Kolejny argument, przyjmujący wartość **true**, określa, że funkcja ma czekać na kombinację wszystkich bitów. Kolejny argument *true* określa, że po wybudzeniu w wyniku reakcji na zdarzenie oraz odczytaniu maski bitowej ustawione bity zostaną automatycznie wyzerowane. Po wywołaniu metody wątek zostanie uśpiony aż do momentu, gdy wątki odpowiedzialne za obsługę klawiatury ustawią wymaganą kombinację bitów. Ustawienie odpowiedniej kombinacji bitów w zdarzeniu bitowym powoduje wybudzenie wątku oraz zmianę stanu wybranej diody LED na przeciwny.

Za tworzenie wątków oraz przekazanie do nich odpowiednich argumentów odpowiedzialna jest funkcja main (**listing 13**).

Na początku tworzony jest obiekt zdarzeń bitowych ***ev***. Następnie tworzone są 4 wątki odpowiedzialne za obsługę diod LED, gdzie jako argumenty przekazane do wątku zostaną odpowiednio: referencja do obiektu zdarzeń bitowych *ev*, numer porządkowy diody LED, którą dany wątek ma kontrolować oraz maskę bitową, na którą wątek ma czekać. Możemy zauważyć, że ostatni wątek odpowiedzialny za czwarta diodę **LED3** ma ustawione bity oczekiwania 3 i 4, a zatem aby dioda zmieniła stan, wymagane będzie wciśnięcie dwóch przycisków **K3** i **K4**. Na zakończenie tworzone jest 5 wątków odpowiedzialnych za odczyt stanu przycisków. Jako argumenty przekazane do wątku zostaną: referencja do zdarzenia bitowego **ev** oraz numer porządkowy przycisku, który dany wątek odczutuje.

Kod źródłowy przykładu możemy znaleźć w pliku: `stm32f411e_disco/e5_events/src/events.cpp`.

Aby uruchomić powyższy przykład, do linii portów **PD0...PD3** należy za pomocą przewodów połączeniowych dołączyć minimoduł *KamodLED8*, natomiast do linii portów **PC0...PC4** minimoduł *WSR-04499*. Zaprogramowanie płytki ewaluacyjnej odbywa się standardowo za pomocą polecenia **waf program**, przy czym do zaprogramowania należy wybrać opcję [6].

Lucjan Bryndza

Listing 11.
```
auto key_scan_task(isix::event& ev, int id) -> void {
        // Suspend resume flag
        for(auto pstate=true;;) {
                //Get key
                const auto val = periph::gpio::get(keys[id]);
                // On rising edge change state
                if(!val && pstate) {
                        ev.set( 1U<<id );
                }
                //Save previous state
                pstate = val;
                isix::wait_ms(10);
        }
}
```

Listing 12.
```
auto blink_task(isix::event& ev, int led, unsigned wait_for ) -> void {
        //! Blinking loop
        for(;;) {
                ev.wait(wait_for, true, true);
                periph::gpio::toggle(leds[led]);
        }
}
```

Listing 13.
```
auto main() -> int
{
        io_config();
        //Event
        static isix::event ev {};
        // Create 4 tasks for led controlling
        static isix::thread ledtasks[] = {
                isix::thread_create_and_run(ISIX_MIN_STACK_SIZE,
                        isix::get_min_priority(),0,blink_task, std::ref(ev), 0, 0x01 ),
                isix::thread_create_and_run(ISIX_MIN_STACK_SIZE,
                        isix::get_min_priority(),0,blink_task, std::ref(ev), 1, 0x02 ),
                isix::thread_create_and_run(ISIX_MIN_STACK_SIZE,
                        isix::get_min_priority(),0,blink_task, std::ref(ev), 2, 0x04 ),
                isix::thread_create_and_run(ISIX_MIN_STACK_SIZE,
                        isix::get_min_priority(),0,blink_task, std::ref(ev), 3, 0x18 ),
        };
        //! Create 5 tasks for keyboard scanning
        static isix::thread scantsk[] = {
                isix::thread_create_and_run(ISIX_MIN_STACK_SIZE,
                        isix::get_min_priority(),0,key_scan_task,std::ref(ev), 0),
                isix::thread_create_and_run(ISIX_MIN_STACK_SIZE,
                        isix::get_min_priority(),0,key_scan_task,std::ref(ev), 1),
                isix::thread_create_and_run(ISIX_MIN_STACK_SIZE,
                        isix::get_min_priority(),0,key_scan_task,std::ref(ev), 2),
                isix::thread_create_and_run(ISIX_MIN_STACK_SIZE,
                        isix::get_min_priority(),0,key_scan_task,std::ref(ev), 3),
                isix::thread_create_and_run(ISIX_MIN_STACK_SIZE,
                        isix::get_min_priority(),0,key_scan_task,std::ref(ev), 4),
        };
        isix::start_scheduler();
}
```


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