Mikroelektronika w Technice i Medycynie

Podstawy projektowania systemów wbudowanych

Instrukcja do ćwiczeń laboratoryjnych

Zastosowanie RTOS do sterowania serwomechanizmem

Mirosław Żołądź 2023

Spis treści

1 Przygotowanie środowiska

System FreeRTOS został skonfigurowany dla wielu różnych architektur i kompilatorów. Każdy port posiada własne wstępnie skonfigurowane demo, tak aby można było szybko zacząć pracę z systemem. Najprostszą drogą do stworzenia własnej aplikacji jest bazowanie na dostarczonych aplikacjach demonstracyjnych dla określonych portów.

a) Pobrać aktualną wersję systemu (www.freertos.org, „Download", pkt. 2).
b) Po rozpakowaniu archiwum usunąć:
    a. w katalogu głównym wszystko oprócz katalogu *FreeRTOS*,
    b. w katalogu *FreeRTOS* wszystko oprócz katalogów *Demo* i *Source*,
    c. w katalogu *Demo* wszystko oprócz katalogu *ARM7_LPC2129_Keil_RVDS*,
    d. w katalogu *ARM7_LPC2129_Keil_RVDS* katalogi *ParTest* i *serial*,
    e. w katalogu *Source\portable* wszystko oprócz katalogów *MemMang* i *RVDS*.
c) Uruchomić plik projektu środowiska Keil *Demo\ARM7_LPC2129_Keil_RVDS\RTOSDemo.Uv2*i.
d) W oknie drzewa projektu usunąć:
    a. katalog *Standard Demo*,
    b. z katalogu *Other* wszystko oprócz plików *startup* i *main*
e) zawartość pliku *main.c*, podmienić na poniższą

```
#include "FreeRTOS.h"
#include "task.h"

int main( void )
{
   while(1);
}
```

f) Skompilować projekt. Kompilacja powinna wykonać się bez błędów.
g) W pliku *Demo\ARM7_LPC2129_Keil_RVDS\FreeRTOSConfig.h* ustawić**:**
    a. configUSE_PREEMPTION na 0
    b. configCPU_CLOCK_HZ na ((unsigned long) 15000000)
    c. configTOTAL_HEAP_SIZE na ((size_t) 6 * 1024)

2 Wielowątkowość

2.1 Uruchamianie wątków

Wprowadzenie: Przedstawiony poniżej program pulsuje diodą 0 z częstotliwością 1 Hz wykorzystując funkcję `Led_Toggle()` znajdującą się w module *Led*. Funkcja ta zmienia na przeciwny stan diody o numerze podanym w argumencie. Do generowania opóźnień wykorzystywana jest funkcja `Delay()` oparta na pętli opóźniającej.

```
#include <lpc21xx.h>
#include "led.h"

void Delay(unsigned int uiMiliSec) {
            unsigned int uiLoopCtr, uiDelayLoopCount;
            uiDelayLoopCount = uiMiliSec*12000;
            for(uiLoopCtr=0;uiLoopCtr<uiDelayLoopCount;uiLoopCtr++) {}
}

int main( void ){
            Led_Init();
            while(1){
                  Led_Toggle(0);
                  Delay(500);
            }
}
```

Zadanie 1:

1. Uruchomić program i sprawdzić czy działa zgodnie z opisem,
2. Napisać program, który pulsuje diodą 0 z częstotliwością 1 Hz, a diodą 1 z częstotliwością 4 Hz,
3. Zastanowić jak napisać program, który będzie pulsował diodami z częstotliwością 3 Hz i 4 Hz.

Komentarz: Generowanie opóźnień z użyciem pętli opóźniającej ma następujące wady:

- konieczność modyfikacji kodu w przypadku zmiany mikrokontrolera/kompilatora (różne mikrokontrolery mogą wykonywać ten sam kod z różną prędkością, różne kompilatory mogą generować różny kod o rożnej prędkości wykonywania),
- skomplikowana implementacja cyklicznego wykonywania wielu zadań o różnych okresach cyklu,
- nieefektywne wykorzystanie mocy obliczeniowej mikrokontrolera.

Wprowadzenie: W poprzednim programie fragment kodu, odpowiadający za cykliczne wykonywania zadania, wykonywany był w pętli głównej znajdującej się w funkcji `main()`. W programach wykorzystujących RTOS może istnieć wiele pętli "głównych" pracujących jednocześnie. Nazywa się je wątkami (*threads*) i umieszcza w funkcjach (patrz funkcja `Led0Blink`). W RTOS funkcja `main()` służy do tworzenia i uruchamia wątków. W poniższym programie funkcja `main()`:

1. Inicjalizuje modułu *Led*,
2. Tworzy wątek (`xTaskCreate`). Funkcja jako pierwszy argument przyjmuje adres funkcji, w której znajduje pętla główna wątku. Znaczenie reszty argumentów jest w tym momencie nieistotne,
3. Uruchamia wszystkie istniejące wątki (`vTaskStartScheduler`),

UWAGA: Program „wychodzi” z `vTaskStartScheduler` tylko w przypadku niewystarczającej ilości pamięci wymaganej do utworzenia wątku "*idle task*" aktywowanego w trakcie bezczynności systemu czyli „normalnie” nigdy. Na wszelki wypadek dobrze jest jednak zostawić na końcu funkcji pętlę nieskończoną a najlepiej jakąś sygnalizację takiej sytuacji

```
#include "FreeRTOS.h"
#include "task.h"
#include "led.h"

void Delay(unsigned int uiMiliSec) {
   unsigned int uiLoopCtr, uiDelayLoopCount;
   uiDelayLoopCount = uiMiliSec*12000;
   for(uiLoopCtr=0;uiLoopCtr<uiDelayLoopCount;uiLoopCtr++){
   }
}

void Led0Blink( void *pvParameters ){
   while(1){
      Led_Toggle(0);
      Delay(500);
   }
}

int main(void){
   Led_Init();
   xTaskCreate(Led0Blink, NULL , 100 , NULL, 2 , NULL );
   vTaskStartScheduler();
   while(1);
}
```

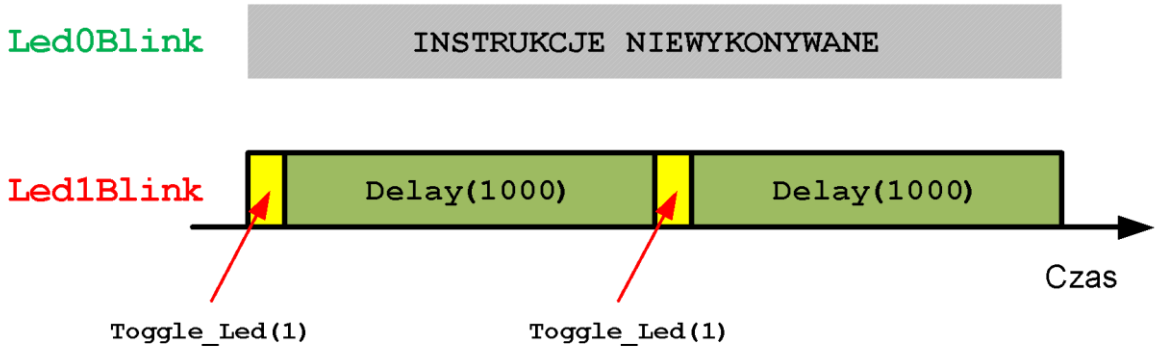
Zadanie 2: Zamienić obecny wątek wątkiem pulsującym diodą 0 z częstotliwością 0.5 Hz.
Uwaga: nie usuwać funkcji obecnego wątku.

Komentarz: Obecna implementacja posiada te same wady co poprzednia i została podana tylko w celu wprowadzenia do RTOS.

2.2 Wykorzystanie opóźnienia systemowego

Zadanie 1: Dodać do programu z poprzedniego zadania wątek pulsujący diodą 1 z częstotliwością 10 Hz. Zaobserwować działanie programu dla różnych kolejności tworzenia wątków.
Podpowiedź: Stworzyć funkcje `Led1Blink` i użyć powtórnie funkcji `xTaskCreate()`.

Komentarz: Wątek, który został utworzony jako ostatni, całkowicie przejmuje moc obliczeniową mikrokontrolera chociaż jego zadaniem jest tylko zmiana stanu jednej diody na przeciwny. Zadanie to zajmuje mniej niż 0.01% dostępnego czasu, resztę czasu zajmuje pętla opóźniająca. Nie jest to dobre rozwiązanie ponieważ z jednej strony nie zostaje dopuszczony do pracy drugi wątek, a z drugiej strony moc obliczeniowa pierwszego wątku marnowana jest na wykonywanie pętli opóźniającej. Ilustruje to rysunek 3-1:

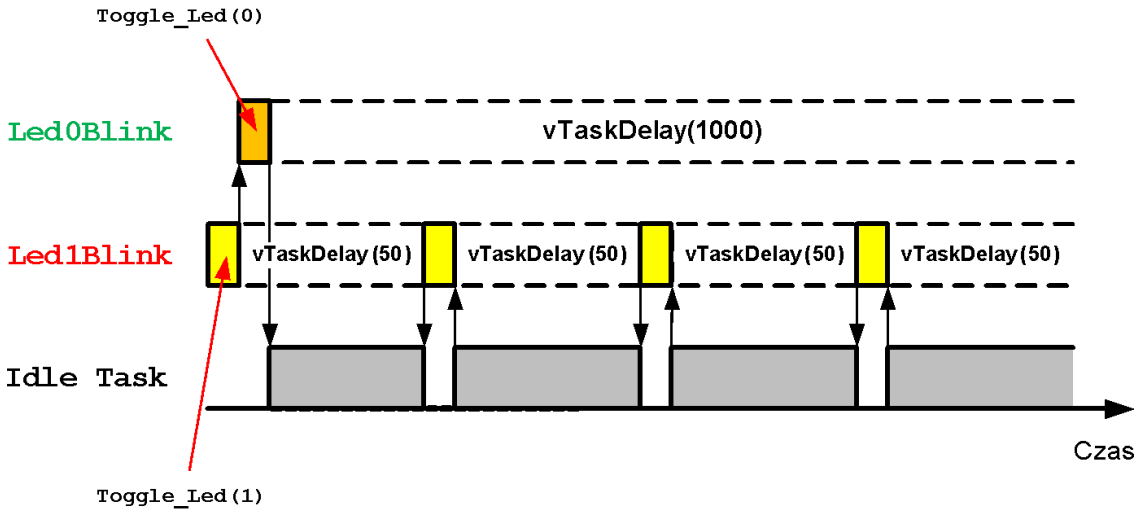


Rys. 2-1 Blokowanie wątku Led0Blink spowodowane użyciem pętli opóźniającej.

Rozwiązaniem tego problemu jest wykorzystanie funkcji opóźniającej `vTaskDelay()` dostarczanej wraz z RTOS (tzw. systemowej).

Zadanie 2: W programie z poprzedniego zadania zamiast funkcji `Delay()` użyć funkcji systemowej `vTaskDelay()`. Jak zmieniło się działanie programu?

Komentarz: W odróżnieniu od funkcji `Delay()` opartej na pętli opóźniającej funkcja `vTaskDelay()` zawiesza wykonywanie wątku i przekazuje sterowanie do systemu, co pozwala systemowi wykorzystać moc obliczeniową do wykonywania innych wątków. Jednocześnie `vTaskDelay()` informuje system, że powinien zwrócić sterowanie (moc obliczeniową) do zawieszonego wątku po ilości tyknięć systemu określonej w argumencie wywołania (`vTaskDelay(xx)`). Częstotliwość pracy systemu to 1000 Hz więc jedno tyknięcie trwa 1 ms. Działanie opóźnienia systemowego ilustruje rys. 3-2.



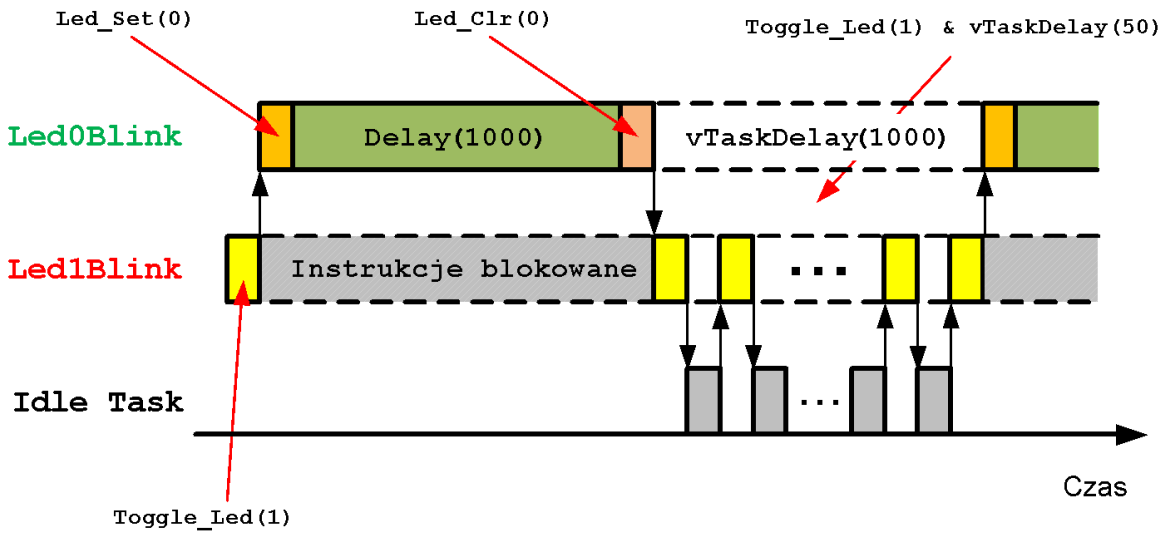
Rys. 2-2 Wykorzystanie funkcji vTaskDelay() do tworzenia nieblokujących opóźnień.

2.3 Równoległe wykonywanie zadań

Wprowadzenie: Powyższe rozwiązanie działa poprawnie (zadania wykonywane są równolegle) jeżeli zadania wykonywane w wątkach są stosunkowo krótkie. Może jednak zaistnieć sytuacja w której zadanie w jednym z wątków rzeczywiście potrzebuje więcej czasu na wykonanie. Ilustruje to program z poniższego zadania.

Zadanie 1: Przerobić funkcję `Led0Blink` tak, aby opóźnienie między zaświeceniem diody, a jej zgaszeniem było generowane funkcją `Delay()` (symulacja czasochłonnego zadania), a opóźnienie między zgaszeniem, a zapaleniem diody było realizowane funkcją systemową. Zaobserwować działanie.

Komentarz: Jak widać wątek `Led0Blink` blokuje wykonywanie wątku `Led1Blink` na czas wykonywania czasochłonnego zadania. Ilustruje to rys. 3-3.



Rys. 2-3 Blokowanie jednego z wątków przez wątek o długim czasie wykonania bez wywłaszczania.

Naszym celem jest spowodowanie żeby, krótkie zadania były wykonywane nieprzerwanie z założoną częstotliwością.

Istnieje możliwość aby wymusić zwrócenie sterowania przez wątek (w omawianym przykładzie podczas wykonywania funkcji `Delay()`). Jedną z podstawowych cech RTOS jest możliwość wywłaszczania (ang. preemption) wątków. System może w każdej chwili zawiesić wykonywanie wątku o niższym priorytecie i przekazać sterowanie do wątku o wyższym priorytecie.

Zadanie 2:

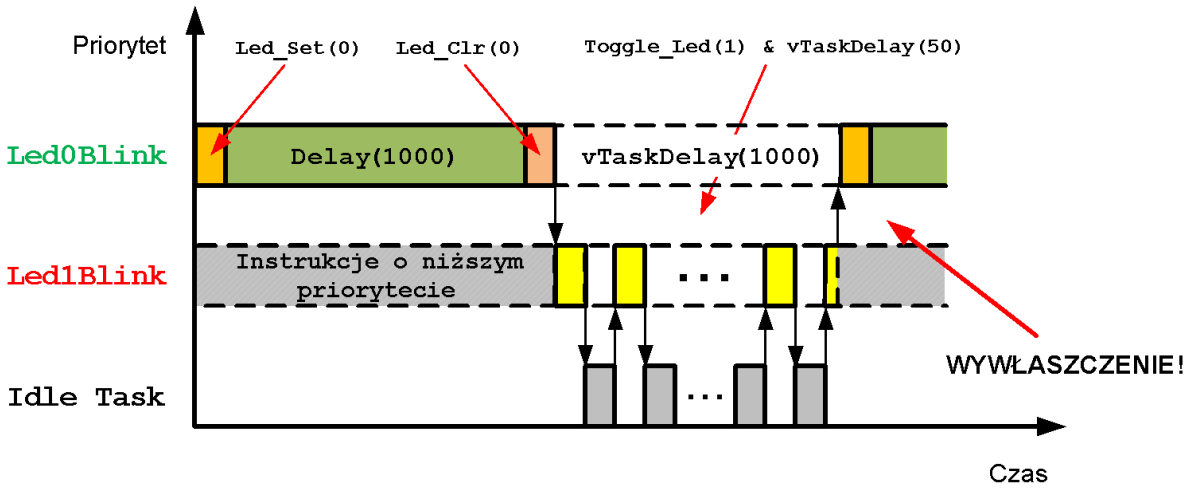
Włączyć wywłaszczanie (***FreeRTOSConfig.h,*** „`configUSE_PREEMPTION 1`”).

Ustawić niższy priorytet dla wątku pulsującego z wyższą częstotliwością (jeden z argumentów argument funkcji `xTaskCreate`)

Komentarz: Działanie programu z *Zadania 2* (z wywłaszczaniem) jest bardzo podobne do działania programu z *Zadania 1* (bez wywłaszczania). Zarówno w jednym jak i drugim wątek `Led0Blink` blokował na pół okresu (czas trwania „1”) wykonywanie wątku `Led1Blink`. Istnieje jednak pewna różnica między tymi przypadkami.

W przypadku programu bez wywłaszczania zadanie z wątku `Led0Blink` (`Delay`) mogło przejąć sterowanie dopiero po zwróceniu go przez wątek `Led1Blink` (`vTaskDelay`). Ilustruje to rysunek 3-3.

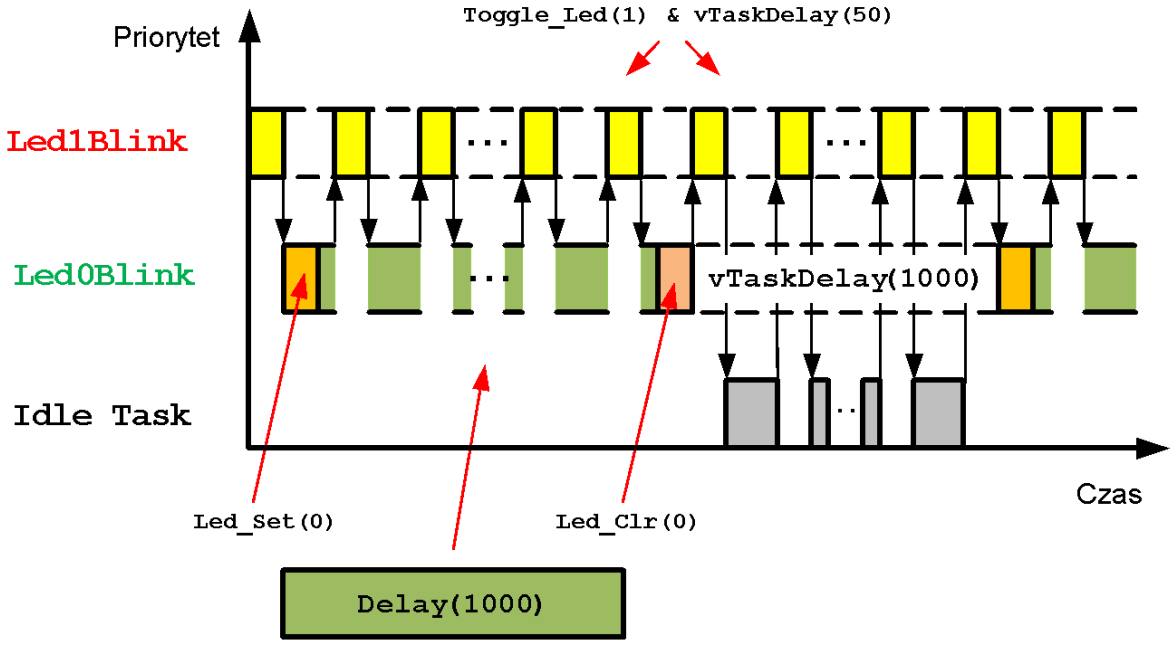
W przypadku programu z wywłaszczaniem zadanie z wątku `Led0Blink` (`Delay`) mogło przejąć sterowanie w każdym momencie, również w trakcie wykonywania przez wątek `Led1Blink` funkcji `Toggle_Led()`. Ilustruje to rysunek 3-4.



Rys. 2-4 Wywłaszczenie wątku o niższym priorytecie.

Zadanie 3: Odwrócić priorytety wątków i zaobserwować działanie programu.

Komentarz: Jak widać obecna wersja programu (*Zadanie 3*) działa w zamierzony sposób, t.j. czasochłonne zadanie z wątku `Led0Blink` nie blokuje wykonywania krótkich zadań z wątku `Led1`. Cel ten został osiągnięty przez użycie wywłaszczania oraz przyporządkowanie wyższego priorytetu wątkowi wykonującemu krótkie zadania (`Led1Blink`). Ilustruje to rysunek 3-5.



Rys. 2-5 Wywłaszczanie czasochłonnych instrukcji przez wątek o wyższym priorytecie.

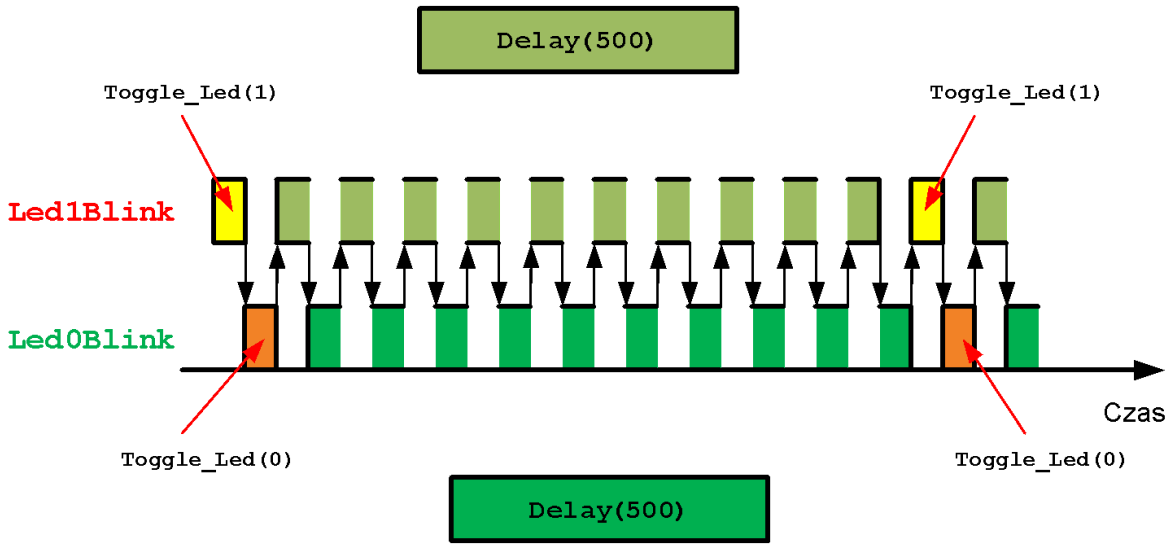
Wprowadzenie: Można sobie wyobrazić sytuację, w której oba wątki muszą równocześnie wykonywać czasochłonne zadanie. Pokazuje to program z Zadania 4.

Zadanie 4:

Zmodyfikować program z poprzedniego zadania tak, aby oba watki pulsowały diodą z okresem 1 Hz i używały tylko opóźnienia programowego (`Delay(500)`).

Ustawić równe priorytety obu wątków.

Komentarz: Jak można zaobserwować czas wykonywania zadań z poszczególnych wątków wydłużył się dwukrotnie (2 razy mniejsza częstotliwość pulsowania). Wynika to z faktu, że moc obliczeniowa dzielona jest równo na dwa wątki. Ilustruje to rysunek 3-6.



Rys. 2-6 Równoczesne wykonywanie dwóch czasochłonnych wątków z włączonym wywłaszczaniem.

Częstotliwość przełączania między wątkami może być ustawiana w pliku konfiguracyjnym za pomocą makra "`configTICK_RATE_HZ`" ustawionego domyślnie na 1000 Hz. Należy przy tym pamiętać, że samo przełączanie między wątkami wymaga pewnej ilości czasu i mocy obliczeniowej. Czyli, że zwiększając częstotliwość zwiększamy procent mocy obliczeniowej wykorzystywany przez funkcje systemowe, a nie przez wątki użytkownika.

Wniosek: Jeżeli nie ma takiej konieczności nie używać wywłaszczania.

3 Kontrola wątków

3.1 Przekazywanie argumentów

Wprowadzenie: Podczas tworzenia wątku (`xTaskCreate`) można przekazać do niego parametr. Parametr ten ma postać wskaźnika typu `void` czyli wskaźnika na zmienną o nieokreślonym typie. Poniższy program ma za zadanie uruchamiać miganie diody o okresie określonym w zmiennej zdefiniowanej w funkcji *main.*

```
void LedBlink( void *pvParameters ){

   unsigned char ucFreq = *((unsigned char*)pvParameters);

   while(1){
      Led_Toggle(0);
      vTaskDelay((1000/ucFreq)/2);
   }
}

int main( void )
{
   unsigned char ucBlinkingFreq =  10;

   Led_Init();
   xTaskCreate(LedBlink, NULL , 100 , &ucBlinkingFreq, 2 , NULL );
   vTaskStartScheduler();
   while(1);
}
```

Opis programu:

`main`

1. Stworzenie zmiennej `ucBlinkingFreq` określającej częstotliwość pulsacji diody,
2. Zainicjowanie modułu *Led*,
3. Stworzenie wątku `LedBlink` i przekazanie mu wskaźnika na `uiBlinkingFreq`. Wskaźnik pojawia się jako `pvParameters` w funkcji wątku `LedBlink`,
4. wystartowanie Shedulera.

`LedBlk`

1. Kopiowanie wartości wskazywanej przez `pvParameters` do zmiennej lokalnej `ucFreq` (należy zauważyć, ze zmienna ta jest tego samego typu co `ucBlinkingFreq`).
   Wymaga to:
   - zrzutowania wskaźnika typu `void` (typ nieokreślony) na wskaźnik typu `unsigned char`, `*((`**`unsigned char*)pvParameters`**`))`,
   - odwołania się do zmiennej wskazywanej przez wskaźnik – **`*`**`((unsigned char*)pvParameters))`.
2. Zmiana stanu diody na przeciwny oraz opóźnienie zależne od przekazywanego parametru.

Zadanie 1: Sprawdzić działanie programu dla różnych częstotliwości.

Komentarz: Dotychczas funkcja wątku pracowała na kopii zmiennej przechowującej informacje o częstotliwości (`ucFreq`), czyli nawet jeżeli zmienilibyśmy wartość zmiennej `ucBlinkingFreq` to i tak nie miałoby to wpływu na częstotliwość pulsowania.

Zadanie 2: Dorobić wątek `LedCtrl` (Control), który za pośrednictwem zmiennej `ucBlinkingFreq`, będzie co sekundę zmieniał częstotliwość pulsowania diody. (zmodyfikować program tak aby częstotliwość pulsacji zależała od aktualnej wartości zmiennej `ucBlinkingFreq`).

Zadanie 3: Przerobić program z poprzedniego zadania tak aby wątek `LedCtrl` co sekundę zmieniał częstotliwość pulsowania diody, a co dwie sekundy numer pulsującej diody (trzeba stworzyć strukturę).

3.2 Zawieszanie i odwieszanie wątków

Wprowadzenie: RTOS pozwala na zatrzymywanie i ponowne uruchamianie wątków. Służą do tego funkcje odpowiednio `vTaskSuspend i vTaskResume`. Jako argument funkcje przyjmują tzw. uchwyt wątku będący zmienną typu `xTaskHandle`. Uchwyt do wątku jest zwracany przez `xTaskCreate( …, &xMyHandle )`.

Zadanie 1: Napisać program który cyklicznie przez sekundę pulsuje i przez sekundę nie pulsuje diodą. Należy użyć dwóch wątków. Jeden odpowiedzialny za pulsowanie drugi odpowiedzialny za zawieszanie/odwieszanie pierwszego.

4 Mechanizmy synchronizacji

4.1 Semafory

4.1.1 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 );
```

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.

Wątek `PulseTrigger` powinien co sekundę wprowadzać semafor w stan *wolny*

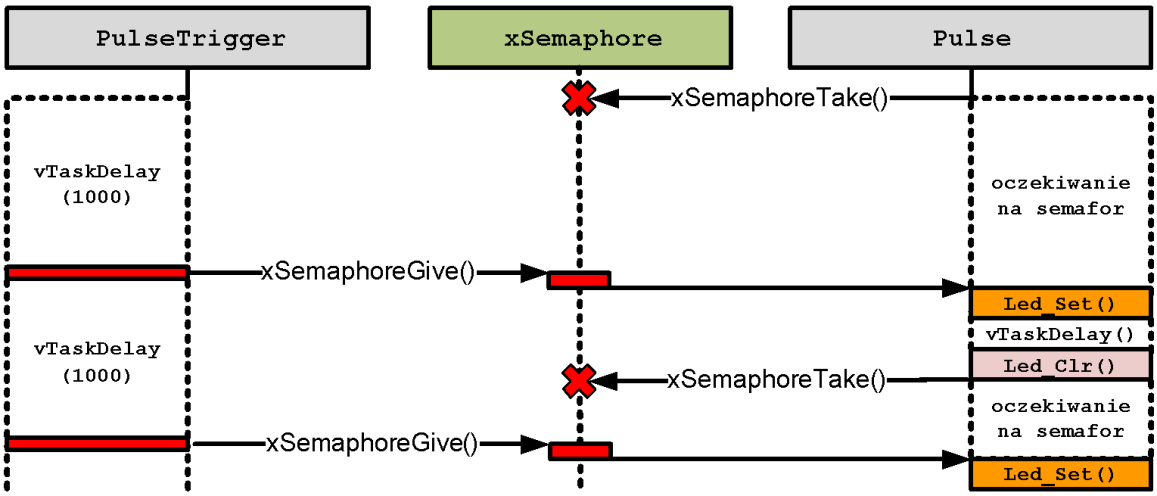
Wątek `Pulse_LED0` powinien po każdym zwolnieniu semafora zaświecić a `LED0` na czas 0.1s.

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

UWAGI:

- Dołączyć moduł semafora - `#include "semphr.h"`
- Semafor powinien być zmienną globalną

Zasadę działanie programu ilustruje poniższy rysunek.



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

Zadanie 3: Dodać do programu z zadnia **1** wątek `Pulse_LED1` sterujący `LED1` wyzwalany z tego samego semafora co `Pulse_LED0`.

Zadanie 4. Zrealizować funkcjonalność z zadania 3, ale z użyciem jednej funkcji do sterowania diodą (`Pulse_LED`). Podpowiedź: Jedna funkcja (sparametryzowana) dwa wątki.

4.1.2 Ochrona zasobów

Zalecenie: Testy poniższych zadań należy przeprowadzać w pierwszej kolejności z uzyciem symulatora a następnie z użyciem zestawu uruchomieniowego.

Zadanie 1 Wstawić do pliku `main` i doprowadzić do działania poniższy kod. W tym celu dołączyć moduł `string` i `uart`.

```
void LettersTx (void *pvParameters){

while(1){
   Transmiter_SendString("-ABCDEEFGH-\n");
   while (eTransmiter_Status()!=FREE){};
         vTaskDelay(300);
}
}

int main( void ){
UART_InitWithInt(9600);
xTaskCreate(LettersTx, NULL, 128, NULL, 1, NULL );
vTaskStartScheduler();
while(1);
}
```

Zadanie 2 Zmniejszyć prędkość transmisji do 300 bodów. Oprócz zmiany wartości argumentu przekazywanego do funkcji `UART_InitWithInt` konieczna jest podmiana definicji funkcji na podaną w załączniku na stronie laboratorium. (Tak niska prędkość transmisji wymaga bardziej złożonej inicjalizacji UARTA)

Zadanie 3 Dodać wątek klawiatury `KeyboardTx`, którego zadaniem jest wysyłanie łańcucha znakowego ("-Keyboard-\n") po każdym naciśnięciu dowolnego przycisku. Należy użyć modułu `keyboard`. Modułu nie należy modyfikować. Transmisja łańcucha powinna odbywać się dokładnie tak jak w wątku `LettersTx`.

Zadanie 4 Jak widać łańcuchy znakowe „przerywają się” co należy uznać za efekt niepożądany. Jednym ze sposobów, aby go uniknąć jest zastosowanie sekcji krytycznych. Sekcje powinny obejmować fragmenty kodu bezpośrednio związane z wysyłaniem łańcuchów. Sekcje należy zrealizować z użyciem semafora.

Zadanie 5 W obecnej wersji programu występują dwa powtarzające się fragmenty kodu. Należy je wyodrębnić do nowej funkcji o nazwie `Rtos_Transmiter_SendString`.

Zadanie 6 W obecnej wersji programu zdarza się, że jeden z wątków musi czekać aż drugi wątek wyjdzie z sekcji krytycznej. W pewnych okolicznościach może to być wadą. Aby to zilustrować należy dodać funkcji wątku `LettersTx` pomiar czasu potrzebnego na wykonanie funkcji `Rtos_Transmiter_SendString`. Do pomiaru czasu należy użyć funkcji `xTaskGetTickCount`. Informacja o czasie powinna być wyrażona w *tick-ach* systemowych oraz wyświetlana w formacie cheksadecymalnym na końcu łańcucha wysyłanego dotąd przez letters (`-ABCDEEFGH-:0x0230`). Nie zapomnieć o znaku następnej linii (\n). Zaobserwować jak zmienia się czas wykonanie wspomnianej funkcji podczas naciskania przycisków.

4.2 Kolejki

4.2.1 UART

Zadanie 1 Aby usunąć występujące w zadaniu 6 blokowanie się wątków można użyć kolejki. Watki `LettersTx` i `KeyboardTx` powinny wstawiać łańcuchy do kolejki. Odczyt kolejki oraz zapis do Uart-a powinien znajdować się w oddzielnym wątku. Sugeruje się zaadoptowanie funkcji `Rtos_Transmiter_SendString`. Jako rozmiar kolejki przyjąć 5 elementów.
Ile wynosi (w tickach) czas zapisu do kolejki ?

Zadanie 2 Dodać do wątku LettersTx funkcjonalność polegającą na zmianie stanu LED0 na przeciwny, jeżeli wpisanie do kolejki nie powiodło się. Zaobserwować efekt, przedyskutować z prowadzącym.

4.2.2 Servo

Zadanie 1 Przygotować zawartość pliku `main.c`, tj. :

- usunąć wszystkie funkcje oraz związane z nimi fragmenty kodu oprócz funkcji `main`,
- pozostawić uruchamianie shedulera,
- odłączyć moduł `UART`, dołączyć moduł `Servo`.
- dodać do programu inicjalizację serwomechanizmu oraz wątek klawiatury jak poniżej,
- zweryfikować inicjalizację modułu servo (timer zerowy oraz zerowy slot przerwań są zarezerwowane dla systemu FreeRTOS)

Sprawdzić działanie na serwomechanizmie dla częstotliwości 100 Hz.

```
void Keyboard (void *pvParameters){

	while(1){

		swith(){
			case BUTTON_1: Servo_Callib();
			break;

			case BUTTON_2: Servo_GoTo(50);
			break;

			case BUTTON_3: Servo_GoTo(100);
			break;

			case BUTTON_4: Servo_GoTo(150);
			break;
		}

		vTaskDelay(100);
	}
}
```

Przerobić program tak aby reagował na naciśniecie a nie na stan przycisków.

Zadanie 2: Przerobić moduł `Servo` żeby do cyklicznego wywołania automatu sterującego silnikiem krokowym zamiast timera i przerwania wykorzystywał mechanizmy RTOS.
Sprawdzić działanie dla 100 Hz.
Zastanowić się a następnie sprawdzić czy potrzebne jest wywłaszczanie ?

Zadanie 3: Przerobić program tak aby pracował z częstotliwością 200 Hz.
Podpowiedź**:** Sprawdzić w czym wyrażony jest argument opóźnienia systemowego oraz przejrzeć zawartość pliku konfiguracji systemu.

Zadanie 4: W obecnej wersji sterowanie pozycją serwomechanizmu odbywa się za pośrednictwem zmiennej globalnej `sServo.uiDesiredPosition` za pośrednictwem funkcji `Servo_GoTo` oraz `Servo_Callib`.
Jaki problem mógłby wystąpić jeśli program działałby na mikrokontrolerze 8-bitowym ?
Zamiast zmiennej globalnej użyć kolejki nie modyfikując w żaden sposób pliku `main`.

Zadanie 5: Zmodyfikować program tak aby naciśniecie przycisku `BUTTON_4` powodowało przejście przez pozycje: 12, 0, 24, 0, 36, 0.

Zadanie 6: Dodać do modułu `servo` funkcję `Servo_Wait`. Czas oczekiwania powinien być wyrażony w tick-ach systemowych. Test przeprowadzić za pomocą następującej sekwencji pozycji i okresów przestojów**:** 12, 100, 0, 24, 200, 0, 36, 300, 0.

Zadanie 7: Dodać do modułu `servo` funkcję `Servo_Speed`. Prędkość powinna być wyrażona w tick-ach systemowych pomiędzy następującymi po sobie krokami silnika. Test przeprowadzić za pomocą następującej sekwencji pozycji i ustawień prędkości **:** 8, 12, 4, 24, 2, 36, 1, 0.

5 Obsługa przerwań

5.1 Timer

Zadanie 1. Zamienić zwartość pliku `main.c` na poniższą a następnie doprowadzić do działania poniższy program.

```
#include "FreeRTOS.h"
#include "task.h"
#include "led.h"
#include "timer_interrupts.h"

void LedBlink(void)
{
	Led_Toggle(0);
}

int main(void)
{
	Led_Init();
	Timer1Interrupts_Init(500, &LedBlink);
	vTaskStartScheduler();
	while(1);
}
```

Zadanie 2. W programie z zadania 1 wywołanie funkcji `Led_Toggle` zsynchronizowane jest z Timer-em za pomocą przerwań.

Zsynchronizować wywołanie funkcji `Led_Toggle` z wystąpieniem przerwania używając semafora.

UWAGA: operowanie na semaforze z przerwań wymaga zastosowania odpowiednich funkcji (patrz API systemu na stronie FreeRTOS).

5.2 UART – Odbiornik

Zadanie 1 Doprowadzić do kompilacji poniższy program, w tym dopasować nazwy funkcji z modułu uart do nazw z poniższego listingu. Sprawdzić działanie programu.

```
#include "FreeRTOS.h"
#include "task.h"

#include "led.h"
#include "uart.h"

void UartRx( void *pvParameters ){

    char acBuffer[UART_RX_BFFER_SIZE];

    while(1){
        while (eUartRx_GetStatus()==EMPTY){};
        Uart_GetStringCopy(acBuffer);
        Led_Toggle(0);
    }
}

int main( void )
{
    Led_Init();
    UART_InitWithInt(9600);


    xTaskCreate( UartRx,  NULL , 100  , NULL, 1 , NULL );
    vTaskStartScheduler();

    while(1);
}
```

Zadanie 2 Zaimplementować odbiornik łańcuchów znakowych z użyciem kolejki systemowej.

W tym celu należy:

1. Usunąć z modułu UART wszystkie funkcje i zmienne związane z odbiorem komend a następnie doprowadzić do kompilacji modułu.
2. Dodać do funkcji interfejsowych funkcję `char cUart_GetChar(void)`, na razie w postaci mockup-a.
3. Zastąpić zawartość pętli głównej wątku `UartRx` następujacą linijka kodu:

   ```
   Led_Toggle(cUart_GetChar()-'0');
   ```

   Zlinkować program (build all)
4. Dodać do modułu `UART` kolejkę o długości i typie elementów takich jak bufor odbiornika z wcześniejszego programu.

   Zmodyfikować funkcję obsługi przerwania uart-a tak aby dodawała do kolejki odebrane znaki.

   Zmodyfikować funkcję `cUart_GetChar` tak aby odbierała z kolejki znaki i zwracała je jako wartość wyjściową.

   Dokonać innych modyfikacji koniecznych do poprawnego działania programu.

Zadanie 3

Plik main.c

Zmodyfikować zawartość pętli głównej wątku `UartRx` jak poniżej.

```
Uart_GetString(acBuffer);
Led_Toggle(acBuffer[0]-'0');
```

Moduł `UART`

Zastąpić funkcję `cUart_GetChar` funkcją `void Uart_GetString(char *)`. Funkcja powinna zwracać za pośrednictwem przekazanego do niej wskaźnika odczytany z kolejki odbiornika łańcuch znakowy zakończony znakiem terminatora, przy czym znak terminatora powinien być zastąpiony wartością `NULL`.

Przetestować program.

Zadanie 4 Zmodyfikować wcześniejszy program tak aby po odebraniu komendy „zero" zmieniał na przeciwny stan LED_0 a po odebraniu komendy „jeden" zmieniał na przeciwny stan LED_1.

Użyć funkcji do porównywania łańcuchów lub funkcji do dekodowania.

5.3 UART – Nadajnik

Zadanie 1 Zastąpić zawartość pętli głównej wątku `UartRx` następującym kodem:

```
Uart_GetString(acBuffer);
Uart_PutString(acBuffer);
```

Zmodyfikować moduł UART tak aby:

- funkcja `Uart_PutString` wstawiała przekazany do niej łańcuch znakowy do kolejki oraz inicjalizowała wysyłanie znaków,
- funkcja obsługi przerwania „wysyłała" znaki odebrane z kolejki (znak NULL powinien być zastępowany znakiem „\r").

Usunąć z modułu UART wszystkie zbędne fragmenty kodu.

Sprawdzić działanie programu.

Zadanie 2

a) Zainicjalizować acBuffer łańcuchem znakowym „0123456789\n".

Zastąpić zawartość pętli głównej wątku UartRx następującym kodem:

```
vTaskDelay(500);
Uart_PutString(acBuffer);
Uart_PutString(acBuffer);
Uart_PutString(acBuffer);
```

Sprawdzić działanie programu (wystarczy w symulatorze).

b) Zmodyfikować zawartość pętli głównej wątku UartRx jak po niżej:

```
vTaskDelay(500);
Uart_PutString(acBuffer);
vTaskDelay(10);
Uart_PutString(acBuffer);
vTaskDelay(10);
Uart_PutString(acBuffer);
```

Sprawdzić działanie programu (wystarczy w symulatorze).

c) Zastanowić się skąd bierze się niepoprawne działanie programu z podpunktu. Przedyskutować z prowadzącym

6 Program finalny

6.1 Moduły klawiatury i zegarka

Zadanie 1

Moduł `Keyboard` powinien posiadać:

- funkcję `eReadButtons` wykonującą dokładnie to samo co wcześniej funkcja `eKeyboardRead`,
- Jednoelementową kolejkę, z elementem tego samego typu jaki zwraca `eReadButtons`,
- Wątek `Keyboard_Thread`, odpowiedzialny wykrycie naciśnięcia przycisku i wstawienie informacji o naciśniętym przycisku do kolejki,
- funkcję `eKeyboard_Read` zwracającą informację o naciśniętym przycisku odczytaną z kolejki,
- Funkcję `Keyboard_Init` zawierającą odpowiednie inicjalizacje.

Zmodyfikować plik **`main.c`** tak aby naciśniecie przycisku powodowało zmianę na przeciwny stanu odpowiadającego mu leda (S1-D1, S2-D2, itd.).

Zadanie 2 Zmodyfikować moduł Watch w sposób analogiczny jak moduł Keyboard. Plik nagłówkowy modułu powinien wyglądać w sposób następujący.

```
enum TimeUnit {SECCONDS, MINUTES};

struct WatchEvent  {
      enum TimeUnit eTimeUnit;
      char TmeValue;
};

void  Watch_Init(void);
struct WatchEvent sWatch_Read(void);
```

Zmodyfikować plik **`main.c`** tak aby zmiana sekund powodowała zmianę na przeciwny stanu D1, zmiana minut powodowała zmianę na przeciwny stanu D2.

Zadanie 3 Napisać program, który będzie wysyłał przez UART zarówno informację o naciśnietym przycisku jak i zmianie sekund/minut. (buton 0x000X, sek 0x000X, min 0x000X)

6.2 Wspólna kolejka zdarzeń

Zadanie 1 Zmodyfikować plik **`main.c`** tak aby zawierał jedną wspólną kolejkę zdarzeń.

Elementem kolejki powinny być tablice znaków o długości 20. Tablice te będą służyć do przechowywania zdarzeń w postaci łańcuchów znakowych.

Źródłem zdarzeń powinny być: klawiatura i odbiornik UART-a. W pliku main.c należy umieścić wątki które będą odbierać zdarzenia z wymienionych modułów i wstawiać je do wspólnej kolejki zdarzeń.

W przypadku modułów klawiatury zdarzenia powinny mieć formę jak w zadaniu 3 z podpunktu 6.1. W przypadku odbiornika UART-a zdrzenie powinno być kopią odebranego łańcucha.

Zdarzenia ze wspólnej kolejki powinny być odbierane i obsługiwane (wykonanie odpowiedniej akcji) w wątku Executor. Obsługę powinno poprzedzać dekodowanie zdarzeń, czyli łańcuchów.

W ramach testu należy zaimplementować jednoczesne:

- wykonywanie komend (z uarta): `id`, `callib`, `goto`,
- sterowanie Servo za pomocą przycisków (`callib, goto`).

UWAGA:

- koniecznie usunąć z projektu moduły led i watch
- po odebraniu `calib` i `goto` program powinien odsyłać `ok`
- zmniejszyć długość tablic na łańcuchy odbieranie i wysyłane (moduł uart) do 20

6.3 ->>>>

6.3 Odczyt statusu serwomechanizmu

Zadanie 1 Dodać możliwość odczytu statusu serwomechanizmu

a) Dodać do funkcji wątku obsługi zdarzeń deklarację zmiennej.

```
struct ServoStatus sServoStatus;
```

oraz case

```
case STATE:
sServoStatus = Servo_State();
switch (sServoStatus.eState){
    case _CALLIBRATION: CopyString("state callib ",cString);      break;
    case _IDDLE       : CopyString("state iddle ",cString);       break;
    case _IN_PROGRESS : CopyString("state in_proggres ",cString); break;
    case _WAITING     : CopyString("state waiting ",cString);     break;
    default:break;
};
AppendUIntToString(sServoStatus.uiPosition,cString);
AppendString("\n",cString);
UART_PutString(cString);
break;
```

Doprowadzić program do bezbłędnej kompilacji. Sprawdzić działanie programu.

b) Zastąpić aktualną implementację funkcji `Servo_State` poniższą implementacją

```
struct ServoStatus Servo_State(void){
    struct ServoStatus sServoStatus;

    xQueuePeek(xStatusQueue,&sServoStatus,portMAX_DELAY);
    return sServoStatus;
}
```

Doprowadzić program do bezbłędnej kompilacji. Zapoznać się ze specyfikacja funkcji `xQueuePeek`.

UWAGA: kolejka statusu powinna być jednoelementowa.

c) Wstawić do funkcji wątku `Automat` w module `Servo` kod wstawiający dane do kolejki statusu.

UWAGA:

- funkcja wstawiająca element do kolejki może być użyta maksymalnie 2 razy
- użyć funkcji, która nadpisze element obecny w kolejce

Przeprowadzić testy programu
PODPOWIEDŹ: Zwolnić serwo tak aby móc zaobserwować zmiany stanów i pozycji