AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI, INFORMATYKI I ELEKTRONIKI

KATEDRA METROLOGII

Praca dyplomowa

magisterska

Stanowisko laboratoryjne do nauki Systemu Operacyjnego Czasu Rzeczywistego

Laboratory setup for Real-time operating systems teaching

Imię i nazwisko: *Tomasz Biel*
Kierunek studiów: Elektrotechnika
Opiekun pracy: *dr inż. Mirosław Żołądź*

Kraków, rok 2012

Spis treści

CEL PRACY

Głównym celem pracy jest przygotowanie stanowiska laboratoryjnego do nauki systemu operacyjnego czasu rzeczywistego na przykładzie darmowego systemu FreeRTOS. Aby zrealizować cel pracy należy:

- Zapoznać się z działaniem systemu FreeRTOS oraz jego podstawowymi mechanizmami.
- Napisać instrukcję konfiguracji systemu dla wybranego portu oraz środowiska programistycznego.
- Stworzyć zestaw ćwiczeń do samodzielnego wykonania.
- Przygotować przykładowe rozwiązania do zamieszczonych ćwiczeń.
- Pokazać możliwość przeniesienia napisanych wcześniej programów na inny mikrokontroler.

1 WSTĘP

System operacyjny to program lub zespół programów odpowiedzialny za zarządzanie zasobami sprzętowymi komputera lub mikrokontrolera. Oprócz kontroli zasobów sprzętowych system odpowiedzialny jest za wykonywanie powierzonych mu zadań. Jedną z najważniejszych cech takiego systemu jest jego wielozadaniowość. Uruchamiając kilka aplikacji użytkownik ma wrażenie, że wszystkie zadania wykonywane są jednocześnie. Zasada działania polega na tym, że każdy wątek korzysta z procesora przez pewien czas, a dostatecznie szybkie przełączanie pomiędzy aktualnie wykonywanymi zadaniami daje wrażenie równoległego wykonywania operacji [3].

Systemy operacyjne czasu rzeczywistego charakteryzują się tym, że reakcja na zaistniałe zdarzenia w świecie rzeczywistym powinna być zapewniona w deterministycznym czasie. Wykorzystuje się je w szeroko pojętym przemyśle i automatyce, gdzie ważnym elementem systemu jest jego niezawodność oraz bezpieczeństwo. Ze względu na wymagania czasowe systemy czasu rzeczywistego można podzielić na dwa rodzaje [3]:

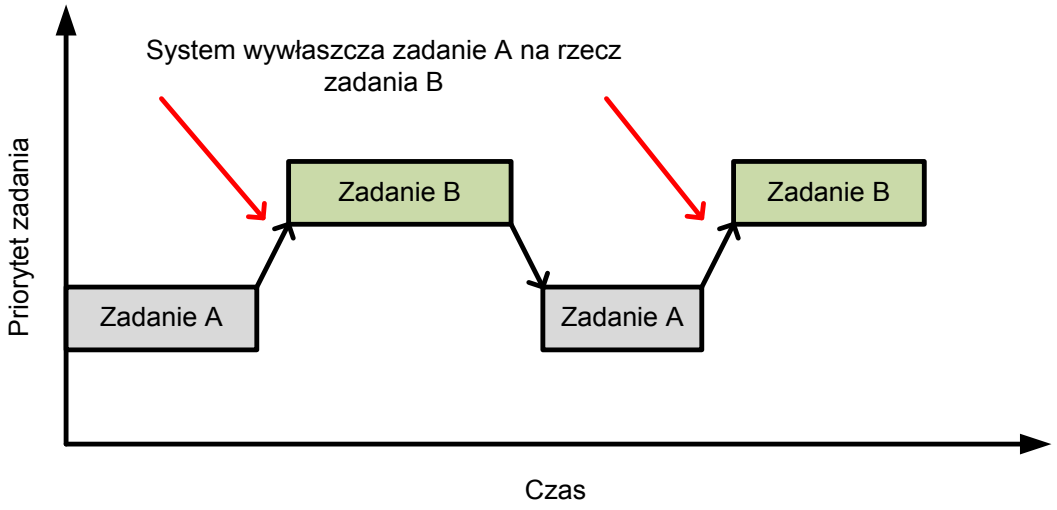
- twarde – maksymalny czas reakcji na zdarzenie zewnętrzne jest znany, oraz wiadome jest, że nie zostanie on przekroczony,
- miękkie – maksymalny czas reakcji znany jest z pewną dokładnością, oraz wiadome jest, że nie zostanie on przekroczony w większości przypadków.

1.1 System FreeRTOS

FreeRTOS jest systemem operacyjnym czasu rzeczywistego przeznaczonym do małych platform, gdzie ilość dostępnych zasobów jest ich głównym ograniczeniem (np. pamięć operacyjna). Jest to darmowy system oparty na modyfikowanej licencji GPL (z ang. General Public License) dającej możliwość wykorzystywania systemu w aplikacjach komercyjnych bez udostępniania kodu źródłowego [1]. W odróżnieniu od innych systemów operacyjnych FreeRTOS wymaga aby aplikacje zostały skompilowane razem z jądrem systemu i razem z nim wgrane do obsługiwanego sprzętu.

1.2 Zasada działania systemu

FreeRTOS jest systemem operacyjnym z wywłaszczaniem zadań. Polega to na tym, że aktualnie wykonywane zadanie może zostać przerwane na rzecz zadania o wyższym priorytecie wykonania [3]. Zasada działania tego procesu pokazana jest na rys. 1-1.



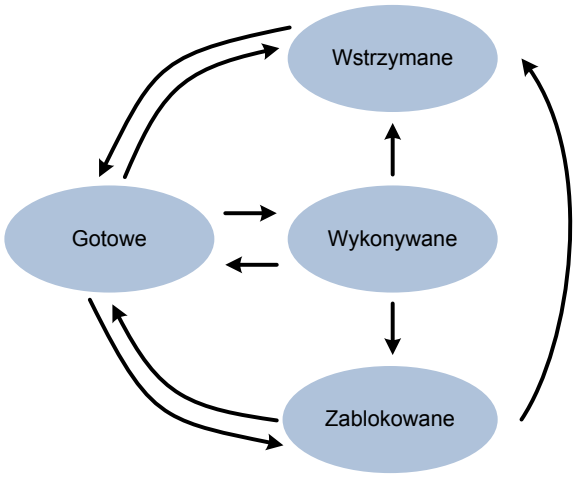
Rys. 1-1 Zasada działania wywłaszczania zadań.

Procesy w systemie FreeRTOS wykonywane są na dwa sposoby: za pomocą zadań (Tasks) lub współprogramów (Co-routines). Zadania to całkowicie niezależne procesy posiadające własny stos. Współprogramy działają podobnie lecz, w odróżnieniu od zadań, posiadają wspólny stos i wykorzystuje się je głównie w architekturach o ograniczonych zasobach sprzętowych (posiadających małą ilość pamięci RAM).

Każde zadanie występujące w systemie może posiadać określony stan. Podzielić je można następująco [2]:

- wykonywane – zadanie korzysta aktualnie z zasobów mikrokontrolera,
- gotowe do wykonania – zadanie czeka na zwolnienie zasobów mikrokontrolera,
- zablokowane – zadanie czeka na pewne zdarzenie, np. upłynięcie określonego czasu,
- wstrzymane – zadanie nie jest uwzględniane przez system i czeka na ewentualne wznowienie.

Możliwe przejścia pomiędzy poszczególnymi stanami widoczne są na rys. 1-2.



Rys. 1-2 Możliwe przejścia między stanami pracy zadania.

Nad wykonywaniem poszczególnych zadań czuwa tzw. planista (scheduler). Jest to element systemu, który odpowiada za kolejkowanie poszczególnych zadań w zależności od przypisanych im priorytetów wykonania. Częstotliwość przełączeń pomiędzy poszczególnymi zadaniami określana jest jako wewnętrzne taktowanie systemu. Jego wartość ustala się przy wstępnej konfiguracji systemu (opisana później).

2 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. Można wykorzystać istniejące funkcje lub całkowicie je usunąć.

2.1 Wymagania

Do wykonania ćwiczeń potrzebny jest zestaw uruchomieniowy ZL5ARM oraz kod źródłowy systemy FreeRTOS. Głównym elementem płytki jest mikrokontroler LPC2129. Na płytce testowej oprócz mikrokontrolera znajduje się zestaw układów peryferyjnych wystarczających do przeprowadzenia prostych ćwiczeń. Przed przystąpieniem do realizacji ćwiczenia należy zainstalować kompilator Keil.

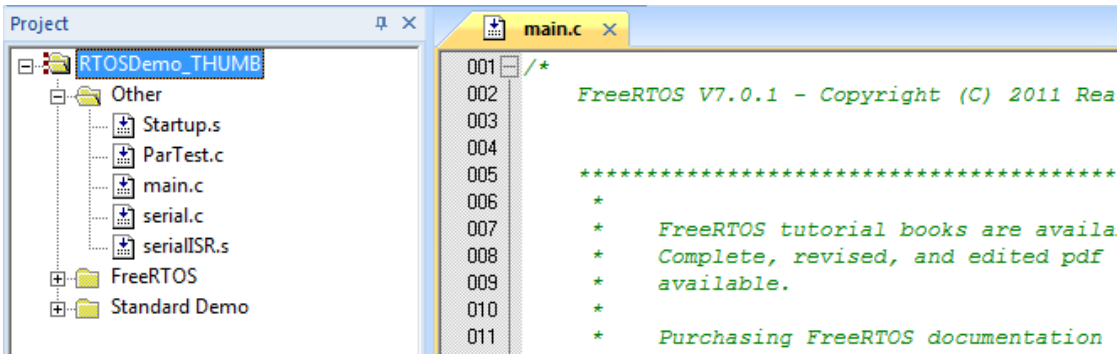
2.2 Tok postępowania

1. Z oficjalnej strony FreeRTOS *www.freertos.org* należy pobrać aktualną wersję systemu.
2. Po ściągnięciu i rozpakowaniu archiwum widoczna jest następująca struktura katalogów:

```
FreeRTOS
¦
+-Demo      - aplikacje demonstracyjne
+-License   - informacje o licencji
+-Source    - źródło systemu
+-TraceCon  - narzędzie do zapisu danych od planisty
```

3. W celu zmniejszenia wielkości projektu należy usunąć katalogi *License* oraz *TraceCon*. W folderze *Demo* należy usunąć wszystko oprócz katalogu: *ARM7_LPC2129_Keil_RVDS*. W folderze *Source\portable* usunąć wszystko oprócz katalogów *MemMang* i *RVDS*.

4. Należy wejść do katalogu *Demo\ARM7_LPC2129_Keil_RVDS* i uruchomić plik projektu *RTOSDemo.Uv2*. W lewym górnym rogu powinno być widoczne drzewo projektu zwierające pliki źródłowe systemu oraz programy demonstracyjne. W ćwiczeniu nie będą one wykorzystywane więc folder *Standard Demo* można usunąć. Pliki *ParTest.c, serial.c, serialISR.s* również nie będą wykorzystywane i powinny być skasowane. Struktura projektu systemu FreeRTOS przedstawiona została na rys. 2-1.



Rys. 2-1 Struktura projektu FreeRTOS

5. Należy uruchomić plik *main.c*, usunąć zawartość i wkleić następujący kod:

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

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

Listing 2-1 Szkielet programu sprawdzający poprawność przygotowania systemu FreeRTOS.

6. Należy skompilować projekt. Kompilacja powinna wykonać się bez błędów. Jest to tylko szkielet programu mający na celu sprawdzenie poprawności przygotowania systemu.

2.3 Konfiguracja systemu FreeRTOS

Przed przystąpieniem do pisania pierwszych programów w systemie FreeRTOS należy określić jego podstawowe właściwości. W tym celu, za pomocą notatnika lub innego edytora tekstów, należy uruchomić plik ***FreeRTOSConfig.h*** znajdujący się w folderze *Demo\ARM7_LPC2129_Keil_RVDS*. Jego fragment zamieszczono na Listing 2-2.

```
#define configUSE_PREEMPTION             0
#define configUSE_IDLE_HOOK              0
#define configUSE_TICK_HOOK              0
#define configCPU_CLOCK_HZ               ( ( unsigned long ) 60000000 )
#define configTICK_RATE_HZ               ( ( portTickType ) 1000 )
#define configMAX_PRIORITIES             ( ( unsigned portBASE_TYPE ) 4 )
#define configMINIMAL_STACK_SIZE         ( ( unsigned short ) 90 )
#define configTOTAL_HEAP_SIZE            ( ( size_t ) 13 * 1024 )
#define configMAX_TASK_NAME_LEN          ( 8 )
#define configUSE_TRACE_FACILITY         0
#define configUSE_16_BIT_TICKS           0
#define configIDLE_SHOULD_YIELD          1
#define configQUEUE_REGISTRY_SIZE        0
#define configUSE_CO_ROUTINES            0
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )
#define INCLUDE_vTaskPrioritySet         1
#define INCLUDE_uxTaskPriorityGet        1
#define INCLUDE_vTaskDelete              1
#define INCLUDE_vTaskCleanUpResources    0
#define INCLUDE_vTaskSuspend             1
#define INCLUDE_vTaskDelayUntil          1
#define INCLUDE_vTaskDelay               1
```

Listing 2-2 Fragment pliku konfiguracyjnego systemu FreeRTOS.

Pierwsza część makrodefinicji ustala parametry pracy samego systemu, a druga włącza wybrane funkcje API dostępne w systemie. Zgodnie z opisem w pliku ***FreeRTOSConfig.h*** ustawienie wartości danego makra na 1 włącza poszczególne funkcje, 0 wyłącza. Najważniejsze makra systemu to [2]:

- `configUSE_PREEMPTION` – uruchomienie wywłaszczeń systemu (preemption),
- `configUSE_IDLE_HOOK` – włączenie obsługi własnej funkcji *vApplicationIdleHook(),* wywoływanej podczas bezczynności systemu,
- `configCPU_CLOCK_HZ` – częstotliwość pracy mikrokontrolera,
- `configTICK_RATE_HZ` – częstotliwość pracy systemu (domyślnie 1 kHz),
- `configMAX_PRIORITIES` – maksymalna liczba priorytetów,
- `configMINIMAL_STACK_SIZE` – minimalny rozmiar stosu,
- `configMAX_TASK_NAME_LEN` – maksymalna długość nazwy zadania.

Ponieważ częstotliwość pracy mikrokontrolera domyślnie dzielona jest na 4 (rejestr `VPBDIV` domyślnie wyzerowany), wartość makra `configCPU_CLOCK_HZ` należy zmienić na 15000000. Pozostałe makra powinny posiadać wartości domyślne. Należy pamiętać także o zapisaniu pliku.

3 Ćwiczenia

W tej części pracy opisane zostały przykładowe programy oraz związane z nimi zadania do samodzielnego wykonania, które pozwalają lepiej zrozumieć możliwości systemu FreeRTOS. Autor rozpoczyna ćwiczenia od prostych wielowątkowych programów operujących głownie na kilku diodach LED. W kolejnych etapach sukcesywnie zwiększana jest trudność ćwiczeń, których głównymi tematami jest komunikacja międzywątkowa, synchronizacja oraz obsługa przerwań. Do wykonania ćwiczeń wymagane są moduły zawierające funkcje i definicje do obsługi podstawowych peryferiów oraz instrukcja do ćwiczeń laboratoryjnych z przedmiotu „*Programowanie mikrokontrolerów ARM w języku C/C++*” [5] nazywanego dalej *Kursem Podstawowym*.

3.1 Wielowątkowość

3.1.1 Uruchamianie wątków w FreeRTOS

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);
            }
}
```

Listing 3-1 Pulsowanie diody 0 z częstotliwością 1 Hz.

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.

Rozwiązanie zad. 1:

Listing 3-2 Pulsowanie diodami z częstotliwością 1 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(Led0Blink(),..)`. 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()`),

4. Przechodzi w pętlę nieskończoną tylko w przypadku niewystarczającej ilości pamięci wymaganej do utworzenia wątku "*idle task*" aktywowanego w trakcie bezczynności systemu.

```
#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);
}
```

Listing 3-3 Tworzenie wątków w systemie FreeRTOS.

Zadanie 2: Zamienić obecny wątek wątkiem pulsującym diodą 0 z częstotliwością 0.5 Hz.

Uwaga: nie usuwać funkcji obecnego wątku.

Rozwiązanie zad. 2:

Listing 3-4 Wątek pulsujący diodą 0 z częstotliwością 0.5 Hz.

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

3.1.2 Wykorzystanie opóźnienia systemowego

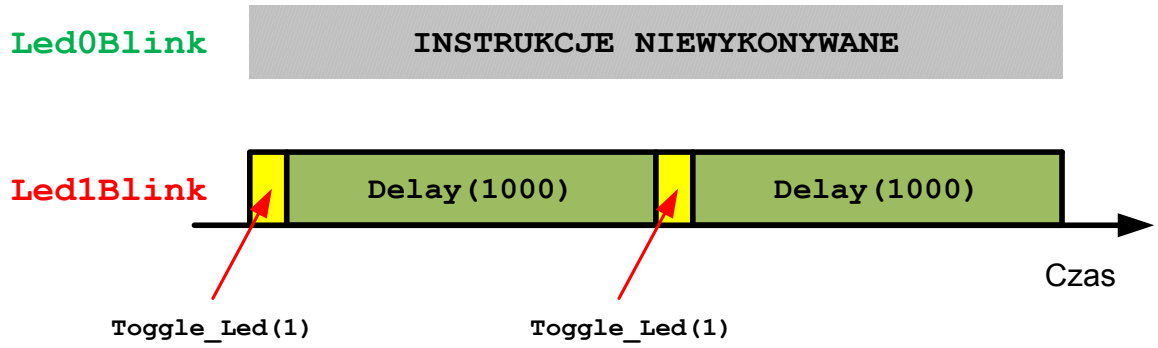
Zadanie 1: Dodać do programu z poprzedniego zadania wątek pulsujący diodą 1 z częstotliwością 10 Hz. Zaobserwować działanie programu dla różnych kolejności tworzenia wątków.

Podpowiedź: Stworzyć funkcje `Led1Blink` i użyć powtórnie funkcji `xTaskCreate()`.

Rozwiązanie zad. 1:

Listing 3-5 Dodatkowy wątek pulsujący diodą 1 z częstotliwością 10 Hz.

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. 3-1 Blokowanie wątku Led0Blink spowodowane użyciem pętli opóźniającej.

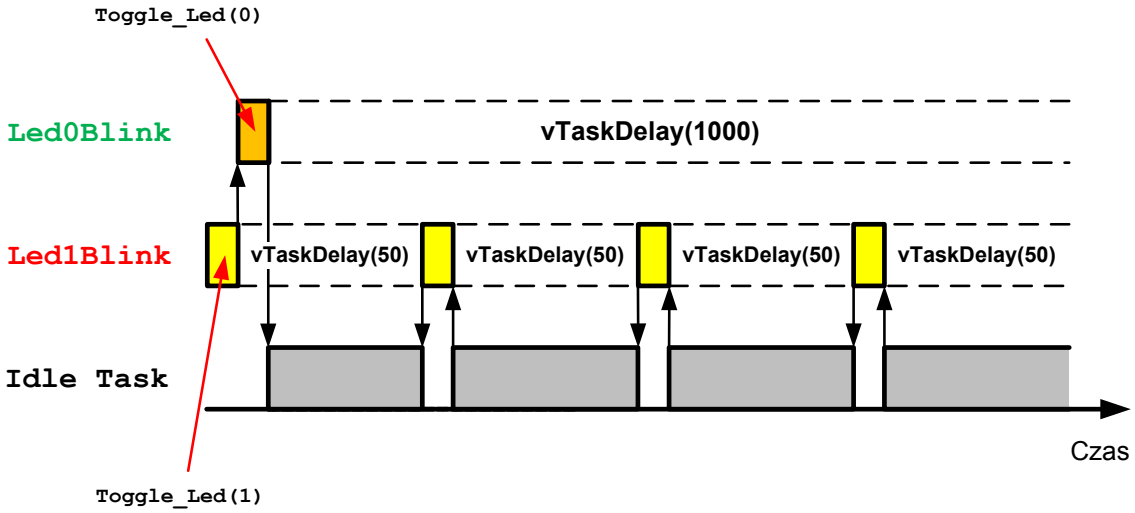
Rozwiązaniem tego problemu jest wykorzystanie funkcji opóźniającej `vTaskDelay()` dostarczanej wraz z RTOS (tzw. systemowej).

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

Rozwiązanie zad. 2:

Listing 3-6 Użycie funkcji systemowej do generowania opóźnień.

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. 3-2 Wykorzystanie funkcji vTaskDelay() do tworzenia nieblokujących opóźnień.

3.1.3 Równoległe wykonywanie zadań

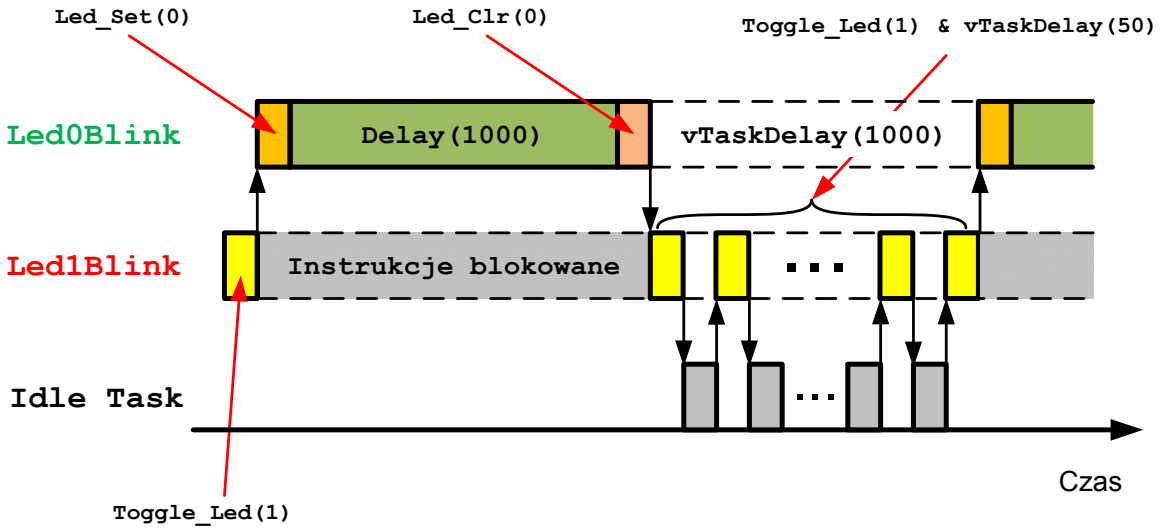
Wprowadzenie: Powyższe rozwiązanie działa poprawnie (zadania wykonywane są równolegle) jeżeli zadania wykonywane w wątkach są stosunkowo krótkie. Może jednak zaistnieć sytuacja w której zadanie w jednym z wątków rzeczywiście potrzebuje więcej czasu na wykonanie. Ilustruje to program z poniższego zadania.

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.

Rozwiązanie zad. 1:

Listing 3-7 Symulacja czasochłonnego zadania.

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



Rys. 3-3 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 – ustawić `"configUSE_PREEMPTION 1"` w pliku konfiguracyjnym ***...\Demo\ARM7_LPC2129_Keil_RVDS\FreeRTOSConfig.h***, ustawić niższy priorytet dla wątku pulsującego z wyższą częstotliwością – drugi od końca argument funkcji `xTaskCreate()`, (większy numer wyższy priorytet), zaobserwować działanie programu.

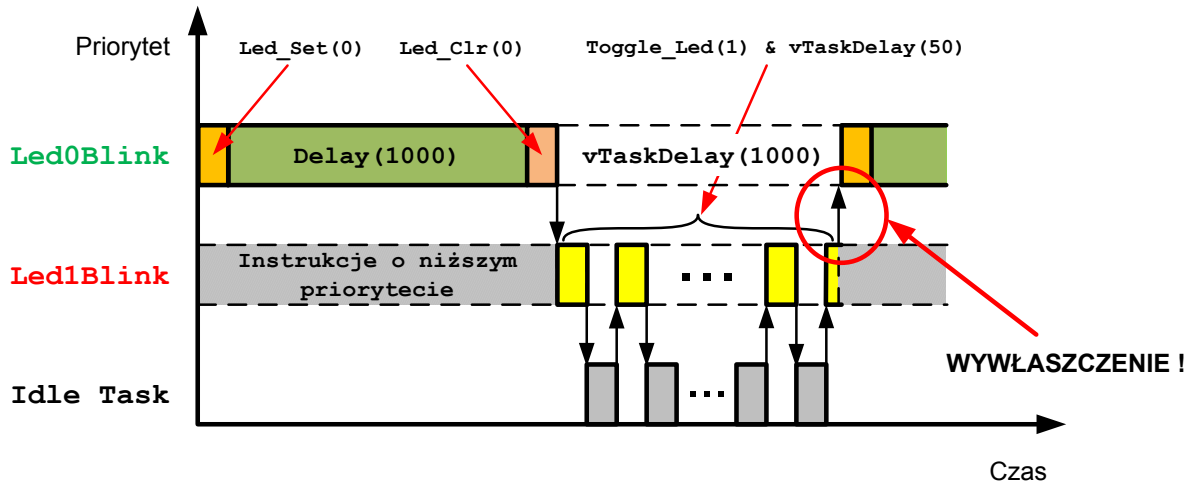
Rozwiązanie zad. 2:

Listing 3-8 Zmiana priorytetu jednego z wątków.

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.



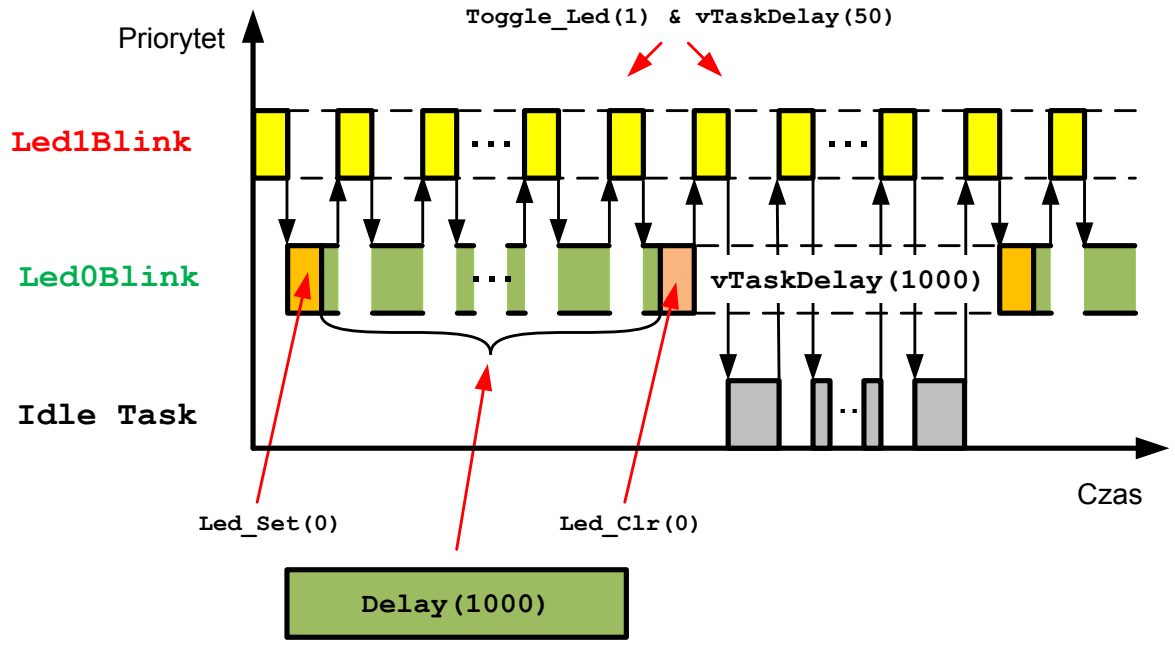
Rys. 3-4 Wywłaszczenie wątku o niższym priorytecie.

Zadanie 3: Odwrócić priorytety wątków i zaobserwować działanie programu.

Rozwiązanie zad. 3:

Listing 3-9 Zamiana priorytetów wątków.

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. 3-5 Wywłaszczanie czasochłonnych instrukcji przez wątek o wyższym priorytecie.

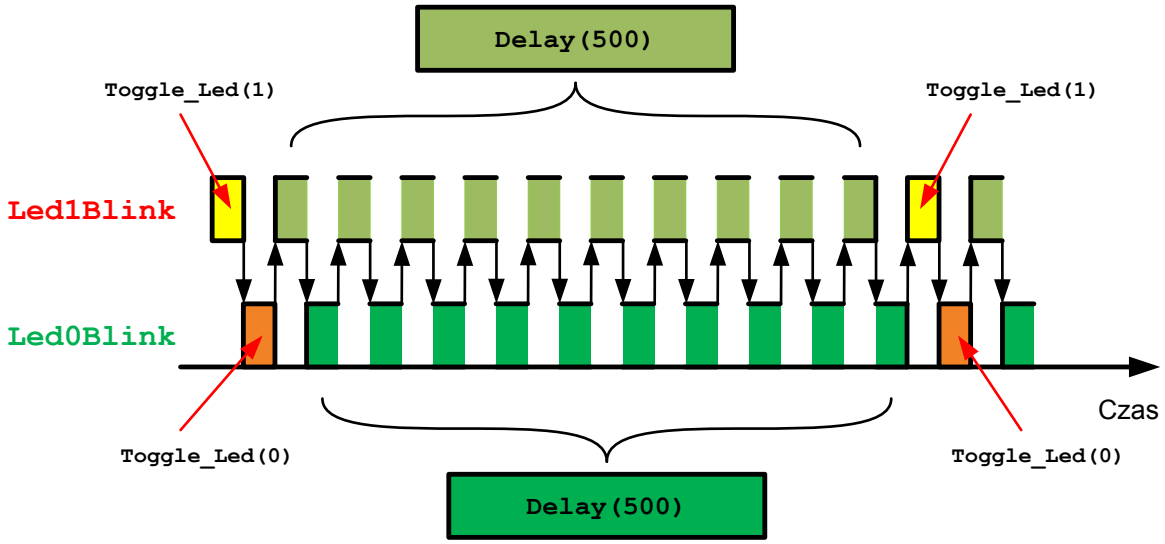
Wprowadzenie: Można sobie wyobrazić sytuację w której oba wątki muszą równocześnie wykonywać czasochłonne zadanie. Pokazuje to program z Zadania 4.

Zadanie 4: Zmodyfikować program z poprzedniego zadania tak, aby oba watki pulsowały diodą z okresem 1Hz i używały tylko opóźnienia programowego (`Delay(500)`). Ustawić równe priorytety obu wątków.

Rozwiązanie zad.4:

Listing 3-10 Ustawienie równych priorytetów dla czasochłonnych 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. 3-6 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.

3.2 Kontrola wątków

3.2.1 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);
}
```

Listing 3-11 Uruchamianie wątków z parametrem.

Opis działania:

`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 miało by 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`).

Rozwiązanie zad. 2:

Listing 3-12 Uruchamianie 2 wątków ze wspólnym parametrem.

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).

Rozwiązanie zad. 3:

Listing 3-13 Przekazywanie do wątków parametru w postaci struktury.

3.2.2 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.

Rozwiązanie zad. 1:

Listing 3-14 Cykliczne zawieszanie i odwieszanie wątku.

Bibliografia

[1] Strona internetowa: *www.freertos.org*

[2] Richard Barry, Using The FreeRTOS Real Time Kernel - A Practical Guide, 2009.

[3] Krzysztof Paprocki, Praca pod kontrolą FreeRTOS, Elektronika Praktyczna, 5/2009.

[4] Atmega32 User Manual.

[5] Instrukcja do ćwiczeń laboratoryjnych z przedmiotu „Programowanie mikrokontrolerów ARM w języku C/C++".

[6] Strona internetowa: *www.atmel.com*