Laboratorium 5 MARM Systemy operacyjne czasu rzeczywistego. ISIXRTOS

Lucjan Bryndza Politechnika Warszawska

12 grudnia 2019

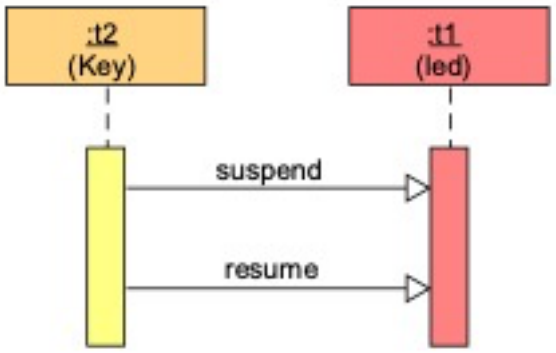
Celem laboratorium jest zapoznanie się systemami czasu rzeczywistego stosowanymi w systemach wbudowanych na przykładzie systemu **ISIXRTOS** oraz mikrokontrolerów STM32. Uczestnik laboratorium zapozna się działaniem systemów operacyjnych z mikrokontrolerami jednoukładowymi, oraz tym w jaki sposób mogą uprościć pisanie oprogramowania wbudowanego. Uczestnik zdobędzie wiedzę związaną z tworzeniem zadań, mechanizmami synchronizacji międzyprocesowej, oraz komunikacji między funkcjami obsługi przerwań, a zadaniami.

1. Zadania do wykonania na ćwiczeniach

Wszystkie ćwiczenia związane laboratorium realizować będziemy z wykorzystaniem płytki ewaluacyjnej **STM32F411E-DISCOVERY**.Po wykonaniu ćwiczenia należy opracować sprawozdanie, którego sposób przygotowania opisano w rozdziale 3.

1.1. Tworzenie wybudzanie oraz usypianie wątków

Utwórz dwa wątki: Wątek **t1**, którego zadaniem będzie błyskanie z częstotliwością 2Hz diodą *LD3*, oraz wątek **t2**, który po każdym wciśnięciu przycisku USER, będzie naprzemiennie usypiał oraz wybudzał watek **t1**



Do tego celu przy tworzeniu wątku za pomocą funkcji *isix::thread_create_and_run* przekaż flagę *isix_task_flag_suspended*, co spowoduje, że nowo utworzony utworzony wątek będzie od razu uśpiony. Sprawdź działanie i poprawność programu.

1.2. Komunikacja z wykorzystaniem wspólnej pamięci oraz semaforów

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 musiał by cyklicznie sprawdzać czy nowe dane nie nadeszły, co w systemie wielowątkowym było by dużym marnotrawstwem mocy obliczeniowej jednostki centralnej. Najprostszym sposobem na synchronizację danych jest wykorzystanie do tego celu semaforów

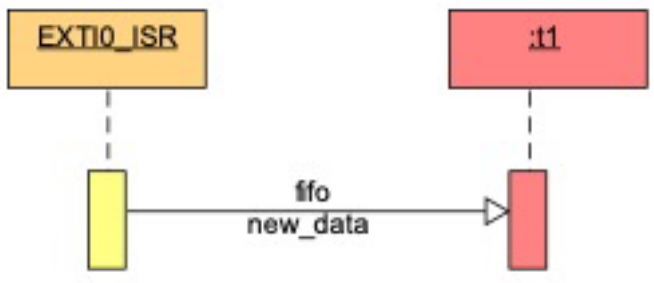
binarnych. Wątek oczekujący na dane, czeka na semaforze, natomiast wątek który przekazuje dane po wypełnieniu pamięci dzielonej danymi podnosi semafor. Wątek oczekujący w wyniku podniesienia semafora zostaje wybudzony i może w tym czasie odczytać z pamięci dzielonej dane przekazane z drugiego wątku.

Napisz program tworzący dwa wątki, 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**.

1.3. Komunikacja z wykorzystaniem kolejek FIFO

System ISIX zapewnia wygodniejszy sposób na przekazywanie danych pomiędzy wątkami lub wątkami, a przerwaniami niż przedstawiony w poprzednim podpunkcie. Do przekazywania danych możemy wykorzystać dedykowany mechanizm w postaci kolejek FIFO. Kolejkę fifo możemy stworzyć korzystając z klasy **isix::fifo<T>**, gdzie w konstruktorze należy podać maksymalną ilość 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.

Napisz program, który po każdym wciśnięciu klawisza **USER** *(PA0)*, generował będzie przerwanie. Procedura obsługi przerwania powinna wpisywać do kolejki FIFO aktualny stan klawisza. Utwórz wątek **t1**, który będzie odczytywał dane z kolejki FIFO i wizualizował stan wciśnięcia klawisza.



W programie należy zainicjalizować kontroler przerwań tak, tak aby każde wciśnięcie przycisku zgłaszało przerwanie od kontrolera EXTI *linia 0* (**PA0**).

Aby program działał poprawnie należy używać metod i funkcji z przyrostkiem **_isr**, które są przeznaczone do wywoływania z procedur obsługi przerwań.

1.4. Zaawansowana komunikacja z wykorzystaniem mechanizmu zdarzeń bitowych (events)

W zrealizowanych programach zaprezentowano 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 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 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 np. z portu szeregowego i klawiatury. Wystąpienie dowolnego z wybranych warunków powinno powodować wybudzenie oczekującego wątku.

Napisz program który:

- Utworzy obiekt zdarzeń bitowych
- Utworzy 5 oddzielnych wątków odpowiedzialnych za sprawdzanie stanu linii **PC0** ... **PC4**, umownie nazwanych dalej klawiszami **K0**...**K3**. Każda linia osobny wątek
- Utworzy 4 oddzielne wątki odpowiedzialne za sterowanie diodami LED **LD3**...**LD6**
- Każde wciśnięcie klawiszy **K0**-**K2** spowoduje odpowiednio naprzemienne włączanie i wyłączanie diod **LD3**-**LD5**
- Do zmiany stanu diody **LD6**, wymagane będzie wciśnięcie w dowolnej kolejności klawiszy **K3** oraz **K4**

Zadanie można zrealizować za pomocą odpowiedniej ilości semaforów, jednak dużo wygodniejsze w tym przypadku będzie skorzystanie z jednego obiektu zdarzeń bitowych **isix::events**.

Działanie programu reprezentuje poniższa tabela:

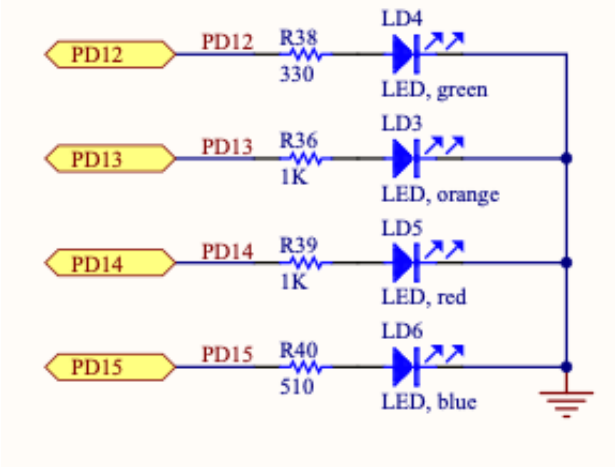
Wątek/klawisz	Ustawia bit	Wątek/Dioda	Czeka na bity
K0 (PC0)	0	LD3 (PD13)	0
K1 (PC1)	1	LD4 (PD12)	1
K2 (PC2)	2	LD5 (PD14)	2
K3 (PC3)	3	LD6 (PD15)	3 oraz 4

Ponieważ w zestawie ewaluacyjnym jest tylko jeden przycisk, do realizacji ćwiczenia możemy wykorzystać zestawy przewodów i zwierać odpowiednie linie do masy (GND). Należy pamiętać aby ustawić rezystory podciągające w górę przy konfiguracji portów wejściowych.

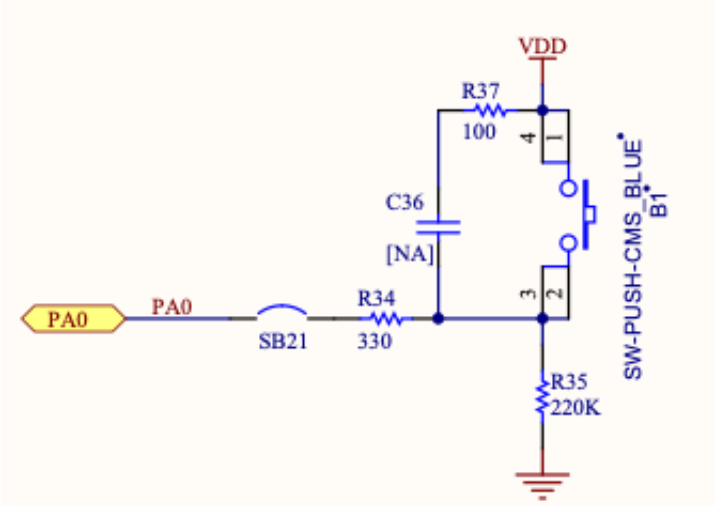
2. Materiały pomocnicze

2.1. Diody LED w zestawie STM32F411E-DISCO

Zestaw ewaluacyjny posiada 4 diody LED podłączone do portu **PD** do pinów 12...15.



Jeśli chodzi o przyciski wejściowe do dyspozycji mamy tylko jeden przycisk o nazwie **USER** dołączony do portu **PA0**.



2.2. Konfiguracja pobranie oraz uruchomienie projektu

Aby pobrać repozytorium możemy się posłużyć komendą, którą należy uruchomić w wierszu polecenia *GIT Bash*:

```
git clone --recursive -b pub/pw/lab5 https://boff.pl/cgit/
  ↪ public/isixsamples/
```

W kolejnym kroku należy zmienić katalog na *isixsamples* oraz pobrać konfigurację dla środowiska *VSCode*:

```
cd isixsamples
curl http://bryndza.boff.pl/downloads/prv/vscodecfg.zip --
  ↪ output vscodecfg.zip
```

Pobrany plik należy rozpakować bezpośrednio do katalogu *isixsamples*.

```
unzip vscodecfg.zip
```

Konfiguracja projektu dla zestawu STM32F411 discovery odbywa się za pomocą polecenia:

```
python waf configure --debug --board=stm32f411e_disco
```

Kompilacje projektu możemy wykonać za pomocą polecenia:

```
python waf
```

Natomiast zaprogramowanie zestawu odbywa się za pomocą polecenia:

```
python waf program
```

Projekt możemy również konfigurować oraz uruchamiać bezpośrednio z *Visual Studio Code*. Najpierw należy w pliku *.vscode/task.json* zmienić argumenty dla polecenia *waf configure*. Następnie za pomocą polecenia **CTRL+P** otwieramy wiersz polecenia i wpisujemy *task waf configure* w kolejny kroku należy zbudować projekt za pomocą polecenia *task waf*, a w kolejnym kroku zaprogramować poleceniem *task waf program*.

Debugowanie programu następuje po wciśnięciu klawisza **F5** wcześniej jednak należy w pliku *.vscode/launch.json* ustawić odpowiednio ścieżkę do uruchamianego programu.

Opis skrótów klawiaturowych dla środowiska **VSCode** możemy znaleźć tutaj: [4].

2.3. Podstawowe API systemu ISIX do ćwiczenia

Podstawową funkcją diagnostyczną służącą do wypisywania danych na domyślny port diagnostyczny (terminal) jest funkcja:

```
#define dbprintf(fmt, ...) fnd::tiny_printf("%s:%d|" fmt "\r\n",
  ↪ _isix_dbglog_extract_basename(__FILE__),__LINE__,##
  ↪ __VA_ARGS__)
```

Funkcja przyjmuje identyczny zestaw argumentów jak standardowa funkcja **printf** pozbawiona jest jedynie formatowania i wyświetlania liczb zmiennoprzecinkowych.

Tworzenie wątków możliwe jest dzięki zastosowaniu funkcji:

```
/** \brief Create thread and run
 * \param[in] size Stack size
 * \param[in] priority Thread priority
 * \param[in] flags Thread flags
 * \param[in] fn Function executed in separate thread
 * \param[in] args Arguments passed to the function
 */
template <typename FN, typename ... ARGS>
       thread thread_create_and_run( const size_t size,
          ↪ const osprio_t priority, unsigned flags, FN&&
          ↪ fn, ARGS&&... args ) noexcept
```

Jako argumenty funkcja przyjmuje odpowiednio: rozmiar stosu, priorytet wątku, flagi startowe, funkcje o dowolnej sygnaturze, oraz argumenty przekazane do funkcji.

Parametry flags umożliwiają sterowanie zachowaniem nowo utworzonego wątku, przy czym najważniejsza jest flaga **isix__task__flag__suspended** która tworzy zadanie uśpione.

Funkcja zwraca uchwyt, obiekt klasy **isix::thread__base** która umożliwia kontrolowanie utworzonego wątku. Najważniejsze metody przedstawiono poniżej:

```
//! Get raw task handler
ostask_t tid() const noexcept;
//! Kill the selected thread
void kill() noexcept;
//! Change priority
int change_prio( osprio_t new_prio ) noexcept;
//! Get isix task priority
int get_prio() const noexcept;
//! Get task inherited priority
int get_inherited_prio() const noexcept;
//! Return thread free stack space
ssize_t free_stack_space() const noexcept;
//! Suspend the task
int suspend() noexcept;
//! Resume the task
int resume() const noexcept;
//! Get task scheduler state
int get_state() const noexcept;
```

```
        //! Wait for the task
        int wait_for() const noexcept;
```

Za realizację semafora w systemie ISIX odpowiada klasa **isix::semaphore**. Aby utworzyć semafor należy podać jego wartość początkową oraz wartość maksymalną jaką może osiągnąć, dzięki czemu możemy utworzyć semafor binarny lub semafor zliczający.

```
    //Semafor binarny
    isix::semaphore bin_semaphore {0, 1}
    //Semafor zliczajacy
    isix::semaphore couting_semaphore { 0 };
```

Gdy mamy utworzony semafor możemy, go kontrolować za pomocą następujących metod:

```
        /** Check the fifo object is in valid state
        * @return true if object is in valid state otherwise
          ↪  return false
        */
        bool is_valid() const noexcept;
        /** Wait for the semaphore for selected time
        * @param[in] timeout Max waiting time
        */
        int wait(ostick_t timeout) noexcept;
        /** Get the semaphore from the ISR context
        * @return ISIX_EOK if the operation is completed
          ↪ successfully otherwise return an error code
        */
        int trywait() const noexcept;
        /** Signaling the semaphore
        * @return ISIX_EOK if the operation is completed
          ↪ successfully otherwise return an error code
        */
        int signal() noexcept;
        /** Signal the semaphore from the ISR context
        * @return ISIX_EOK if the operation is completed
          ↪ successfully otherwise return an error code
        */
        int signal_isr() noexcept;
        /** Reset value of the semaphore (From interrupt
          ↪ context)
        * @param[in] val Value of the semaphore
        * @return ISIX_EOK if the operation is completed
          ↪ successfully otherwise return an error code
```

```
             */
             int reset_isr( int val ) noexcept;
             /** Reset value of the semaphore (From interrupt
                ↪ context)
             * @param[in] val Value of the semaphore
             * @return ISIX_EOK if the operation is completed
                ↪ successfully otherwise return an error code
             */
             int reset_isr( int val ) noexcept
```

Należy pamiętać, że jedynymi dozwolonymi metodami jakich możemy użyć z funkcji obsługi przerwania to metoda **trywait** oraz metody **signal_isr** oraz **reset_isr**.

Kolejki fifo umożliwiają przekazywanie danych pomiędzy zadaniami, oraz pomiędzy zadaniami i przerwaniami. W systemie ISIX kolejkę fifo możemy utworzyć za pomocą klasy **isix::fifo**. Jest to klasa wzorcowa aby np utworzyć kolejkę mogącą przechować 20 elementów typu *int* należy użyć następującego kodu:

```
isix::fifo<int> int_queue { 20 };
```

Po utworzeniu kolejki fifo jej stan możemy kontrolować za pomocą następujących metod:

```
/** Check if the fifo object is in valid state
 * @return true if object is ok else return false
 */
bool is_valid() const;
 /* Push data in the queue
      * @param[in] c Reference to the object
      * @param[in] timeout Wait time when queue is not empty
      * @return ISIX_EOK if success else return an error
 */
 int push(const T &c,ostick_t timeout=0);
 /* Push data in the queue from the ISR
  * @param[in] c Reference to the object
  * @param[in] timeout Wait time when queue is not empty
  * @return ISIX_EOK if success else return an error
  */
 int push_isr(const T &c);
/** Get available elements in the fifo
  * @return available elements in the fifo
 */
 auto size() const;
    /** Pop the element from the queue
 * @param[in] c Reference to the buffer
```

```
     * @param[in] timeout Max waiting time
     * @return ISIX_EOK if success else return an error
     */
    int pop(T &c, ostick_t timeout=0);
    /** Pop the element from the queue. Called from the ISR
     * @param[in] c Reference to the buffer
     * @param[in] timeout Max waiting time
     * @return ISIX_EOK if success else return an error
    */
    int pop_isr(T &c);
```

Należy pamiętać, że w procedurach obsługi przerwań możemy używać jedynie metod z sufiksem **__isr**.

Jeśli chcemy synchronizować wiele wątków jednocześnie możemy do tego celu wykorzystać mechanizm zdarzeń bitowych, za który odpowiada klasa **isix::event**. Utworzenie obiektu klasy event realizujemy za pomocą następującego kodu:

```
isix::event my_event_bits;
```

Po utworzeniu obiektu możemy go kontrolować za pomocą zestawu nastepujących metod:

```
/** Check if the object is in valid state
 * @return true if object is properly initialized
 */
bool is_valid() const;
/** Wait for event
 * @param[in] bits_to_set Bits to set in event
 * @param[in] bits_to_wait Bits to wait for
 * @param[in] timeout Isix timeout
 * @return Changed bitset or error code
 */
osbitset_ret_t sync( osbitset_t bits_to_set,
  ↪ osbitset_t bits_to_wait, ostick_t timeout =
  ↪ ISIX_TIME_INFINITE );
/** Wait for event bit set
* @param[in] bits_to_wait Bits to wait for
* @param[in] clear_on_exit Clear bits on exit
* @param[in] wait_for_all Wait for all bits
* @param[in] timeout  Timeout to wait for sync
* @return Bits which are set
*/
osbitset_ret_t wait( osbitset_t bits_to_wait, bool
  ↪ clear_on_exit,
```

```
                bool wait_for_all, ostick_t timeout =
                  ↪ ISIX_TIME_INFINITE );
        /** Clear the selected bits from the event
        * @param[in] bits_to_clear
        * @return Changed bits
        */
        osbitset_ret_t clear( osbitset_t bits_to_clear );
        osbitset_t clear_isr( osbitset_t bits_to_clear );
        /** Isix set bits
        * @param[in] bits_to_clear
        * @return Changed bits
        */
        osbitset_t set( osbitset_t bits_to_set );
        osbitset_t set_isr( osbitset_t bits_to_set );
        /** Get the events from the interrupt context
        * @return Bit state
        */
        osbitset_ret_t get_isr( );
        osbitset_ret_t get( );
```

Należy pamiętać, że w procedurach obsługi przerwań możemy używać jedynie metod z sufiksem **_isr**.

W systemie ISIX wektory przerwań mają inne nazwy niż w przypadku przykładów z biblioteki **LL**. W przypadku układów przerwań EXTI nazwy wektorów przerwań są następujące:

```
ISR_VECTOR(exti0_isr_vector);
ISR_VECTOR(exti1_isr_vector);
ISR_VECTOR(exti2_isr_vector);
ISR_VECTOR(exti3_isr_vector);
ISR_VECTOR(exti4_isr_vector);
//...
ISR_VECTOR(exti14_isr_vector);
ISR_VECTOR(exti15_isr_vector);
```

Należy pamiętać, że w przypadku deklaracji wektorów w języku C++ należy użyć konstrukcji **extern "C"**, a funkcje wektorów przerwań powinny być zadeklarowane jako **void fun(void)**.

3. Instrukcja opracowania sprawozdania

- W sprawozdaniu umieścić kod źródłowy programów
- Opisać zasadę działania napisanych programów oraz sposób ich realizacji

- Podczas pisania kodu należy umieścić funkcje diagnostyczne **dbprintf**, a w sprawozdaniu załączyć logi z konsoli szeregowej z uruchomionych programów
- Odpowiedzieć na pytanie: W jakich zastosowaniach użycie mechanizmu mutex jest lepsze niż zastosowanie semaforów?
- Odpowiedzieć na pytanie: Do czego potrzebujemy systemów operacyjnych czasu rzeczywistego w mikrokontrolerach?

Literatura

[1] *STM32F411E-DISCOVERY* kit user manual

[2] *STM32F411* reference manual

[3] *ISIX-RTOS* API examples

[4] *Visual Studio Code* keyboards shortcuts