System operacyjny

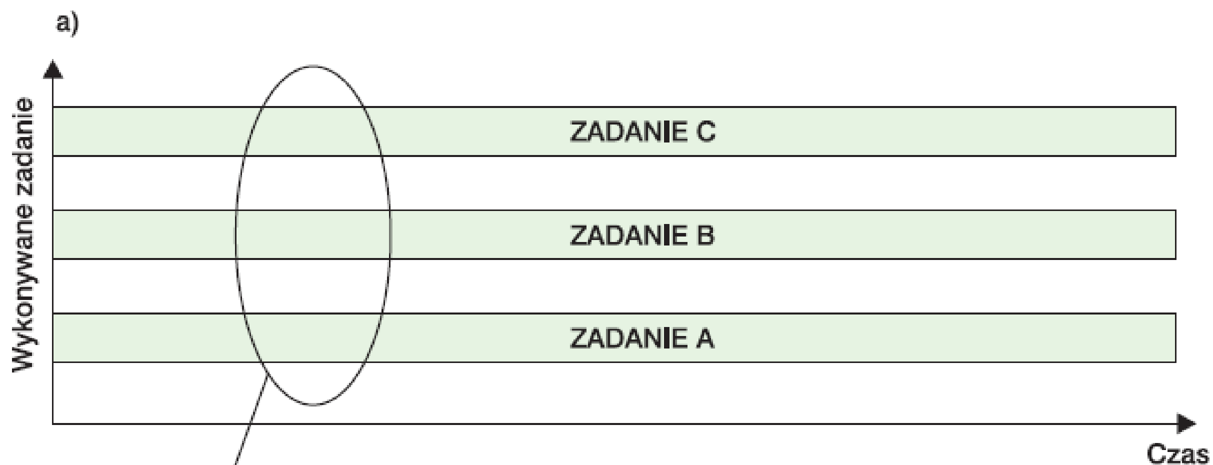
Zadaniem każdego **systemu operacyjnego** jest zarządzanie sprzętem w taki sposób, aby zadania stawiane przez użytkownika były wykonane przy optymalnym wykorzystaniu zasobów i mocy obliczeniowej.

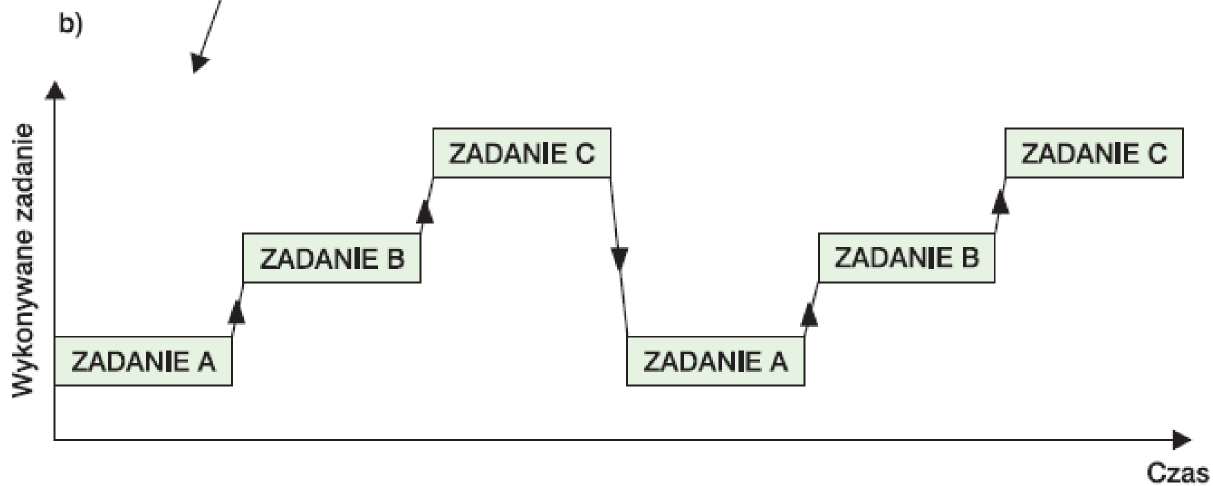
Z systemami operacyjnymi stykamy się praktycznie codziennie używając komputerów osobistych. Współczesne komputery mają bardzo duże zasoby sprzętowe i szybkie, wielordzeniowe mikroprocesory.

Pracując z komputerem często korzysta się z wielu programów pracujących jednocześnie. Można mieć włączony edytor tekstu, przeglądarkę internetową, odtwarzacz muzyki itp.

Z punktu widzenia sprzętu, system operacyjny ma do dyspozycji jeden procesor (współcześnie stosowane są procesory wielordzeniowe, ale w tym opisie użyjemy uproszczonej, jednordzeniowej, wersji), a zatem jak to się dzieje, że tyle zadań realizowanych jest równocześnie?

Na rysunku przedstawiono diagram pracy wielozadaniowego OS z uruchomionymi jednocześnie kilkoma zadaniami.


a)
Wykonywane zadanie
ZADANIE C
ZADANIE B
ZADANIE A
Czas


b)
Wykonywane zadanie
ZADANIE A
ZADANIE B
ZADANIE C
ZADANIE A
ZADANIE B
ZADANIE C
Czas

Na rysunku a pokazano, że wszystkie trzy zadania są pozornie wykonywane jednocześnie.

Dopiero rysunek b, który przedstawia w dużym powiększeniu zachowanie się procesów pokazuje, że każde zadanie wykonywane jest tylko przez pewien czas, a pozostałe są wstrzymane i czekają na swoją kolej.

Jest to najprostsza forma systemu operacyjnego, gdzie kolejkowanie odbywa się przy pomocy algorytmu karuzelowego, omówionego w dalszej części.

Każde zadanie uruchomione w systemie otrzymuje do swojej dyspozycji procesor tylko na pewien czas.

Jeśli ten czas będzie dostatecznie krótki, czyli przełączanie pomiędzy uruchamianymi zadaniami będzie dostatecznie szybkie, to użytkownik korzystający z komputera odniesie wrażenie, że wszystkie jego programy pracują jednocześnie.

Wymagania stawiane systemom wbudowanym wymusiły powstanie systemów operacyjnych **czasu rzeczywistego (RTOS – Real Time Operating System)**.

W przeciwieństwie do dużych komputerów, w systemie mikroprocesorowym często zdarza się, że wykonanie danego zadania musi mieścić się w ściśle określonym przedziale czasu.

Ma to znaczenie m.in. dla procesów przemysłowych, ale przede wszystkim wszędzie tam, gdzie od maszyn zależy życie i bezpieczeństwo ludzi.

RTOSy dzielą się dodatkowo na grupy, w zależności od tego, w jakim stopniu jest się w stanie ocenić czas wykonywania operacji.

Z tej perspektywy można rozróżnić **miękkie i twarde systemy operacyjne czasu rzeczywistego**.

Pierwsze pozwalają określić czas wykonywania zadania z pewną dokładnością, natomiast drugie mają do spełnienia najwyższe wymagania i muszą być całkowicie przewidywalne.

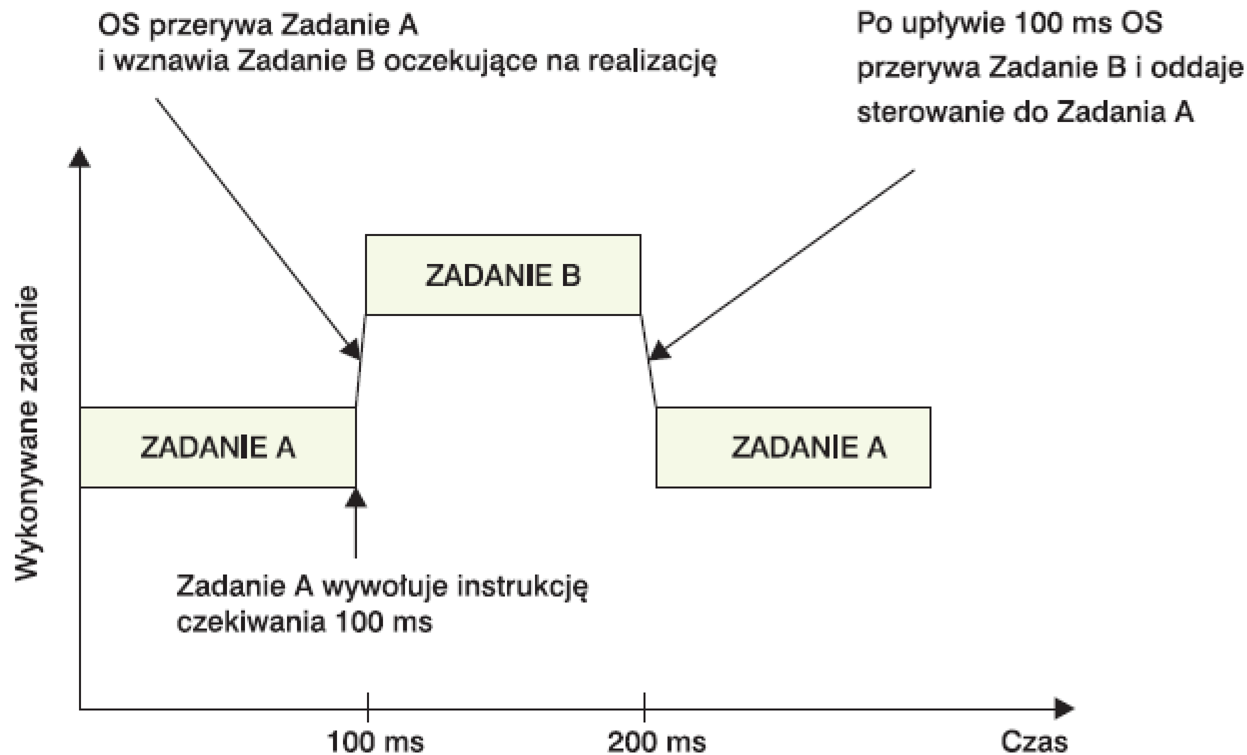
System operacyjny czasu rzeczywistego z wywłaszczeniami zadań

Zazwyczaj schemat pracy systemów mikroprocesorowych polega na wykonaniu instrukcji przez jakąś funkcję (zadanie) i czekaniu na wystąpienie zdarzenia.

Podczas oczekiwania na zdarzenie lub upłynięcie określonego czasu, mikrokontroler nie robi nic, poza wykonywaniem pustych instrukcji.

W układach z wbudowanym systemem operacyjnym w czasie oczekiwania na zdarzenie będą uruchomione pozostałe, oczekujące na czas procesora, zadania. Sytuację taką przedstawiono na rysunku.

Nie można opisanego powyżej zachowania wprost nazwać wywłaszczeniem, ponieważ w tym przypadku przerywane zadanie tak naprawdę nie jest wstrzymywane, bo i tak bezproduktywnie oczekuje na jakieś zdarzenie lub po prostu odlicza określony czas.

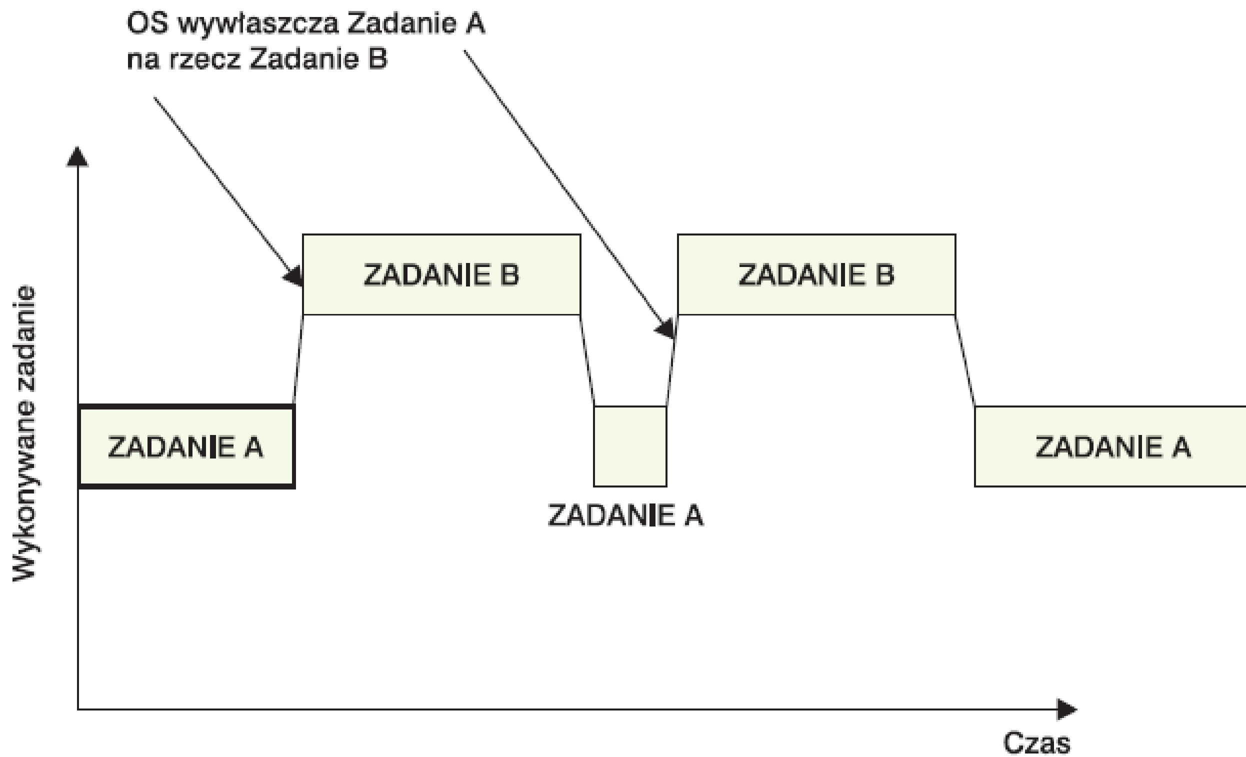

OS przerywa Zadanie A
i wznawia Zadanie B oczekujące na realizację
Po upływie 100 ms OS
przerywa Zadanie B i oddaje
sterowanie do Zadania A
ZADANIE B
ZADANIE A
ZADANIE A
Wykonywane zadanie
Zadanie A wywołuje instrukcję
czekiwania 100 ms
100 ms
200 ms
Czas

Rysunek ilustruje przerwanie ZADANIA A w momencie jego oczekiwania, aż upłynie czas 100 ms.

W tym czasie RTOS oddaje procesor do dyspozycji ZADANIA B, o niższym priorytecie, niż ZADANIE A.

Można sobie to wyobrazić w taki sposób, że ZADANIE A zajmuje się sterowaniem procesu wykonawczego (np. silnika) i tutaj nie mogą wystąpić fluktuacje czasu, natomiast ZADANIE B zajmuje się interfejsem użytkownika, który nie ma narzuconych krytycznych ram czasowych i może być wykonywane „w wolnych chwilach” procesora.

Z prawdziwymi wywłaszczeniami (preemptive) zadań mamy zaś do czynienia wtedy, gdy system operacyjny zachowuje się podobnie, jak to przedstawiono na kolejnym rysunku,


OS wywłaszcza Zadanie A
na rzecz Zadanie B
Wykonywane zadanie
ZADANIE A
ZADANIE B
ZADANIE A
ZADANIE B
ZADANIE A
Czas

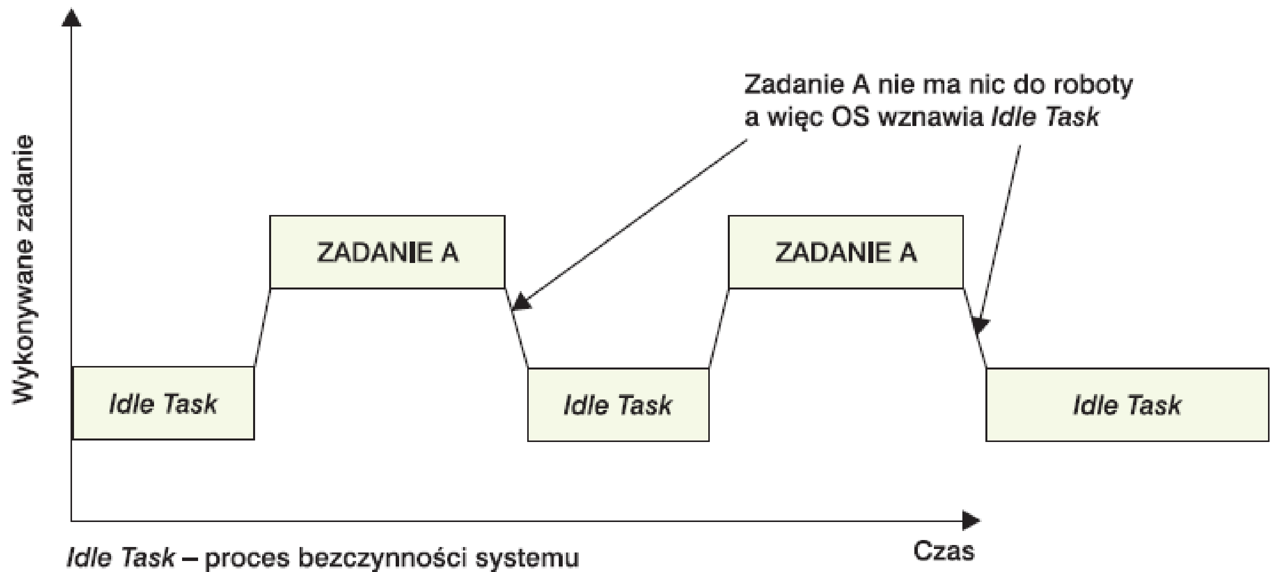
a więc wtedy, gdy obsługiwane zadanie jest przerywane podczas pracy na rzecz innego zadania o wyższym priorytecie, nie tolerującego zwłoki w obsłudze.

W czasie pracy mikrokontroler może nie mieć żadnych zadań do obsługi, z tego powodu w systemie operacyjnym występuje ważny proces nazywany procesem bezczynności systemu – **Idle task**.

Idle task jest uruchamiany zawsze wtedy, gdy system nie ma żadnych zadań do realizacji.

Proces bezczynności systemu zajmuje się czyszczeniem pamięci po zakończonych zadaniach, oraz, gdy faktycznie nie ma nic do zrobienia, może wprowadzać tryby obniżonego poboru mocy.

Zazwyczaj priorytet tego zadania jest najniższy z możliwych.



Wykonywane zadanie
Zadanie A nie ma nic do roboty
a więc OS wznawia Idle Task
ZADANIE A
ZADANIE A
Idle Task
Idle Task
Idle Task
Czas
Idle Task – proces bezczynności systemu

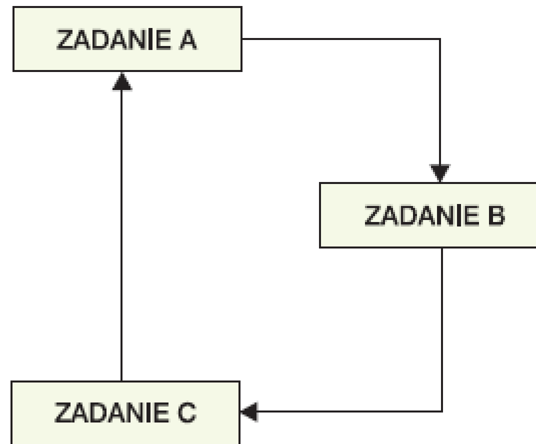
Algorytm szeregowania

Algorytm szeregowania (scheduler – planista), jest częścią jądra systemu i ma za zadanie optymalnie rozdzielić czas procesora oraz zasoby systemu mikroprocesorowego pomiędzy uruchomione zadania.

Uwzględnia przy tym priorytety zadań i wymagania związane z czasem ich realizacji.

Implementacja dobrego algorytmu szeregowania nie należy do łatwych, a jego skomplikowanie rośnie wraz z wymaganiami stawianymi przed systemem operacyjnym.

Generalnie można rozróżnić kilka podstawowych rodzajów algorytmów szeregowania. Najprostszym jest algorytm karuzelowy, którego zasadę działania przedstawiono na rysunek.



Nazwa tego rodzaju planisty bardzo dobrze oddaje jego faktyczne zachowanie.

Zadania wykonywane są w ustalonej kolejności, po czym proces się powtarza, czyli mamy swoistą karuzelę.

W systemach operacyjnych czasu rzeczywistego algorytm karuzelowy w swej czystej postaci nie jest specjalnie użyteczny.

Zgodnie z tym, co zostało już wcześniej napisane, w RTOS krytyczne zadania nie mogą czekać na swoją kolejkę, tylko muszą być wykonywane natychmiast, czyli muszą wywłaszczać procesy o niższych priorytetach.

System operacyjny FreeRTOS

FreeRTOS to system operacyjny czasu rzeczywistego z wywłaszczeniami.

Jest to bezpłatny system open source i może być wykorzystywany w aplikacjach komercyjnych.

Wszystkie niezbędne informacje do rozpoczęcia pracy z tym systemem, aktualna wersja do pobrania oraz forum dyskusyjne są dostępne na stronie internetowej www.freertos.org.

W swojej podstawowej konfiguracji system FreeRTOS składa się z trzech plików źródłowych, które są wspólne dla wszystkich architektur mikrokontrolerów.

Te pliki to: **tasks.c, queue.c, list.c**.

Powyższe pliki to jądro systemu operacyjnego.

Do poprawnej pracy w docelowym mikrokontrolerze wymagane są jeszcze pliki programów zapewniających komunikację pomiędzy sprzętem, a jądrem systemu.

W sumie przygotowanych jest prawie 20 dystrybucji tego systemu (każda z przykładową aplikacją).

Wersje systemu FreeRTOS opracowane są dla: Atmel AVR, STM32, NXP (LPC2106, LPC2124, LPC2129), Microchip (PIC18, PIC24, dsPIC, PIC32), Freescale (Cold-Fire), Xilinx (Microblaze, PowerPC – „miękkie" procesory do implementacji w układach FPGA), Texas Instruments (MSP430), LuminaryMicro (LM3Sxxxx – rdzeń Cortex M3).

Argumentem przemawiającym za stosowaniem w swoich aplikacjach systemu FreeRTOS jest dostępność również jego wersji komercyjnych (OpenRTOS i SafeRTOS).

Dzięki temu, jeśli projekt stanie się bardziej wymagający, można skorzystać z systemów operacyjnych z pełnym wsparciem i certyfikatami bezpieczeństwa.

System FreeRTOS jest w pełni skalowalny, co oznacza, że można dopasowywać jego stopień zaawansowania do wymagań projektu.

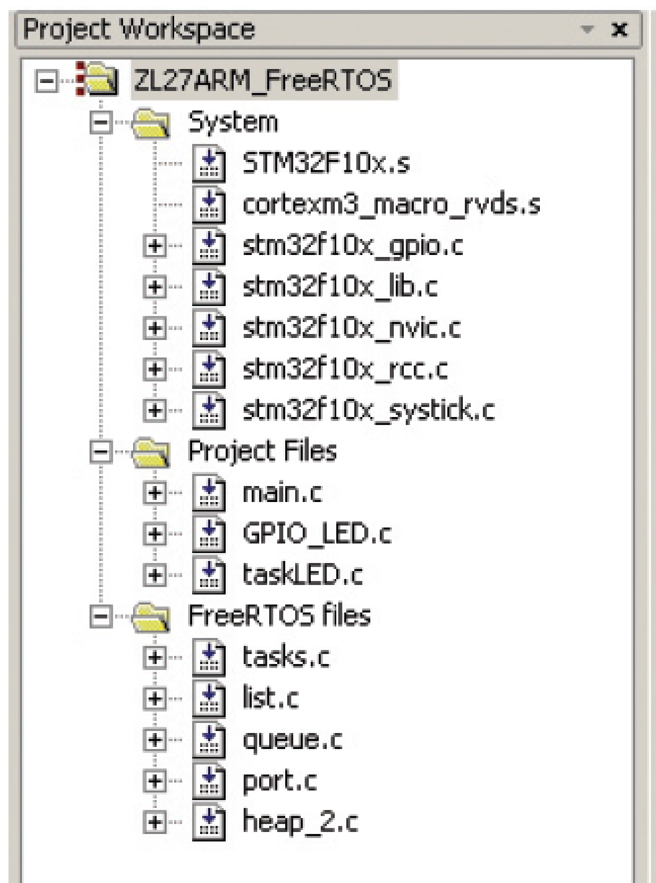
Struktura plików systemu FreeRTOS

Wszystkie projekty wykorzystujące system operacyjny FreeRTOS mają podobną strukturę plików, a przynajmniej programista powinien dążyć do jej standaryzacji.

Powyższa filozofia pisania aplikacji dotyczy oczywiście wszystkich prac związanych z jakimkolwiek dojrzałym projektowaniem.

Standaryzacja działań projektowych zawsze pozwala zaoszczędzić mnóstwo czasu i pieniędzy.

Na rysunku zamieszczono zrzut ekranowy przedstawiający drzewo plików projektu wykorzystującego system FreeRTOS.


Project Workspace
ZL27ARM_FreeRTOS
System
STM32F10x.s
cortexm3_macro_rvds.s
stm32f10x_gpio.c
stm32f10x_lib.c
stm32f10x_nvic.c
stm32f10x_rcc.c
stm32f10x_systick.c
Project Files
main.c
GPIO_LED.c
taskLED.c
FreeRTOS files
tasks.c
list.c
queue.c
port.c
heap_2.c

System operacyjny FreeRTOS w swej podstawowej formie jest niezależny od sprzętu.

Aby jądro systemu mogło nawiązać współpracę z danym mikrokontrolerem należy mu zaimplementować interfejs.

We wszystkich przykładach, dostępnych na stronie domowej systemu FreeRTOS, kod zapewniający poprawną pracę z daną architekturą umieszczono w pliku **port.c**.

Również we własnych projektach i przy tworzeniu nowej dystrybucji FreeRTOSa na nową platformę, należy stosować ten sam standard.

Zadaniem kodu w pliku port. c jest konfiguracja do pracy oraz obsługa przerwań i wyjątków systemowych, które są ściśle związane z architekturą rdzeni Cortex M3 (np. timer SysTick, wyjątek PendSV).

Zasada działania systemu FreeRTOS. Zadania (Tasks) i współprogramy (Co-routines)

Procesy w systemie FreeRTOS mogą być realizowane na dwa sposoby: za pomocą **zadań (task)** lub **współprogramów (Co-routines)**.

Standardowo, zadania wykorzystywane są w systemach operacyjnych czasu rzeczywistego.

Są to zupełnie niezależne procesy. Każde zadanie posiada swój własny kontekst, czyli z perspektywy takiego zadania, wszystkie rejestry i stos należą tylko do niego. O tym, które zadanie jest wykonywane w danym momencie, decyduje algorytm szeregowania (scheduler). Planista kontroluje uruchomione zadania, przerywa je i wznawia, a system operacyjny dba o przełączanie kontekstów. Każde zadanie posiada swój własny stos, którego zawartość umieszczona jest w pamięci RAM.

Współprogramy działają podobnie do zadań, choć są pomiędzy nimi istotne różnice. Współprogramy dzielą jeden stos, a co za tym idzie, do swego działania potrzebują mniejszej ilości pamięci RAM.

Ponadto, ważne jest, że zadania i współprogramy nie mogą się komunikować między sobą za pomocą kolejek i semaforów, a zadania zawsze są ważniejsze od

współprogramów.

Każde z zadań występujące w systemie FreeRTOS ma określony stan.

W sumie wyróżniamy cztery możliwe stany, w których zadanie może być:

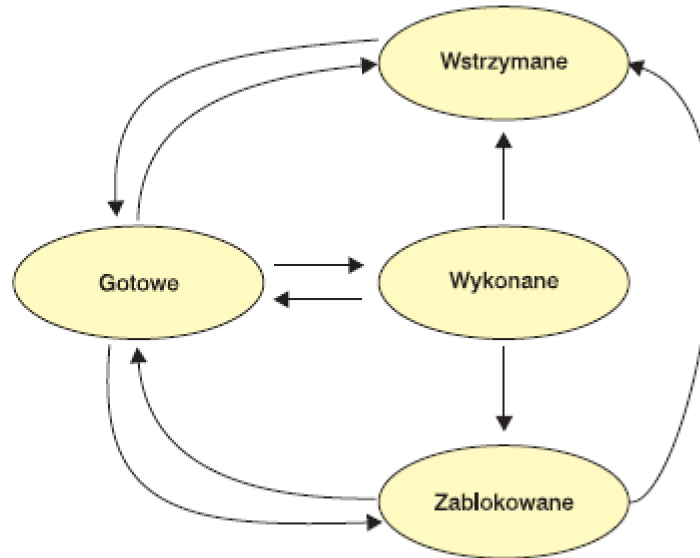
– **Wykonywane (Running)**, zadanie aktualnie korzysta z zasobów mikrokontrolera.

– **Gotowe do wykonywania (Ready)**, może być wykonywane, ale czeka na zwolnienie zasobów przez inne zadanie.

– **Zablokowane (Blocked)**, zadanie czeka na zdarzenie, przykładowo upłynięcie zadanego czasu lub zewnętrzne przerwanie.

– **Wstrzymane (Suspended)**, zadanie, które zostało wstrzymane nie jest uwzględniane przez planistę, ale może być wznowione.

Możliwe przejścia pomiędzy wszystkimi
stanami przestawiono na rysunku.



Zadania mają przypisany priorytet, przy czym priorytet 0 jest domyślnie nadany procesowi bezczynności systemu (Idle task).

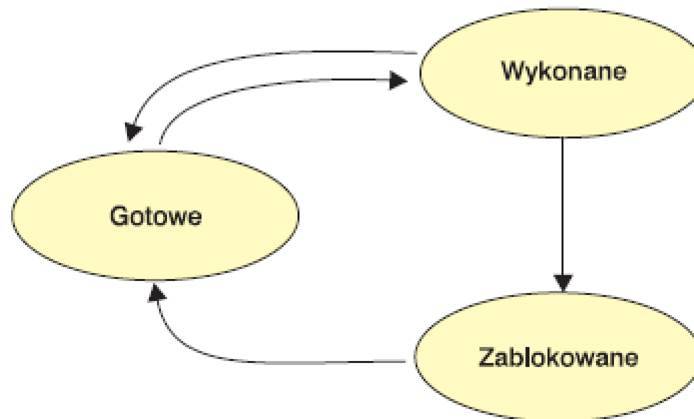
Współprogramy mają trzy dozwolone stany:

– Gotowe do wykonywania (Ready).

– Wykonywane (Running).

– Zablokowane (Blocked).

Diagram przejść pomiędzy powyższymi stanami został zamieszczono na rysunku.



Konstrukcja i uruchamianie zadania w systemie FreeRTOS

Każde zadanie w systemie operacyjnym FreeRTOS jest zazwyczaj funkcją, która musi być napisana według określonych standardów.

Przykładowy fragment kodu zawierający puste zadanie zamieszczono na listingu.

```
void vTaskX(void * pvParameters)
{
    for(;;)
    {
    // Tutaj kod realizowanego zadania
    }
}
```

Zarówno wartość zwracana przez funkcję, jak i argument są typu void.

Argument przekazywany do funkcji zadania może służyć do przekazywania informacji każdego typu.

Ważnym elementem konstrukcji funkcji zadania jest jego nieskończoność. Innymi słowy, kod zadania musi być umieszczony w pętli nieskończonej for lub while, czyli raz wywołane zdanie nigdy samoczynnie nie powróci do miejsca swojego wywołania.

Zadania tworzone są za pomocą funkcji **xTaskCreate()**, a usuwane po wywołaniu funkcji **vTaskDelete()**.

Literka v przed nazwą funkcji oznacza, że nie zwraca ona żadnej wartości.

Rzecz jasna, funkcje tworzenia i usuwania zadania muszą mieć przekazane odpowiednie parametry. Sposób tworzenia, a następnie usuwania zadania przestawiono na list. 2.

```
void vStartLEDTasks(unsigned portBASE_TYPE uxPriority)
{
     xTaskHandle xHandleTaskLED;
     // Tworzenie zadania
     xTaskCreate(vTaskLED, ( signed portCHAR * ) „LED”,
     STACK_SIZE, NULL, uxPriority, &xHandleTaskLED);
     // Usuwanie zadania
     vTaskDelete(xHandleTaskLED);
}
```

Proces bezczynności systemu (Idle task) jest tworzony automatycznie przez algorytm szeregowania, a więc nie wymaga jawnych operacji włączania.

Komentarza wymaga lista argumentów funkcji tworzącej zadanie xTaskCreate().

Licząc od lewej strony, najpierw przekazujemy nazwę funkcji zadania, w tym przykładzie jest to vTaskLED.

Kolejnym argumentem jest nazwa zadania, która pozwala je zidentyfikować.

Następnie określamy rozmiar stosu, czy będą przekazane jakieś parametry (NULL= brak), priorytet, a na końcu uchwyt do zadania, który może być dalej wykorzystywany do sterowania jego pracą.

Usunięcie zadania ogranicza się do wywołania funkcji vTaskDelete() z uchwytem do zadania w argumencie.

Podstawowe sterowanie zadaniami

Poza tworzeniem i usuwaniem zadań, dobrze by było, gdyby system udostępniał mechanizmy pozwalające na wprowadzanie opóźnień czasowych, czy też wstrzymywanie i wznawianie wykonywania zadania.

System operacyjny FreeRTOS udostępnia kilka funkcji API, pozwalających na sterowanie i kontrolę nad zadaniami.

W celu wprowadzenia opóźnień w wykonywanym zadaniu mogą być użyte dwie funkcje: **vTaskDelay()** i **vTaskDelayUntil()**.

Funkcja vTaskDelay() przyjmuje jeden argument, który jest liczbą taktów zegara systemu operacyjnego, na jaką dane zadanie zostanie zablokowane.

Czym dokładnie są takty systemu operacyjnego, to będzie opisane w dalszej części , przy okazji przedstawiania pliku konfiguracyjnego systemu FreeRTOS. Tutaj wystarczy wiedzieć, że jest to najmniejszy kwant czasu rozróżniany przez OS.

Istotne jest, że czas zablokowania jest ściśle związany z częstotliwością zegara systemowego i zmiana tego parametru może spowodować zmiany opóźnień.

W celu zabezpieczenia się przed taką ewentualnością, do wyznaczenia liczby taktów wykorzystuje się całkowite dzielenie przez stałą portTICK_RATE_MS, która pozwala obliczyć opóźnienia z dokładnością do jednego taktu.

```
void vTaskLED(void * pvParameters)
{
    // Nieskonczona petla zadania
    for(;;)
    {
    // Wprowadzenie opoznienia 500ms
    vTaskDelay(500/portTICK_RATE_MS);
    // Zmiana stanu wyprowadzenia PC6 (LD1) na przeciwny
    vhToggleLED();
    }
}
```

Na listingu przedstawiono zadanie, które zajmuje się cykliczną zmianą stanu wyprowadzeń mikrokontrolera, gdzie w roli funkcji opóźnienia wykorzystano vTaskDelay().

Fizyczną zmianą stanu wyprowadzenia zajmuje się poniższa funkcja vhToggleLED():

```
void vhToggleLED(void)
{
    // Zamiana stanu wyprowadzenia PC6 na przeciwny
    GPIO_WriteBit(GPIOC, GPIO_Pin_6, (BitAction) ((1-
    GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_6))));
}
```

Tworzenie nowej funkcji dla jednej linii kodu jest uzasadnione tym, że projekt powinien być tworzony w sposób warstwowy.

Przedrostek nazwy funkcji zawierający literkę h nie jest przypadkowy, a oznacza, że funkcja działa wprost na sprzęcie.

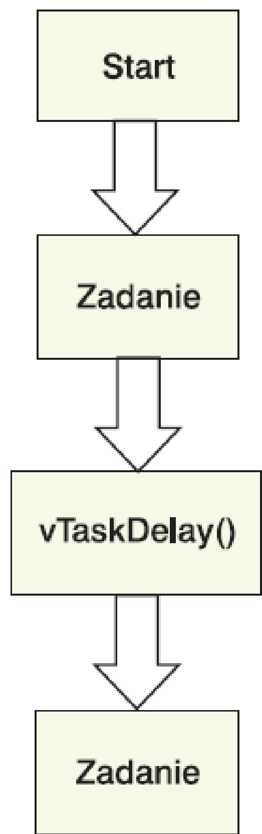
Takie podejście doskonale wpływa na czytelność projektu i zabezpiecza przed przypadkowym odwołaniem się do urządzeń peryferyjnych, ponieważ wiadomo, że tylko funkcje z przedrostkiem h mogą to robić.

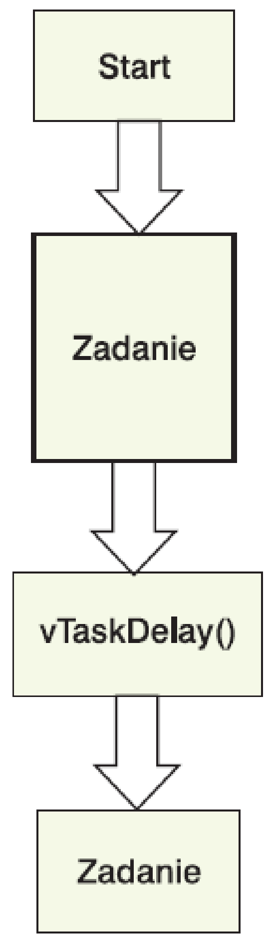
Generalnie, wykorzystywanie funkcji vTaskDelay() do sterowania pracą zadań cyklicznych, w szczególności takich, w których opóźnienia (częstotliwość) muszą być dokładne, nie jest dobrym pomysłem.

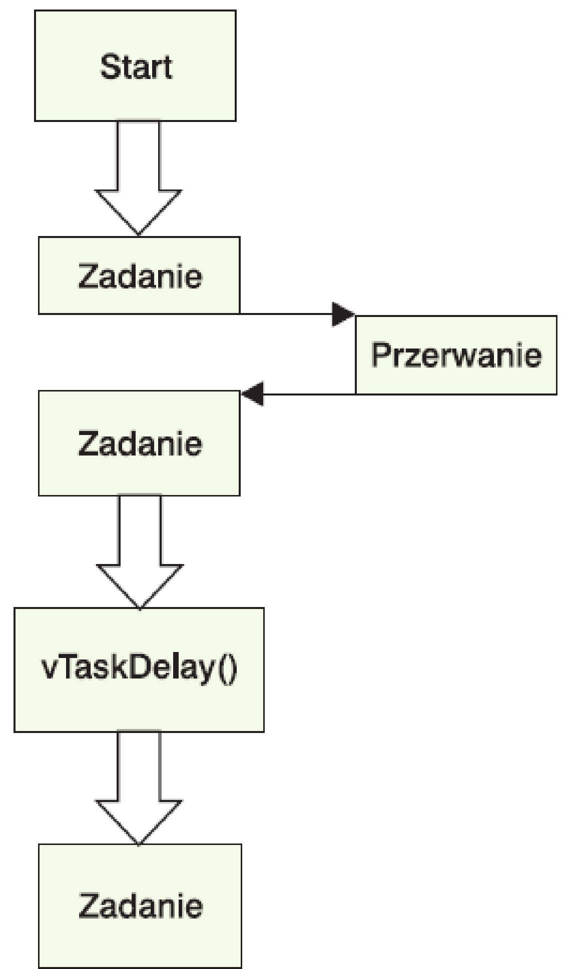
Omawiana funkcja nie gwarantuje, że opóźnienia czasowe wprowadzane za jej pomocą będą zawsze tym zaprogramowanym.

W przypadkach zadań, które wymagają dokładności w generowaniu zwłok czasowych dużo lepszym rozwiązaniem jest druga z funkcji opóźniających, a mianowicie vTaskDelay-Until().

Niedoskonałość funkcji vTaskDelay() wynika z tego, że od jednego jej wywołania do drugiego mogą upływać różne czasy, co nie jest uwzględniane. Problem przedstawiono na rysunku.


a)
Start
Zadanie
vTaskDelay()
Zadanie


b)
Start
Zadanie
vTaskDelay()
Zadanie


c)
Start
Zadanie
Przerwanie
Zadanie
vTaskDelay()
Zadanie

Funkcja vTaskDelayUntil() opóźnia wykonywanie kodu o czas obliczony na podstawie dwóch przekazanych argumentów.

Pierwszy jest liczbą taktów systemu od chwili uruchomienia planisty, natomiast drugi liczba taktów, na jaką wykonywanie zadania ma być wstrzymane.

W ten sposób, dodając do siebie obie wartości, funkcja API otrzymuje liczbę, po której osiągnięciu przez licznik taktów systemu operacyjnego, zostanie wznowione wykonywanie zablokowanego zadania.

Dzięki takiemu podejściu, osiągnięto niezależność w stosunku do czasu wykonywania kodu samego zadania, czyli nie są istotne tutaj wywłaszczenia zadania itd.

Wykorzystanie funkcji vTaskDelayUntil() przedstawiono na listingu.

```
void vTaskLED(void * pvParameters)
{
    portTickType xLastFlashTime;
    // Odczytanie stanu licznika systemowego
    xLastFlashTime = xTaskGetTickCount();
    // Nieskonczona petla zadania
    for(;;)
    {
    // Wprowadzenie opoznienia 500ms
    vTaskDelayUntil( &xLastFlashTime,
500/portTICK_RATE_MS );
    // Zmiana stanu wyprowadzenia PC6 (LD1) na przeciwny
    vhToggleLED();
    }
}
```

Liczbę taktów systemu operacyjnego otrzymuje się za pomocą wywołania funkcji xTaskGetTickCount(), natomiast liczbę taktów opóźnienia wyznaczono za pomocą całkowitego dzielenia przez stałą portTICK_RATE_MS.

Każde zadanie ma swój ustalony priorytet. Naturalną konsekwencją tego jest potrzeba istnienia w systemie mechanizmów umożliwiających odczytywanie priorytetu zadania oraz jego ustawianie.

Do realizacji powyższych zadań utworzono funkcje **uxTaskPriorityGet()** oraz **vTaskPrioritySet()**. Wykorzystanie w programie przedstawionych funkcji przedstawiono na listingu.

```
void vStartLEDTasks(unsigned portBASE_TYPE uxPriority)
{
    xTaskHandle xHandleTaskLED;
    unsigned portBASE_TYPE uxTaskLEDPriority;
    // Tworzenie zadania
    xTaskCreate( vTaskLED, ( signed portCHAR * ) „LED”,
    STACK_SIZE, NULL, uxPriority, &xHandleTaskLED);
    // Ustawienie priorytetu zadania
    vTaskPrioritySet(xHandleTaskLED, 3);
    // Odczytanie priorytetu zadania
    uxTaskLEDPriority =
uxTaskPriorityGet(xHandleTaskLED);
}
```

Zadaniem tego programu jest – po utworzeniu nowego zadania – zmiana priorytetu, a następnie jego odczyt.

Każde zadanie może być w jednym z kilku stanów. System operacyjny FreeRTOS umożliwia wstrzymywanie i wznawianie zadania, odpowiednio za pomocą funkcji: **vTaskSuspend()** i **vTaskResume()**.

Wstrzymywanie zadania jest bardzo proste i wymaga przekazania jedynie w argumencie do funkcji API uchwytu (nazwy) wstrzymywanego zadania. Jeśli np. chcemy wstrzymać zadanie LedTaskHandle, to wystarczy w kodzie umieścić linię:

```
vTaskSuspend(LedTaskHandle);
```

Ciekawą możliwością jest wstrzymywanie zadania przez samego siebie. Można tego dokonać wywołując funkcję wtrzymującą z argumentem NULL:

```
vTaskSuspend(NULL);.
```

Powyższy kod spowoduje wstrzymanie zadania, aż do momentu jego wznowienia za pomocą funkcji vTaskResume(), wywołanej w jakimś innym zadaniu uruchomionym w systemie.

Komunikacja między uruchomionymi zadaniami, kolejki i semafory, synchronizacja procesów

Podstawowym narzędziem, służącym do komunikacji pomiędzy zadaniami w systemie FreeRTOS, są **kolejki (Queues)**.

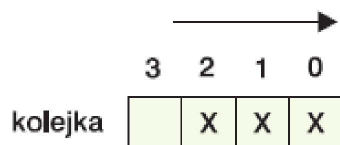
Mechanizm kolejek pozwana na przesyłanie informacji również pomiędzy zadaniami, a przerwaniami.

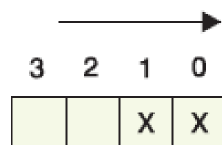
Konstrukcja kolejek opiera się o kolejkę FIFO, co oznacza tyle, że pierwsza zapisana do niej dana będzie pierwszą odczytaną (Firt In/First Out).

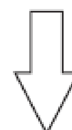
Zasadę działania wymiany informacji między zadaniami za pomocą kolejki ilustruje rysunek, na którym przedstawiono proces komunikacji za pomocą kolejki czteroelementowej.

Kierunek przesuwania
elementów po odczycie

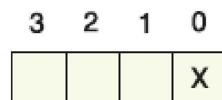
b) Odczyt kolejki

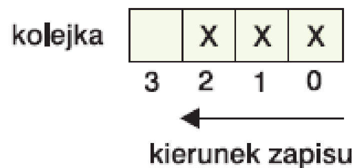






a) Zapis kolejki





ZADANIE A zapisuje bufor FIFO począwszy od pozycji zerowej (rys. a).

Po zapisaniu trzech elementów ZADANIE B rozpoczyna odczytywanie zawartość kolejki, począwszy również od elementu zerowego. Ważne jest, że po każdym odczycie zawartość całej kolejki jest przesuwana o jedną pozycję w kierunku pozycji zerowej, co przedstawiono na rys. b.

W ten sposób zawsze pierwszy z wpisanych do kolejki elementów będzie pierwszym odczytanym. Zarówno typ danych przechowywanych w kolejce, jak i jej rozmiar, są definiowane podczas tworzenia kolejki.

Jeśli istnieje potrzeba przekazywania większych ilości danych, to można wykorzystać do tego celu wskaźniki, które będą zapisywane do kolejki, jednak w tym przypadku należy zadbać o to, aby zawsze było wiadomo, kto zapisał wskaźniki do kolejki.

Tworzenie kolejki w systemie FreeRTOS może się odbywać przez wywołanie funkcji **xQueueCreate()**. W argumentach należy podać liczbę elementów kolejki i rozmiar każdego elementu w bajtach.

Jeśli zatem przykładowo kolejka ma mieć rozmiar 5 elementów, a każdy ma mieć 2 bajty, to należy w programie umieścić kod:

```
xQueue5x2 = xQueueCreate( 10,sizeof( unsigned portLONG
2 );
```

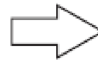
Oczywiście nie można zapominać o utworzeniu wcześniej zmiennej xQueue5x2 typu xQueueHandle.

Semafory są przede wszystkim stosowane do synchronizacji zadań (lub zadań i przerwań) oraz wykluczenia nieuprawnionych dostępów.

Zasadę działania semafora binarnego przedstawiono na rysunku.

a) Semafor nieaktywny – zadanie zablokowane

b) Semafor aktywny – zadanie gotowe



Załóżmy, że nieaktywny semafor blokuje wykonywanie zadania, natomiast jego aktywacja wywołuje nadanie zadaniu statusu gotowego do wykonywania.

Sytuacja, w której semafor jest nieaktywny, a zadanie jest zablokowane, przedstawiono na rys. a.

W tym przypadku wartość semafora ustawiona jest na 0 (wartość umowna, wszystko zależy od przyjętej logiki).

Jeśli teraz jakiś wyjątek, przykładowo przerwanie zewnętrzne spowoduje aktywowanie semafora, czyli ustawienie jego wartości na 1, to jest to znak dla systemu operacyjnego, aby zmienić stan zadania z zablokowanego na gotowe do wykonywania (rys. b)

Konfiguracja systemu FreeRTOS. Plik konfiguracyjny FreeRTOSConfig.h

Podstawowe ustawienia systemu, takie jak częstotliwość taktowania CPU, są ustalane w pliku konfiguracyjnym FreeRTOSConfig.h. Jego fragment zamieszczono na listingu.

```
#define configUSE_PREEMPTION 1
#define configUSE_IDLE_HOOK 0
#define configUSE_TICK_HOOK 0
#define configCPU_CLOCK_HZ ( ( unsigned portLONG )
72000000 )
#define configTICK_RATE_HZ ( ( portTickType ) 1000 )
#define configMAX_PRIORITIES ( ( unsigned portBASE_TYPE )
5 )
#define configMINIMAL_STACK_SIZE ( ( unsigned portSHORT )
128 )
#define configTOTAL_HEAP_SIZE ( ( size_t ) ( 17 *
1024 ) )
#define configMAX_TASK_NAME_LEN ( 16 )
#define configUSE_TRACE_FACILITY 0
```

```
#define configUSE_16_BIT_TICKS 0
#define configIDLE_SHOULD_YIELD 1
/* Co-routine definitions. */
#define configUSE_CO_ROUTINES 0
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )
/*Set the following definitions to 1 to include
the API function, or zero to exclude the API function*/
#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
```

Pierwsza sekcja konfiguruje parametry pracy, natomiast druga włącza lub wyłącza poszczególne funkcje API odpowiedzialne za sterowanie zadaniami.

Jak nietrudno się domyślić, wartość 1 włącza możliwość korzystania z funkcji, a 0 wyłącza.

Warto o tym pamiętać, bo czasem może się zdarzyć, że nie wiadomo z jakiego powodu np. funkcje opóźniające nie chcą działać, a przyczyna jest bardzo prosta – nie włączono obsługi tych funkcji w pliku FreeRTOSConfig.h.

Z przedstawionego na listingu programu wynika, że w tym przypadku włączono wszystko, poza funkcją vTaskCleanUpResources().

Sekcja ustawiania parametrów systemu umożliwia dostosowanie właściwości OS do wymagań aplikacji. Ustawiane parametry pracy systemu to m.in.:
– **configUSE_PREEMPTION** decyduje, czy ma być używane wywłaszczanie (preemption).
– **configUSE_IDLE_HOOK** – ustawienie wartości 1 spowoduje włączenie obsługi funkcji vApplicationIdleHook(), która będzie wywoływana zawsze podczas procesu bezczynności systemu. Jest to stosunkowo prosty mechanizm pozwalający na wprowadzanie mikrokontrolera w tryb obniżonego poboru mocy.
– **configCPU_CLOCK_HZ** informuje jądro systemu o częstotliwości pracy mikrokontrolera; wartość domyślna to 72 MHz.
– **configTICK_RATE_HZ** określa zegar OS, domyślnie 1kHz.
– **configMAX_PRIORITIES** definiuje maksymalną liczba priorytetów.
– **configMINIMAL_STACK_SIZE** określa minimalny rozmiar stosu.
– **configMAX_TASK_NAME_LEN** definiuje maksymalną długość nazwy zadania.

Aplikacja wykorzystująca system FreeRTOS do obsługi wielu zadań

Po tym opisie teoretycznym na temat działania wbudowanych systemów operacyjnych, a w szczególności systemu FreeRTOS, przejdziemy do prostego przykładu praktycznego, ilustrującego sposób budowania aplikacji działającej w oparciu o ten system operacyjny.

Wykonany będzie prosty program mający za zadanie sterowanie trzema zadaniami, a ściślej częstotliwością migotania trzech diod: LD1, LD2, LD3.

```
#define mainFLASH_TASK_PRIORITY ( tskIDLE_PRIORITY + 1 )
static void prvSetupHardware( void );
int main( void )
{
    // Konfiguracja sprzetu
    prvSetupHardware();
    // Uruchomienie zadan
    vStartLEDTasks( mainFLASH_TASK_PRIORITY );
    // Uruchomienie planisty
    vTaskStartScheduler();
    return 0;
}
```

Na listingu przedstawiono główną funkcję programu (main()), której zadaniem jest konfiguracja systemu operacyjnego do pracy, uruchomienie zadań i przekazanie sterowania do jądra systemu FreeRTOS.

Pierwszą czynnością wykonywaną przez mikrokontroler jest ustawienie wszystkich niezbędnych do poprawnej pracy, parametrów.

Wykonuje to funkcja prvSetupHardware(). Jej zawartość niczym nie rożni się od stosowanej we wszystkich przykładach funkcji konfiguracyjnych.

Po skonfigurowaniu sprzętu mikrokontroler przechodzi do uruchomienia zadań, a następnie aktywowany jest planista, czyli algorytm szeregowania – funkcja vTaskStart-Scheduler().

Jeśli tylko aplikacja napisana jest poprawnie, to mikrokontroler nigdy nie wykona instrukcji return, następującej po włączeniu planisty.

Od tego momentu, system operacyjny zajmuje się wykonywaniem uruchomionych zadań.

Na listingu przedstawiono kod zadań sterujących pracą diod LD1, LD2 i LD3,

```
void vTaskLED1(void*);
void vTaskLED2(void*);
void vTaskLED3(void*);
void vStartLEDTasks(unsigned portBASE_TYPE uxPriority)
{
    xTaskHandle xHandleTaskLED1, xHandleTaskLED2,
xHandleTaskLED3;
    // Tworzenie zadania migania LD1 z f = 1Hz
    xTaskCreate(vTaskLED1, ( signed portCHAR * ) „LED1”,
    ledSTACK_SIZE, NULL, uxPriority,
    &xHandleTaskLED1);
    // Tworzenie zadania migania LD2 z f = 0.5Hz
    xTaskCreate(vTaskLED2, ( signed portCHAR * ) „LED2”,
    ledSTACK_SIZE, NULL, uxPriority,
    &xHandleTaskLED2);
    // Tworzenie zadania migania LD3 z f = 0.25Hz
    xTaskCreate(vTaskLED3, ( signed portCHAR * ) „LED3”,
    ledSTACK_SIZE, NULL, uxPriority,
    &xHandleTaskLED3);
}
```

```
void vTaskLED1(void * pvParameters)
{
    portTickType xLastFlashTime;
    // Odczytanie stanu licznika systemowego
    xLastFlashTime = xTaskGetTickCount();
    // Nieskonczona petla zadania
    for(;;)
    {
    // Wprowadzenie opoznienia 1000ms
    vTaskDelayUntil( &xLastFlashTime,
500/portTICK_RATE_MS );
    // Zmiana stanu wyprowadzenia PC6 (LD1) na przeciwny
    vhToggleLED1();
    }
}
void vTaskLED2(void * pvParameters)
{
    portTickType xLastFlashTime;
    // Odczytanie stanu licznika systemowego
    xLastFlashTime = xTaskGetTickCount();
    // Nieskonczona petla zadania
    for(;;)
```

```
    {
    // Wprowadzenie opoznienia 2000ms
    vTaskDelayUntil( &xLastFlashTime,
1000/portTICK_RATE_MS );
    // Zmiana stanu wyprowadzenia PC7 (LD2) na przeciwny
    vhToggleLED2();
    }
}
void vTaskLED3(void * pvParameters)
{
    portTickType xLastFlashTime;
    // Odczytanie stanu licznika systemowego
    xLastFlashTime = xTaskGetTickCount();
    // Nieskonczona petla zadania
    for(;;)
    {
    // Wprowadzenie opoznienia 500ms
    vTaskDelayUntil( &xLastFlashTime,
2000/portTICK_RATE_MS );
    // Zmiana stanu wyprowadzenia PC8 (LD3) na przeciwny
    vhToggleLED3();
    }
```

```
}
```

Natomiast na kolejnym listingu funkcje obsługi wyprowadzeń dla diod LED. Diody będą migać z częstotliwością: LED1 – 1 Hz, LED2 – 0,5 Hz, LED3 – 0,25 Hz.

```
void vhToggleLED1(void)
{
    GPIO_WriteBit(GPIOC, GPIO_Pin_6, (BitAction)
    ((1-GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_6))));
}
void vhToggleLED2(void)
{
    GPIO_WriteBit(GPIOC, GPIO_Pin_7, (BitAction)
    ((1-GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_7))));
}
void vhToggleLED3(void)
{
    GPIO_WriteBit(GPIOC, GPIO_Pin_8, (BitAction)
    ((1-GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_8))));
}
```