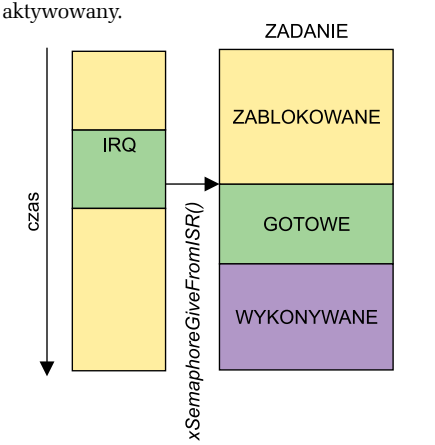
STM32
Free RTOS dla dociekliwych

W EP5/09 zostało przedstawione zagadnienie systemu operacyjnego FreeRTOS w odniesieniu do mikrokontrolerów STM32. Wykorzystując te informacje, w niniejszym artykule przedstawiono sposób tworzenia nieco bardziej zaawansowanych aplikacji z użyciem systemu FreeRTOS i mikrokontrolerów STM32. Pokazano m. in. jak nawiązać wymianę danych pomiędzy uruchomionymi w systemie zadaniami oraz jak zabezpieczyć zasoby mikrokontrolera przed nieuprawnionym dostępem.

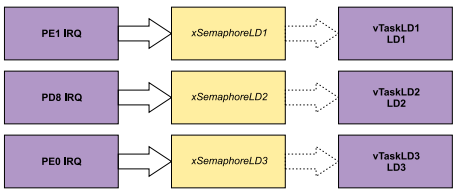
System operacyjny czasu rzeczywistego FreeRTOS udostępnia programistom w sumie pięć mechanizmów wykorzystywanych do komunikacji pomiędzy zadaniami (lub przerwaniami i zadaniami) oraz do zabezpieczania zasobów mikrokontrolera. Są to: semafory binarne, kolejki, semaforów licznikowe, muteksy, muteksy rekurencyjne. Każdy przedstawiony w artykule mechanizm został poparty stosownym przykładem, przygotowanym dla płytki ewaluacyjnej STM3210B-EVAL, która jest wyposażona w mikrokontroler **STM32F103**. Szczegółowych informacji na temat systemu operacyjnego FreeRTOS należy szukać na jego stronie internetowej www.freertos.org.

Semafory binarne

Semafory binarne, służą do sterowania wykonywaniem zadań. Gdy semafor jest nieaktywny, to wykonywanie czynności (zadania) jest zablokowane. Innymi słowy, aby zadanie, którego działanie jest uzależnione od semafora, mogło się wykonać, to semafor musi zostać aktywowany.



Rys. 1.



Rys. 2.

W praktyce systemów wbudowanych semafory binarne są zazwyczaj wykorzystywane od synchronizacji zadań lub zadania i przerwania. Zagadnienie synchronizacji zadania za pomocą przerwania przedstawiono na **rys. 1**. Dopóty, dopóki w systemie nie jest zarejestrowane przerwanie, zadanie pozostaje w stanie „*ZABLOKOWANE*". W chwili, gdy wystąpi przerwanie, a w funkcji jego obsługi nastąpi uaktywnienie semafora, system operacyjny wprowadzi zablokowane zadanie w stan „*GOTOWE DO WYKONANIA*". Od tego momentu zadanie oczekuje na zwolnienie zasobów, a jeśli to nastąpi, to zacznie być realizowane.

Aplikacja działająca w oparciu o identyczny mechanizm została omówiona poniżej, jej zadaniem jest reagowanie na zmiany stanów przycisków zapalaniem lub gaszeniem diod LED na płytce ewaluacyjnej. Sterowane mają być diody LD1, LD2, LD3, za pomocą położenia joysticka odpowiednio: lewo, góra, prawo. Schemat działania programu został przedstawiony na **rys. 2**.

Do komunikacji pomiędzy funkcjami obsługi przerwań i zadaniami wykorzystano trzy semafory binarne, dla każdego zestawu, przycisk i dioda, po jednym. Za stan każdej z diod odpowiada oddzielne zadanie, a więc sumie w systemie są uruchomione trzy zadania: *vTaskLD1()*, *vTaskLD2()*, *vTaskLD3()*. Kod zadania *vTaskLD1()* został zamieszczony na **list. 1**, natomiast funkcja obsługi przerwania dla lewego położenia joysticka znajduje się na **list. 2**. Pozostałe zadania i funkcje obsługi przerwań są różnią się tylko sterowanymi lub monitorowanymi wyprowadzeniami. Pozycja lewa joysticka jest podłączona do wyprowadzenia PE1, stąd wykorzystana jest funkcja obsługi przerwania *EXTI1_IRQHandler()*. Główna funkcja programu *main()*, która została zamieszczona na **list. 3**, ma za zadanie skonfigurować mikrokontroler wraz z wszystkimi wykorzystywanymi peryferiami do pracy – funkcja *prvSetupHardware()*. Ponadto następuje tutaj uruchomienie zadań oraz planisty, ten ostatni jest aktywowany przez wywołanie funkcji *vTaskStartScheduler()*.

Każdy semafor jest tworzony przed wejściem danego zadania do nieskończonej pętli. Zmienna *xSemaphoreLD1* jest zadeklarowana jako globalna, tak jak pozostałe semafory. Sprawdzenie stanu semafora odbywa się wraz z wywołaniem funkcji *xSemaphoreTake()*. W nieco ściślejszym rozumowaniu wymieniona funkcja próbuje „*wziąć*" semafor, co oznacza, że jeśli jest on ustawiony to następuje wykonanie dotychczas zablokowanej części zadania, a semafor zostaje dezaktywowany (skasowany).

Jeśli funkcja *xSemaphoreTake()* zwróci wartość *pdTRUE*, to wtenczas następuje zmiana stanu wyprowadzenia na przeciwny. Jako argumenty do funkcji należy przekazać nazwę semafora oraz (pośrednio) czas, przez jaki zadanie będzie oczekiwać, aż semafor stanie się aktywny. W omawianym przypadku wartość ta wynosi 0, ponieważ zadanie zajmuje się tylko sprawdzaniem stanu semafora i niczym więcej.

List. 1.
```
void vTaskLD1(void * pvParameters)
{
      vSemaphoreCreateBinary(xSemaphoreLD1);
      // Nieskonczona petla zadania
      for(;;)
      {
          if(xSemaphoreTake(xSemaphoreLD1, 0) == pdTRUE)
          {
              vhToggleLD1();
          }
      }
}
```

List. 2.
```
void EXTI1_IRQHandler(void)
{
    static portBASE_TYPE xHigherPriorityTaskWoken;
    xHigherPriorityTaskWoken = pdFALSE;
    if(EXTI_GetITStatus(EXTI_Line1) != RESET)
    {
       xSemaphoreGiveFromISR(xSemaphoreLD1,
       &xHigherPriorityTaskWoken);

          EXTI_ClearITPendingBit(EXTI_Line1);
    }
}
```

Głównym zadaniem mikrokontrolera w funkcji obsługi przerwania jest aktywowanie semafora, dzięki czemu zadanie będzie wiedziało, że należy zmienić stan wyprowadzenia, do którego podłączona jest dioda LED. Odpowiada za to funkcja *xSemaphoreGiveFromISR()*, której należy przekazać dwa argumenty, pierwszy to uchwyt (nazwa) semafora. Drugi, przekazywany przez referencję,

List. 3.
```
int main( void )
{
   // Konfiguracja sprzetu
   prvSetupHardware();

   // Uruchomienie zadan
   vStartLDTasks( TASK_PRIORITY );

   // Uruchomienie planisty
   vTaskStartScheduler();

   return 0;
}
```

argument jest zmienną, która otrzyma wartość *pdTRUE*, jeśli odblokowane przez semafor zadanie będzie miało wyższy priorytet, niż aktualnie wykonywane. Wszystkie nazwy funkcji API systemu FreeRTOS, jakie są używane podczas obsługi przerwań muszą kończyć się przyrostkiem „*ISR*". Jest to niezbędne dla poprawnej pracy mikrokontrolera.

Kolejki

Kolejki są głównym mechanizmem, jaki jest wykorzystywany do wymiany informacji pomiędzy zadaniami. Mogą być wykorzystywane do przesyłania wiadomości między zadaniami oraz pomiędzy przerwaniami i zadaniami. W większości przypadków kolejki są wykorzystywane jako bezpieczne bufory FIFO.

Każda zdefiniowana w systemie kolejka ma ustaloną długość, czyli liczbę elementów, jakie można do niej wpisać, oraz rozmiar pojedynczego elementu. Obydwa parametry są określane na etapie tworzenia (deklarowania) kolejki.

Elementy w kolejce są umieszczane jako kopie danych źródłowych, dzięki czemu komunikujące się zadania nie mogą bezpośrednio uzyskać dostępu do pamięci, w której znajdują się dane źródłowe. Mechanizm kolejek w systemie FreeRTOS ma zaimplementowaną obsługę wszystkich zagadnień związanych z wzajemnymi wykluczeniami, zatem programista nie musi już o to zabiegać.

Jeśli zaistnieje potrzeba kolejkowania danych o większym rozmiarze, niż to przewiduje zadeklarowana kolejka, to wtedy można użyć wskaźnika na element. W takich wypadkach należy się zawsze upewniać, kto (które zadanie) jest właścicielem danej zmiennej, oraz ostrożnie wykonywać operacje na otrzymanym adresie elementu tak, aby nie zdestabilizować pracy całego systemu.

Niekiedy może się zdarzyć, że w kolejce nie ma żadnych danych do odebrania przez określone zadanie. W takich sytuacjach mocy nabiera możliwość ustawienia czasu, a konkretniej liczby taktów zegara systemu operacyjnego, po jakim, jeśli żadne ważne dane nie pojawią się w kolejce, zadanie przejdzie do stanu „*ZABLOKOWANE*". Sytuacja może być również odwrotna: kolejka może być zapełniona, wtenczas zadanie, które chce wpisać dane do kolejki, oczekuje zadeklarowaną ilość taktów zegara na zwolnienie się miejsca w kolejce, gdy to nie nastąpi to również przechodzi do stanu „*ZABLOKOWANE*". Zasada działania kolejki została omówiona w EP5/09.

Sposób użycia kolejek zostanie przedstawiony na przykładzie aplikacji, której zadaniem będzie przetwarzanie A/C i pokazywanie wyniku na graficznym wyświetlaczu LCD zamontowanym na płytce ewaluacyjnej STM3210B-EVAL. Mierzone napięcie pochodzi od potencjometru podłączonego do wyprowadzenia PC4.

Konfiguracja ADC została dokładnie omówiona w EP, zatem tutaj nie będziemy się tym bliżej zajmować, podobnie jak w przypadku aplikacji demonstrującej działanie semaforów, również tutaj wszystkie czynności związane z konfiguracją są umieszczone w funkcji *prvSetupHardware()*.

W systemie są utworzone dwa zadania, natomiast jedyna kolejka – xQueueLCD – została utworzona jako zmienna globalna (uchwyt) typu *xQueueHandle*. Na **list. 4** został zamieszczony kod zadania *vTaskADC()*, które tworzy za pomocą wywołania funkcji *xQueueCreate()* kolejkę. Następnie już w pętli nieskończonej w 300 ms odstępach odczytuje wartość z przetwornika A/C, by w kolejnym kroku zapisać ją do kolejki. Do tego celu użyta jest funkcja *xQueueSend()*, której w argumentach należy podać kolejno: nazwę kolejki, zmienną do wysłania, oraz liczbę cykli systemowych, jakie będą odczekane w razie pełnej kolejki.

Drugie uruchomione w systemie zadanie – *vTaskLCD()* – jest przedstawione na **list. 5**. Pierwszą czynnością, jaką zadanie wykonuje, jest inicjalizacja wyświetlacza LCD, po czym w pętli nieskończonej, również co 300 ms, następuje odbieranie danych z kolejki i wyświetlanie ich na LCD.

Efektem pracy aplikacji jest pokazywanie wyniku przetwarzania, którym jest liczba z zakresu od 0 do 4096. Czas odświeżania został tak dobrany, aby można było dobrze zaobserwować efekt kolejkowania danych. Zmieniając dość szybko położenie potencjometru P1 widać, jak dopiero po chwili wynik osiąga swoją właściwą wartość.

List. 4.
```
void vTaskADC(void * pvParameters)
{
   u16 wynik_adc;
   xQueueLCD = xQueueCreate(10, sizeof(u16));

   // Nieskonczona petla zadania
   for(;;)
   {
      vTaskDelay(300 / portTICK_RATE_MS);  // Odczekanie 300 ms
      wynik_adc = ADC_GetConversionValue(ADC1);
      xQueueSend(xQueueLCD, (void *) &wynik_adc, (portTickType) 10);
   }
}
```

List. 5.
```
void vTaskLCD(void * pvParameters)
{
   u16 wynik;
   char wynik_lcd[5];
   STM3210B_LCD_Init();
   LCD_Clear(Blue);

   // Nieskonczona petla zadania
   for(;;)
   {
      xQueueReceive(xQueueLCD, &wynik, (portTickType) 10);
      vTaskDelay(300 / portTICK_RATE_MS);  // Odczekanie 300 ms
      sprintf(wynik_lcd, „%4d", wynik);
      LCD_DisplayStringLine(0, (u8*) wynik_lcd);
   }
}
```

List. 6.
```
void vTaskSemphr(void * pvParameters)
{
  u8 wynik_sem = 0;
  xQueueLCD = xQueueCreate(10, sizeof(xSemaphoreHandle));
  xSemaphoreCnt = xSemaphoreCreateCounting( 50, 0 );
  // Nieskonczona petla zadania
  for(;;)
  {
    vTaskDelay(1000  portTICK_RATE_MS);   //Odczekanie 1 sek
    xQueueSend(xQueueLCD, (void *) &wynik_sem, 0);
    if(xSemaphoreTake(xSemaphoreCnt,0) == pdPASS)
      wynik_sem = 1;
    else
      wynik_sem = 0;
  }
}
```

Semafory licznikowe

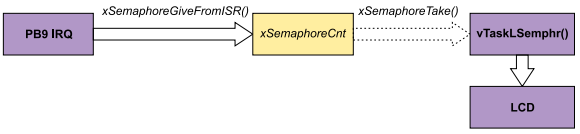
Semafory licznikowe (*Counting Semaphores*) są hybrydą zwykłej kolejki i semafora binarnego, zatem nie niosą ze sobą więcej informacji poza aktualną wartością semafora. Mechanizm semaforów licznikowych jest wykorzystywany przede wszystkim w implementacji zadań, które wymagają zliczania zdarzeń.

Od strony praktycznej wygląda to tak, że np. Zadanie A „daje" (*give*), czyli inkrementuje semafor licznikowy, natomiast Zadanie B „zabiera" (*take*) ten sam semafor – dekrementuje go. Tym sposobem wartość semafora licznikowego

List. 7.
```
void vTask25PWM(void * pvParameters)
{
   xSemaphoreMuteks = xSemaphoreCreateMutex();

   // Nieskonczona petla zadania
   for(;;)
   {
    if(xSemaphoreTake( xSemaphoreJoyUp, 0 ) == pdTRUE)
    {
      if(xSemaphoreTake(xSemaphoreMuteks, 0)  == pdTRUE)
      {
        vhSetPWM();
        vTaskDelay(500 / portTICK_RATE_MS);
        xSemaphoreGive(xSemaphoreMuteks);
      }
    }
   }
}

void vTask75PWM(void * pvParameters)
{
   // Nieskonczona petla zadania
   for(;;)
   {
    if(xSemaphoreTake( xSemaphoreJoyDown, 0 ) == pdTRUE)
    {
      if(xSemaphoreTake(xSemaphoreMuteks, 0)  == pdTRUE)
      {
        vhSetPWM();
        vTaskDelay(500 / portTICK_RATE_MS);
        xSemaphoreGive(xSemaphoreMuteks);
      }
    }
   }
}
```



Rys. 3.

wego określa różnicę w liczbie wystąpień zdarzenia i jego przetworzeń.

Sposób implementacji semaforów licznikowych prezentuje niżej omówiona przykładowa aplikacja. Jej zadaniem jest utworzenie w systemie semafora licznikowego, który ma być inkrementowany przez przerwanie pochodzące od wyprowadzenia PB9, do którego podłączony jest przycisk użytkownika. Proces dekrementowania semafora należy do uruchomionego w systemie zadania *vTaskSemphr()*. Zadanie w odstępach 1 sekundowych sprawdza stan semafora i jeśli nie jest zerowy to go dekrementuje. Dodatkowo aplikacja wykorzystuje omówione poprzednio kolejki do pokazywania na LCD wartości semafora. Schematycznie sposób pracy mikrokontrolera w tym przykładzie ilustruje **rys. 3**.

Semafor licznikowy jest identycznym typem zmiennej jak zwykły semafor binarny, a więc jego deklaracja jest taka sama jak w przypadku tego ostatniego. Dopiero podczas tworzenia semafora licznikowego należy użyć innej funkcji API, ściślej – *xSemaphoreCreateCounting()*. Funkcji tej należy przekazać dwa argumenty: pierwszy to maksymalna wartość semafora, a druga wartość początkowa. W omawianej aplikacji tworzenie semafora odbywa się w zadaniu *vTaskSemphr()*, przedstawionym na **list. 6**. W pętli nieskończonej odczekuje 1 sekundę, po czym sprawdza semafor licznikowy *xSemaphoreCnt* i wysyła do kolejki jego wartość. Zawartość kolejki jest odbierana przez zadanie *vTaskLCD()*, które zajmuje się obsługą wyświetlacza graficznego. Naciskając przycisk użytkownika np. 10 razy widzimy na LCD, że przez czas około 10 sekund semafor będzie jeszcze ustawiony, a dopiero po tym czasie nastąpi jego skasowanie.

Muteksy

Nazwa *„muteks”* jest określeniem angielskim i raczej nieprzetłumaczalnym na język polski. Słowo *„MUTEX”* powstało z połączenia wyrazów *„mutual”* oraz *„exclusion”*. Można, zatem mechanizm działania *muteksów* określić jako „wzajemne wykluczanie”, które dość dobrze oddaje istotę ich działania. *Muteksy* są nieco podobne do binarnych semaforów, wykorzystują te same funkcje API, lecz zostały wzbogacone o system priorytetów. Najistotniejsze jest to, że wykorzystanie semaforów i *muteksów* jest zupełnie różne. O ile semafory, jak już to zostało wyżej napisane, służą najczęściej do synchronizacji zadań, to *muteksy* zostały stworzone przede wszystkim z myślą o implementacjach, w których występuje dzielenie jakiegoś zasobu sprzętowego pomiędzy kilka zadań.

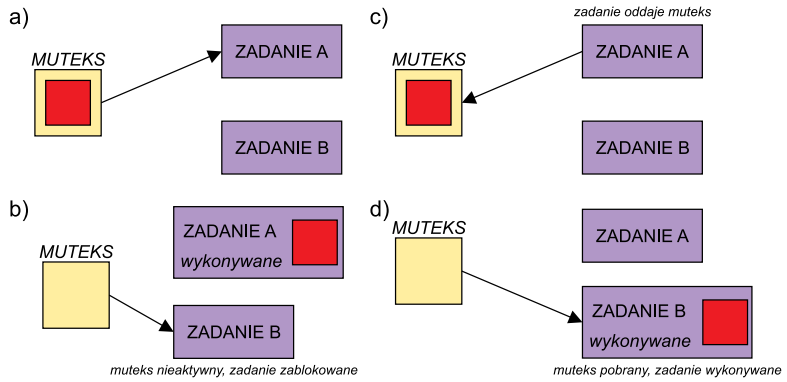
Zasada działania *muteksów* została wyjaśniona na **rys. 4**. Zadanie A, w chwili, gdy potrzebuje dostępu do chronionego zasobu, sprawdza stan *muteksa*, jeśli jest ustawiony, to wtenczas wiadomo, że zasób jest wolny i można z niego skorzystać. W trakcie wykorzystywania chronionego zasobu przez Zadanie A *muteks* jest *„pusty”*. Jeśli w takiej sytuacji Zadanie B podejmie próbę skorzystania z danego zasobu to z racji *„pustego” muteksa* dostęp do zasobu nie będzie możliwy. Dopiero po oddaniu *muteksa* przez Zadanie A, Zadanie B może wykorzystać do swoich celów chroniony zasób mikrokontrolera.

Przedstawimy teraz przykład aplikacji działającej z wykorzystaniem *muteksów* do ochrony zasobów. Załóżmy sytuację, w której dwa zadania, jeśli zaistnieje taka potrzeba, zmieniają wypełnienie generowanego przez mikrokontroler sygnału PWM. Nowowprowadzony współczynnik wypełnienia nie może się zmieniać przez czas 500 ms, a jeśli zostanie zmieniony to może to spowodować nieprawidłowe działanie całego systemu. Aby zabezpieczyć timer TIM3 pracujący w roli generatora PWM, przed nieuprawnionym dostępem zostanie wykorzystany muteks.

W systemie uruchomione są dwa zadania: *vTask25PWM()* oraz *vTask75PWM()*. Wychylnie joysticka na płytce ewaluacyjnej w górę powoduje odblokowanie pierwszego zadania i, jak nietrudno się domyślić, zmianę współczynnika wypełnienia sygnału PWM na 25%. Przeciwna pozycja joysticka (w dół) odblokowuje drugie zadania, a tym samym ustawia wypełnienie na 75%. Generator PWM – timer TIM3 – po pełnym przemapowaniu steruje wyprowadzeniem PC6, a więc diodą LD1. Efektem działania aplikacji jest zmiana intensywności świecenia diody w takt zmian położenia joysticka. Obecność w systemie pracującego *muteksa* można zaobserwować próbę zmian położenia joysticka z częstotliwością większą niż 1 Hz. *Muteks* chroniący zasób w postaci timera TIM3 nie pozwoli na częstsze zmiany intensywności świecenia diody LED niż co 500 ms.

Kod zadań *vTask25PWM()* i *vTask75PWM()* został zamieszczony na **list. 7**. Do synchronizacji z wyprowadzeniami mikrokontrolera wykorzystano omówione już wcześniej semafory binarne. *Muteks* jest tworzony w zadaniu *vTask25PWM()* za pomocą wywołania funkcji *xSemaphoreCreateMutex()*, jeszcze przed wejściem zadania do pętli nieskończonej. Sprawdzenie i próba zabrania *muteksa* odbywa się wraz z wywołaniem znanej już funkcji *xSemaphoreTake()*. Wywołanie to następuje tylko wtedy, kiedy semafor *xSemaphoreJoyUp* jest aktywny, co jest jednoznaczne z położeniem górnym joysticka. Jeśli *muteks* jest dostępny to wtenczas wypełnienie generowanego przebiegu zostanie ustawione na 25%, w przeciwnym wypadku współczynnik wypełnienia pozostanie niezmieniony.

Krzysztof Paprocki
paprocki.krzysztof@gmail.com



Rys. 4.