1.1 Mechanizmy synchronizacji

1.1.1 Semafory

1.1.1.1 Synchronizacja wątków

Wprowadzenie: Jednym z mechanizmów pozwalającym na synchronizację pracy wątków jest semafor. Można go porównać do flagi. Tak samo jak flaga semafor może posiadać dwa stany, zajęty albo wolny.

Do tworzenia semafora służy funkcja `vSemaphoreCreateBinary`. Funkcja zwraca uchwyty do stworzonego semafora. Poniżej znajduje się przykład tworzenia semafora:

```
xSemaphoreHandle xSemaphore;
vSemaphoreCreateBinary( xSemaphore );
```

Listing Błąd! W dokumencie nie ma tekstu o podanym stylu.**-1 Tworzenie semafora w systemie FreeRTOS.**

Podstawowe funkcje do pracy z semaforami to [2] :

- `xSemaphoreTake(xSemaphore,portMAX_DELAY)`, która jeżeli semafor jest:
  - *wolny* – wprowadza go w stan *zajęty* i zwraca `pdTRUE,`
  - *zajęty* – czeka na zwolnienie semafora przez czas określony w drugim argumencie (`portMAX_DELAY` oznacza nieskończoność) i:
    - jeżeli w tym czasie się zwolnił wprowadza go w stan *zajęty* i zwraca `pdTRUE,`
    - jeżeli w tym czasie się nie zwolnił zwraca `pdFALSE,`
- `xSemaphoreGive(xSemaphore)`, która wprowadza semafor w stan *wolny.*

Zadanie 1: Napisać program składający się z dwóch wątków.

Pierwszy wątek (*Pulse*) powinien po każdym zwolnieniu semafora (`xSemaphoreTake`) zaświecić a następnie zgasić diodę na 0.1s.

Drugi wątek (*PulseTrigger*) powinien co sekundę wprowadzać semafor w stan *wolny* (`xSemaphoreGive`)

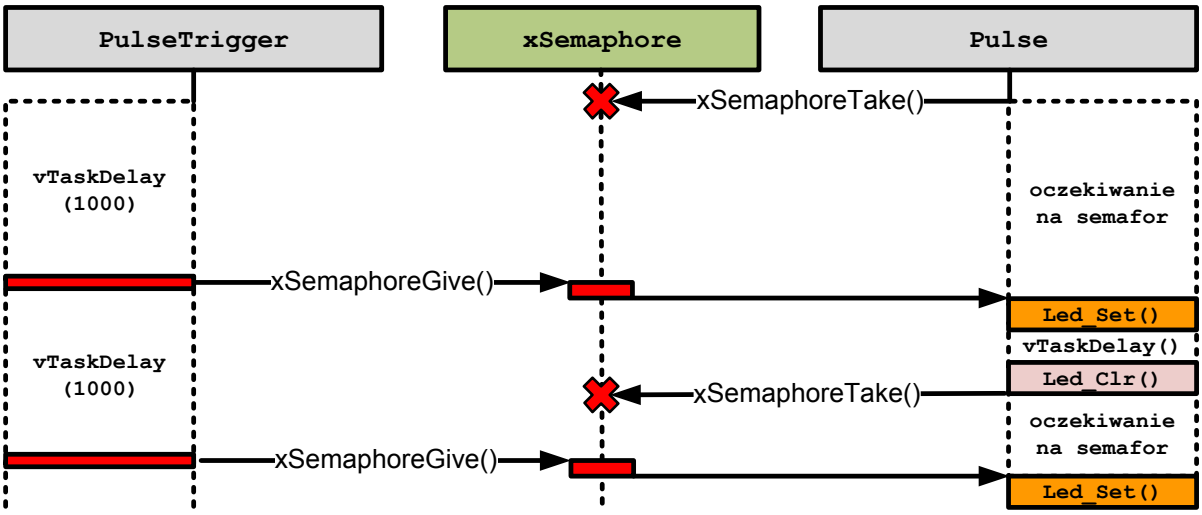
Inaczej mówiąc zadaniem wątku *Pulse* jest generowanie impulsu, a *PulseTrigger* wyzwalanie impulsu

Uwaga: Dołączyć moduł semafora - `#include "semphr.h"`

Rozwiązanie zad. 1:

Listing Błąd! W dokumencie nie ma tekstu o podanym stylu.**-2 Synchronizacja wątków poprzez semafor.**

Komentarz: Zasadę działanie programu ilustruje rysunek 3-7.



Rys. Błąd! W dokumencie nie ma tekstu o podanym stylu.**-1 Wykorzystanie semafora do synchronizacji wątków.**

Zadanie 2: Dodać do poprzedniego programu watek wyzwalający impuls co 1/3 sekundy i rozpoczynający pracę z opóźnieniem 1/3 sekundy (`vTaskDelay` przed `while`).

Rozwiązanie zad. 2:

Listing Błąd! W dokumencie nie ma tekstu o podanym stylu.**-3 Dodatkowy wątek zwalniający semafor.**

1.1.1.2 Ochrona zasobów

Zadanie 1: Na podstawie kodu ustalić funkcjonalność programu. Włączyć wywłaszczanie. Uruchomić program i podstawie jego działania sprawdzić czy działa prawidłowo.

```
void TxLetters (void *pvParameters){

    while(1){
```

```
            vTaskDelay(300);
            UART_Transmiter_SendString("-ABCDEFGHIJK-\n");
            while (cUART_Transmiter_IsFree()!=1){};
    }
}

void TxDigits (void *pvParameters){

      vTaskDelay(300);
      while(1){
            vTaskDelay(300);
            UART_Transmiter_SendString("-123456789-\n ");
            while (cUART_Transmiter_IsFree()!=1){};
      }
}

int main( void )
{
      UART_Init(150);
      xTaskCreate(TxLetters, NULL, 128, NULL, 1, NULL );
      xTaskCreate(TxDigits, NULL, 128, NULL, 1, NULL );
      vTaskStartScheduler();
      while(1);
}
```

Listing Błąd! W dokumencie nie ma tekstu o podanym stylu.**-4 Transmisja znaków z wykorzystaniem modułu UART mikrokontrolera.**

Komentarz: Powodem niezgodnego z oczekiwanym działania programu jest brak ochrony zasobu *UART_Transmiter* przed jednoczesnym użyciem przez dwa lub więcej wątków. Jednym z zastosowań semaforów jest ochrona zasobów. Niektóre z zasobów mikrokontrolera czy programu użytkownika nie mogą być używane jednocześnie przez wiele wątków.

Zadanie 2: Zmodyfikować przy użyciu funkcji `xSemaphoreTake` i `xSemaphoreGive` funkcje obu wątków tak aby program działał poprawnie.

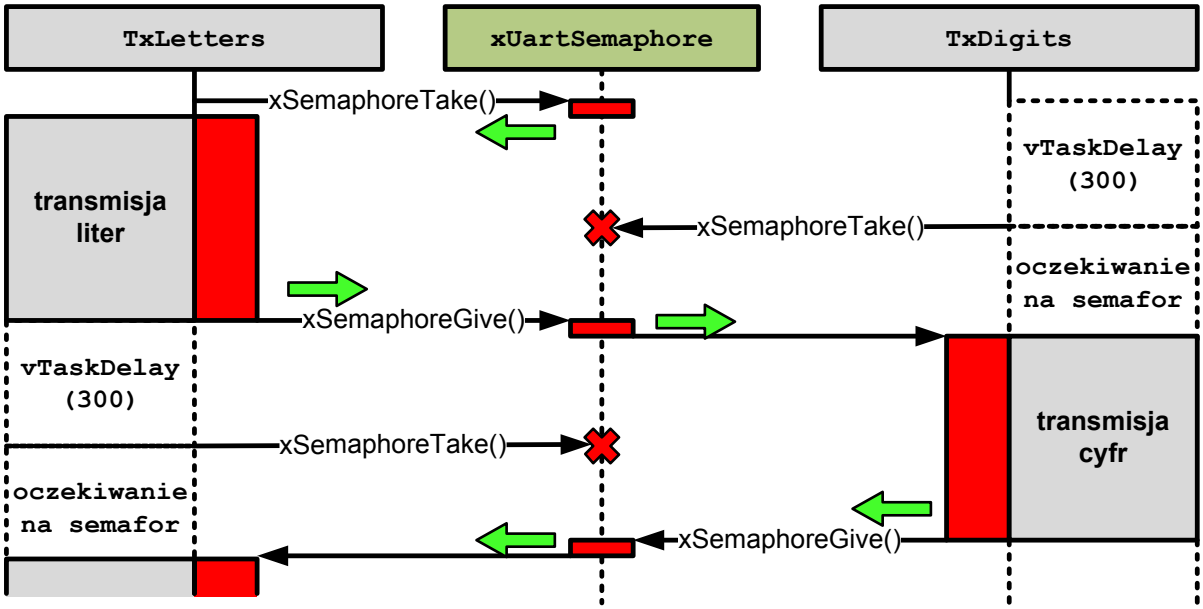
Rozwiązanie zad. 2:

```
xSemaphoreHandle xUartSemaphore;

void TxLetters (void *pvParameters){

    while(1){
            vTaskDelay(300);
            xSemaphoreTake(xUartSemaphore,portMAX_DELAY);
            UART_Transmiter_SendString("-ABCDEFGHIJK-\n");
            while (cUART_Transmiter_IsFree()!=1){};
            xSemaphoreGive(xUartSemaphore);
    }
}
```

```
void TxDigits (void *pvParameters){

    vTaskDelay(150);
    while(1){
            vTaskDelay(300);
            xSemaphoreTake(xUartSemaphore,portMAX_DELAY);
            UART_Transmiter_SendString("-123456789-\n ");
            while (cUART_Transmiter_IsFree()!=1){};
            xSemaphoreGive(xUartSemaphore);
    }
}

int main( void )
{
      UART_Init(150);
      vSemaphoreCreateBinary( xUartSemaphore );
      xTaskCreate(TxLetters, NULL, 128, NULL, 1, NULL );
      xTaskCreate(TxDigits, NULL, 128, NULL, 1, NULL );
      vTaskStartScheduler();
      while(1);
}
```

Listing Błąd! W dokumencie nie ma tekstu o podanym stylu.**-5 Wykorzystanie semafora do ochrony zasobów.**

Komentarz: Dostęp do modułu *UART_Transmiter* jest możliwy tylko gdy semafor jest wolny. Ilustruje to rysunek 3-8.



Rys. Błąd! W dokumencie nie ma tekstu o podanym stylu.**-2 Wykorzystanie semafora do ochrony zasobów.**

Kod będący rozwiązaniem zadania 2 posiada jednak pewną nadmiarowość. Sekwencja komend od `xSemaphoreTake` do `xSemaphoreGive` różni się tylko wysyłanym łańcuchem.

Zadanie 3: Stworzyć funkcje `RTOS_UART_Transmiter_SendString`, która sama będzie dbała o ochronę modułu `UART_Transmiter` przed jednoczesnym użyciem przez wiele wątków.

Rozwiązanie zad. 3:

```
```

Listing Błąd! W dokumencie nie ma tekstu o podanym stylu.**-6 Funkcja do ochrony modułu UART_Transmiter.**

1.1.2 Kolejki

Wprowadzenie: Zastosowanie semafora w poprzednich podpunktach pozwoliło w poprawny sposób wykorzystywać przez dwa watki jeden zasób (nadajnik układu transmisji szeregowej).

Posiada ono jednak wadę, która polega na blokowaniu wątku do momentu zwolnienia się zasobu, z którego chce skorzystać. Jeżeli wątek `TxDigits` chce wysłać łańcuch w momencie gdy wysyłany jest łańcuch przez wątek `TxLetters` to musi poczekać do momentu zakończenia wysyłania.
Można to zaobserwować dodając do obu wątków kod pozwalający mierzyć czas potrzebny na wykonanie jednej pętli.

Zadanie 1: Dołożyć do obu wątków z zadania 2 z poprzedniego rozdziału pomiar czasu wykonania pojedynczej pętli według poniższego przykładu:

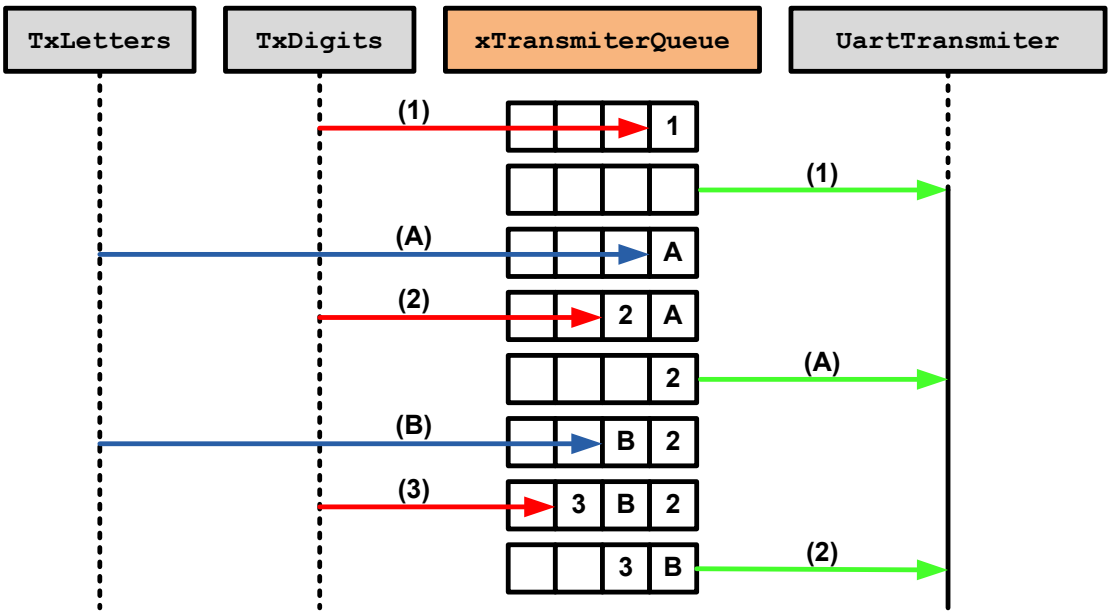
```
while(1){

      StartTick = xTaskGetTickCount();
      vTaskDelay(300);
      CopyString("-ABCDEFGHIJK-: ",cStringToSend);
      AppendUIntToString(TicksDiffer, cStringToSend);
      RTOS_UART_Transmiter_SendString(cStringToSend);
      StopTick = xTaskGetTickCount();
      TicksDiffer=StopTick-StartTick;
}
```

Listing Błąd! W dokumencie nie ma tekstu o podanym stylu.**-7 Pomiar czasu wykonywania wątku.**

Komentarz: Jak widać czasy wykonania pojedynczych wątków są znacznie dłuższe niż wynikało by to z argumentu funkcji opóźniającej. Potwierdza to fakt blokowania wątku spowodowany oczekiwaniem na zwolnienie nadajnika.

Wprowadzenie: Rozwiązaniem problemu blokowania wątku jest użycie *kolejki***.** *Kolejka* jest buforem pozwalającym na przechowywanie jednej lub więcej danych tego samego, dowolnego, typu. *Kolejka* jest buforem typu FIFO (First In, First Out) [2]. Oznacza to, ze elementy z takiego bufora mogą być pobierane tylko w takiej kolejności w jakiej zostały do niego wstawione. *Kolejki* dostarczane przez system operacyjny mogą być używane jednocześnie przez wiele wątków bez utraty integralności danych.

Zadanie 2: Przerobić program z *Zadania 1* tak aby wątki zamiast bezpośrednio wpisywać łańcuchy do nadajnika wstawiały je do kolejki. Pobieraniem danych z kolejki i wysyłaniem ich za pośrednictwem nadajnika powinien zajmować się stworzony tym celu wątek `UART_Transmiter`. Ilustruje to rysunek 3-9.



Rys. Błąd! W dokumencie nie ma tekstu o podanym stylu.**-3 Nadajnik UART – wykorzystanie kolejek.**

Sposób użycia kolejek (funkcje, typy danych) należy odszukać w dokumentacji systemu FreeRTOS [1].

Zaobserwować jak użycie kolejek wpłynęło na czas wykonywania pojedynczej pętli wątku.

Rozwiązanie zad. 2:

Listing Błąd! W dokumencie nie ma tekstu o podanym stylu.**-8 Wykorzystanie kolejki do transmisji znaków poprzez moduł UART.**

Zadanie 3: Przerobić program z *Zadania 3* z poprzedniego rozdziału tak aby nie dochodziło do blokowania wątków (użyć kolejki).

Rozwiązanie zad. 3:

```
void RTOS_UART_Transmiter_SendString(char* cString){

      xQueueSendToBack( xTransmiterQueue,cString, 0 );
}
```

Listing Błąd! W dokumencie nie ma tekstu o podanym stylu.**-9 Funkcja do transmisji znaków z wykorzystaniem kolejki.**

Zadanie 4: Dołączyć moduł *Servo* (*...\FreeRTOS\Demo\ARM7_LPC2129_Keil_RVDS*), a następnie wstawić i sprawdzić działanie programu testowego ćwiczenia 2 z rozdziału 9.2 *Kursu Podstawowego.*

Uwaga: Odczyt przycisków wstawić do wątku *Keyboard.* Stan przycisków powinien być sprawdzany z częstotliwością 10Hz.

Rozwiązanie zad. 4:

```

```

Listing Błąd! W dokumencie nie ma tekstu o podanym stylu.**-10 Sterowanie serwomechanizmem za pomocą przycisków.**

Zadanie 5: Przerobić moduł *Servo* żeby zamiast *Timerów* i *Przerwań* wykorzystywał systemową funkcję opóźniającą do cyklicznego wywołania automatu sterującego silnikiem krokowym. W tym celu stworzyć funkcję wątku `ServoTask`, której zadaniem jest cyklicznie wywoływać automat serwomechanizmu. Funkcja powinna znajdować się w module *Servo.* Wątek wykorzystujący funkcję powinien być tworzony i uruchamiany w funkcji `Servo_Init`.

Rozwiązanie zad. 5:

```

```

Listing Błąd! W dokumencie nie ma tekstu o podanym stylu.**-11 Wątek jako automat do sterowania serwomechanizmem.**

Komentarz: W obecnej wersji sterowanie pozycją serwomechanizmu odbywa się za pośrednictwem zmiennej `sServo.uiDesiredPosition` przy użyciu funkcji

`Servo_GoTo`. W takiej wersji programu sekwencja następujących po sobie wywołań `Servo_GoTo` spowoduje przejście serwomechanizmu od razu do pozycji wskazanej przez ostatnie wywołanie `Servo_GoTo`.

Zadanie 6: Przerobić program z *Zadania 5* tak aby sekwencja następujących po sobie wywołań `Servo_GoTo` powodowała przechodzenie serwomechanizmu po kolei przez wszystkie pozycje wskazane przez `Servo_GoTo`.

Należy w tym celu:

1. Stworzyć w module *Servo* kolejkę `ServoPositions` (wstawianie do kolejki powinno znajdować się w funkcji `Servo_GoTo`),
2. Wstawić do stanu *Idle* oczekiwanie na element z kolejki (następne `DesiredPosition`).

Działanie programu sprawdzić wywołując z maina sekwencje funkcji `Servo_GoTo(270), (180), (90),(0).`

Rozwiązanie zad. 6:

Listing Błąd! W dokumencie nie ma tekstu o podanym stylu.**-12 Wykorzystanie kolejki do kontrolowania pozycji serwomechanizmu.**

Zadanie 7: Przerobić program z poprzedniego zadania tak aby w kolejka przekazywała informacje zarówno o pozycji jak i o stanie , do którego ma przejść automat.

Działanie programu sprawdzić wywołując z maina sekwencje funkcji: `Servo_GoTo(270), Servo_Callib(), Servo_GoTo(180), Servo_Callib(), Servo_GoTo(90), Servo_Callib().`

Uwaga: Należy zmodyfikować również funkcję `Servo_Callib`.

Rozwiązanie zad. 7:

Listing Błąd! W dokumencie nie ma tekstu o podanym stylu.**-13 Wykorzystanie kolejki do kontrolowania pozycji i stanu serwomechanizmu.**