FreeRTOS w PSOC5LP

Do czego służy i jak zaimplementować FreeRTOS w układach Cypress PSoC 5LP

v. 1.2 Bogdan Pankiewicz, marzec 2016, 2017

Materiały źródłowe

[1] Elektronika Praktyczna, 5/2009, Krzysztof Paprocki „Mikrokontrolery STM32 Praca pod kontrolą FreeRTOS”.

[2] Elektronika Praktyczna, 7/2009, Krzysztof Paprocki „STM32 FreeRTOS dla dociekliwych”.

[3] www.freertos.org

[4] www.cypress.com, przykład aplikacyjny „User Guide for FreeRTOS on the CY8CKIT-050 (PSoC 5LP Developers Kit)”.

[5] K. Paprocki, „Mikrokontrolery STM32 w praktyce”, BTC, Legionowo 2011.

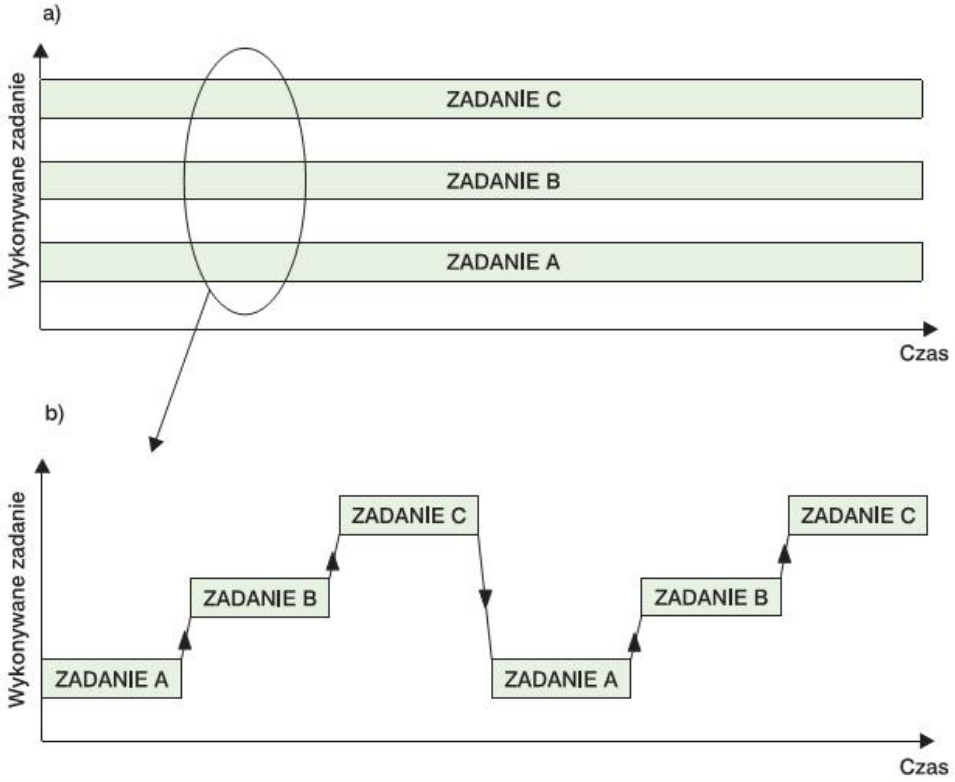
RTOS [1] - [3]

- Ang. Real Time Operating System - system operacyjny czasu rzeczywistego.
- System taki ma na celu spełnienie wymagań czasowych narzuconych na wykonywane zadania.
- Zastosowania: wojskowe (np. sterowanie rakietami), sterowanie procesami przemysłowymi, ABS, ...
- Podział:
  - Twarde: znany jest najdłuższy możliwy czas realizacji zadania
  - Miękkie: działają z priorytetem jak najszybszego wykonania zadań ale czas wykonania nie jest gwarantowany

Wybrane systemy RTOS do implementacji w mikrokontrolerach

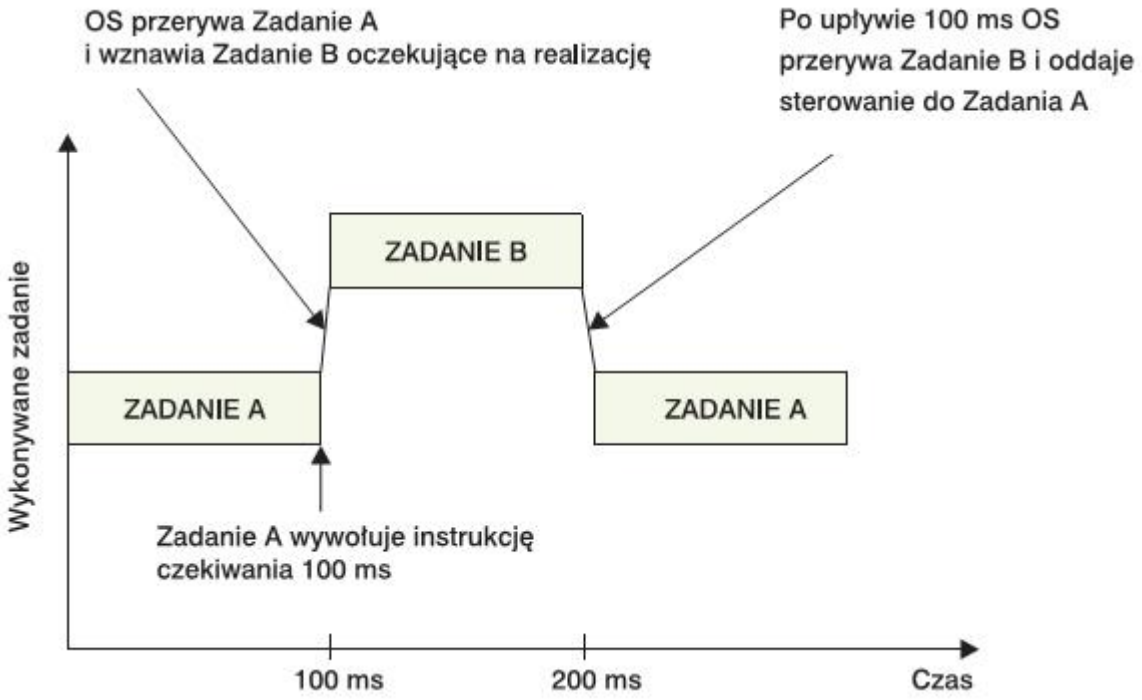
- FreeRTOS: www.freertos.org
- Micrium RTOS: www.micrium.com
- ISIX RTOS: bryndza.boff.pl
- VxWORKS
- QNX
- RTX: https://www.arm.com/products/tools/software-tools/mdk-arm/middleware-libraries/rtx-real-time-operating-system.php
- I wiele innych: https://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems

RTOS – zasada działania [1-2,5]



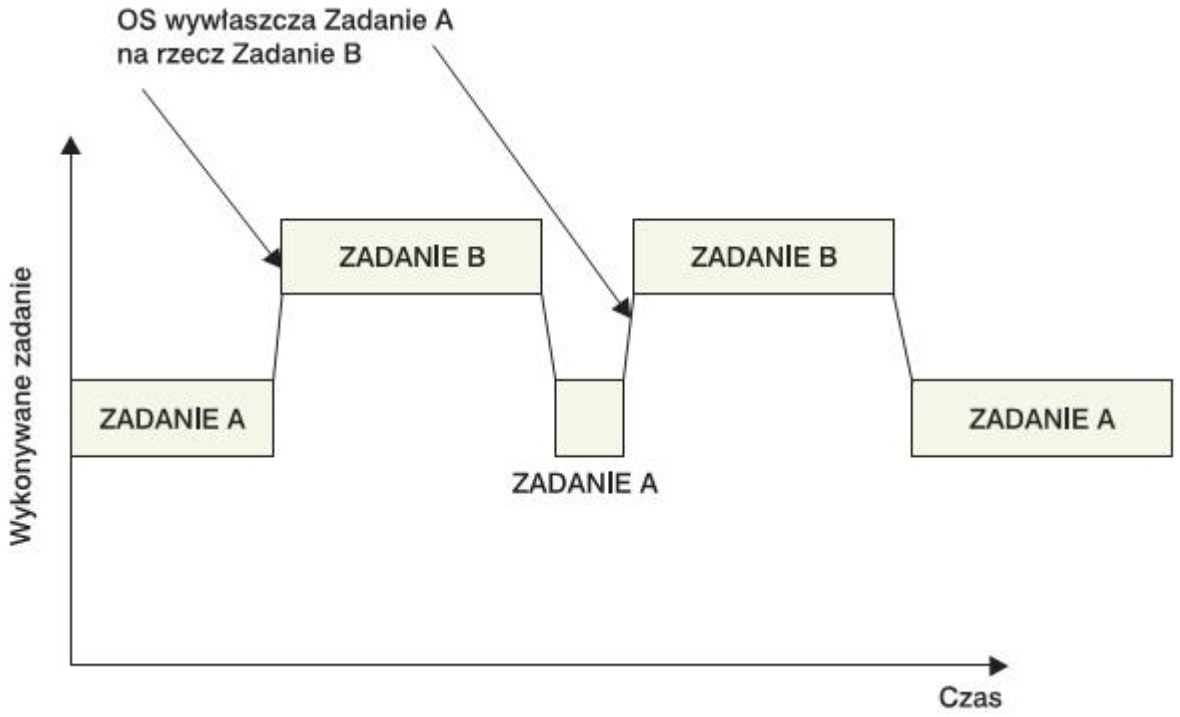
Przebieg pracy wielozadaniowego systemu operacyjnego [1]

RTOS – zasada działania, c.d. [1-2 ,5]



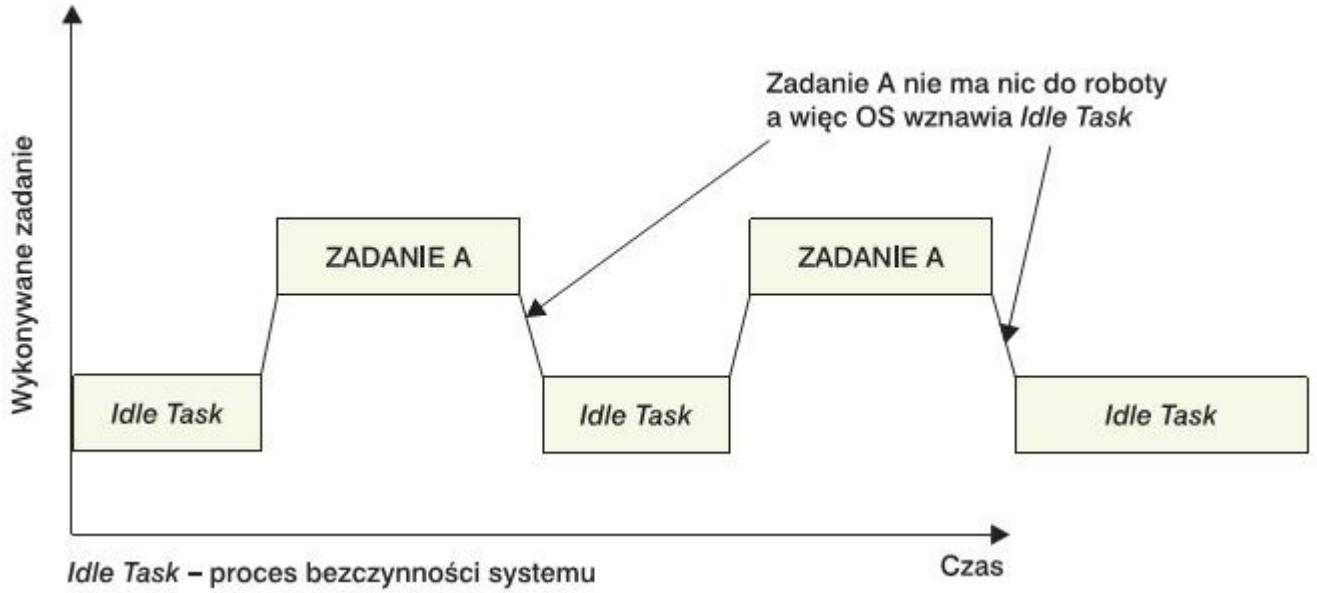
Przełączanie pomiędzy zadaniami co 100ms.

RTOS – zasada działania, c.d. [1-2 ,5]



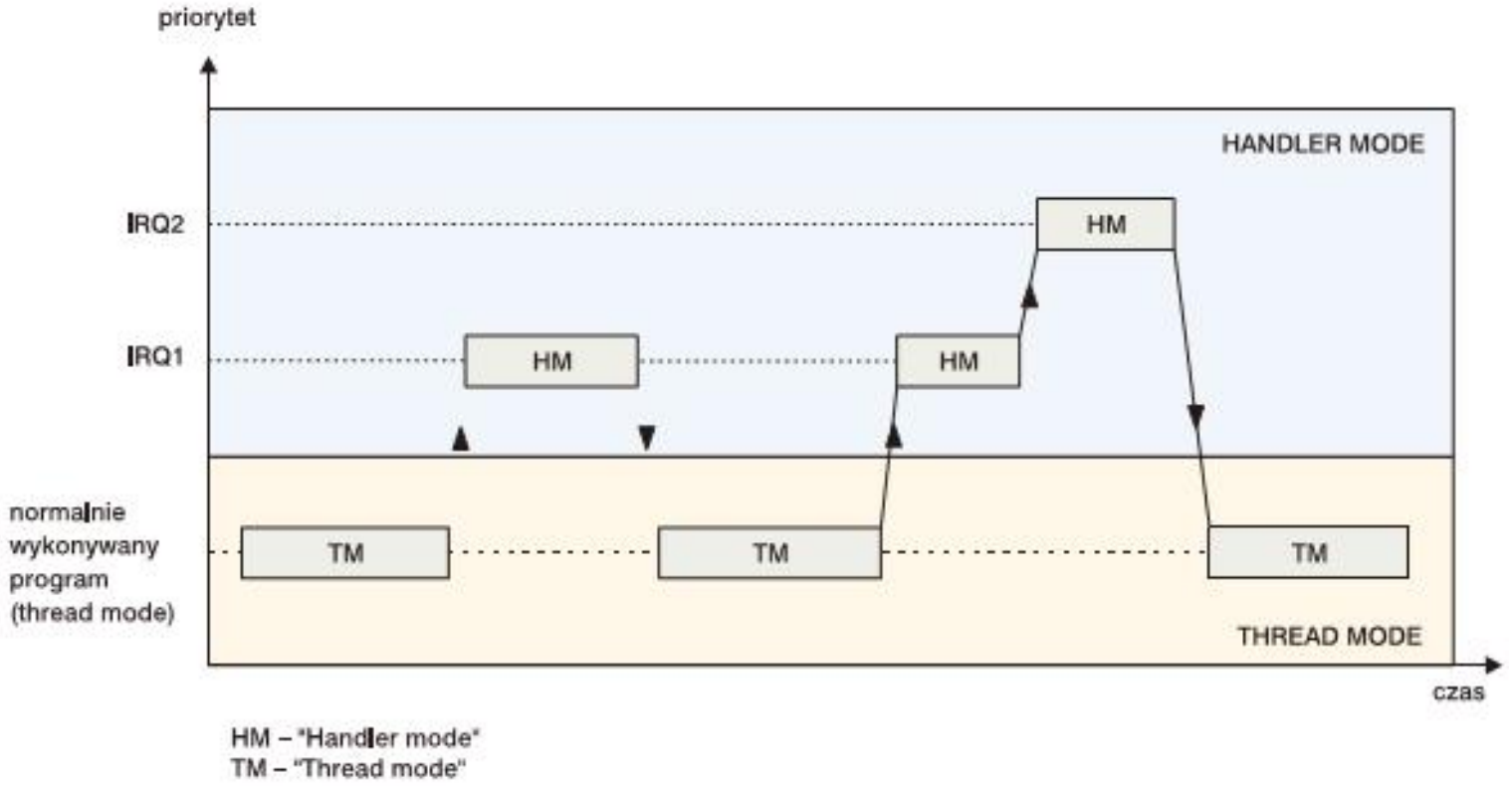
System operacyjny z wywłaszczaniem (ang. preemptive). Zadanie B ma wyższy priorytet niż zadanie A.

RTOS – zasada działania, c.d. [1-2 ,5]



Zadanie bezczynności (ang. idle task) jest uruchamiane automatycznie zawsze przez system operacyjny kiedy nie ma innych zadań do wykonania.

Cechy rdzenia Cortex-M3 wspomagające implementację RTOS [5]



Cechy rdzenia Cortex-M3 wspomagające implementację RTOS [5] c.d.

CONTROL[0]:

0 – tryb uprzywilejowany

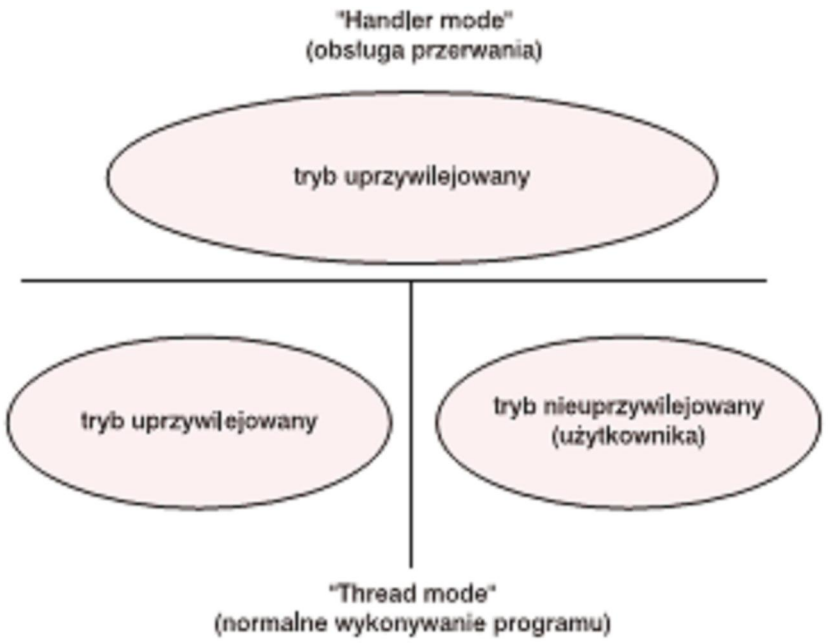
1 – tryb użytkownika

CONTROL[1]:

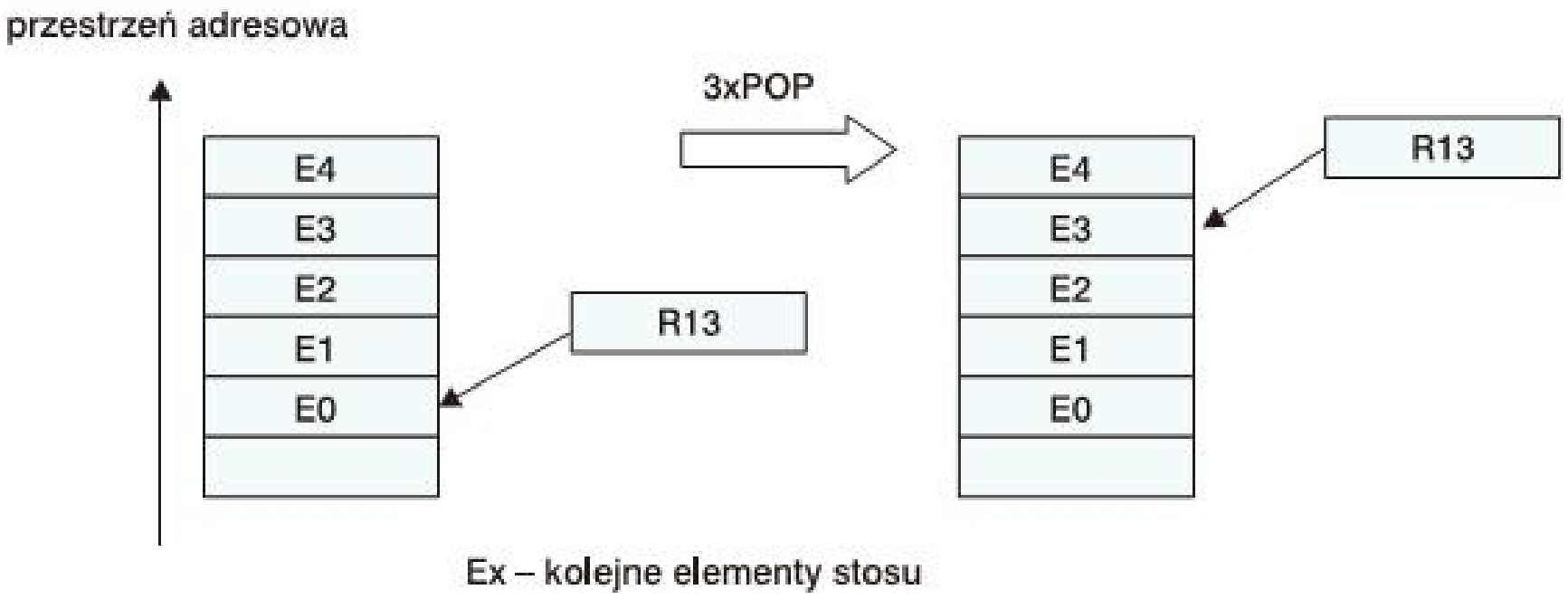
0 – wykorzystywany MSP

1 – wykorzystywany PSP*

* – Oczywiście tylko w czasie normalnego wykonywania programu, w obsłudze przerwania jest używany MSP

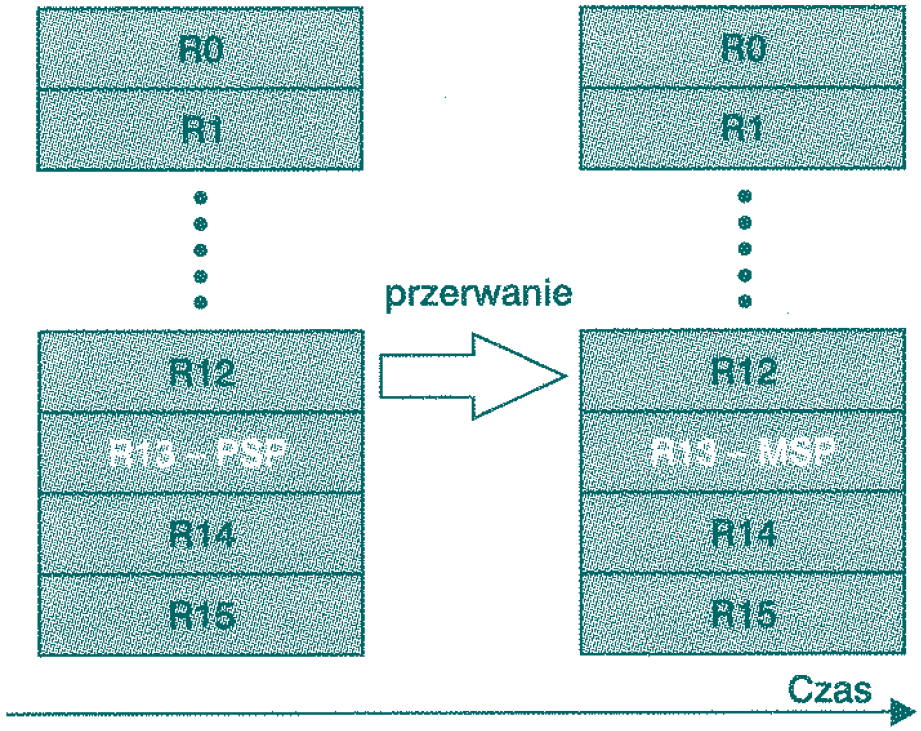


Cechy rdzenia Cortex-M3 wspomagające implementację RTOS [5] c.d.



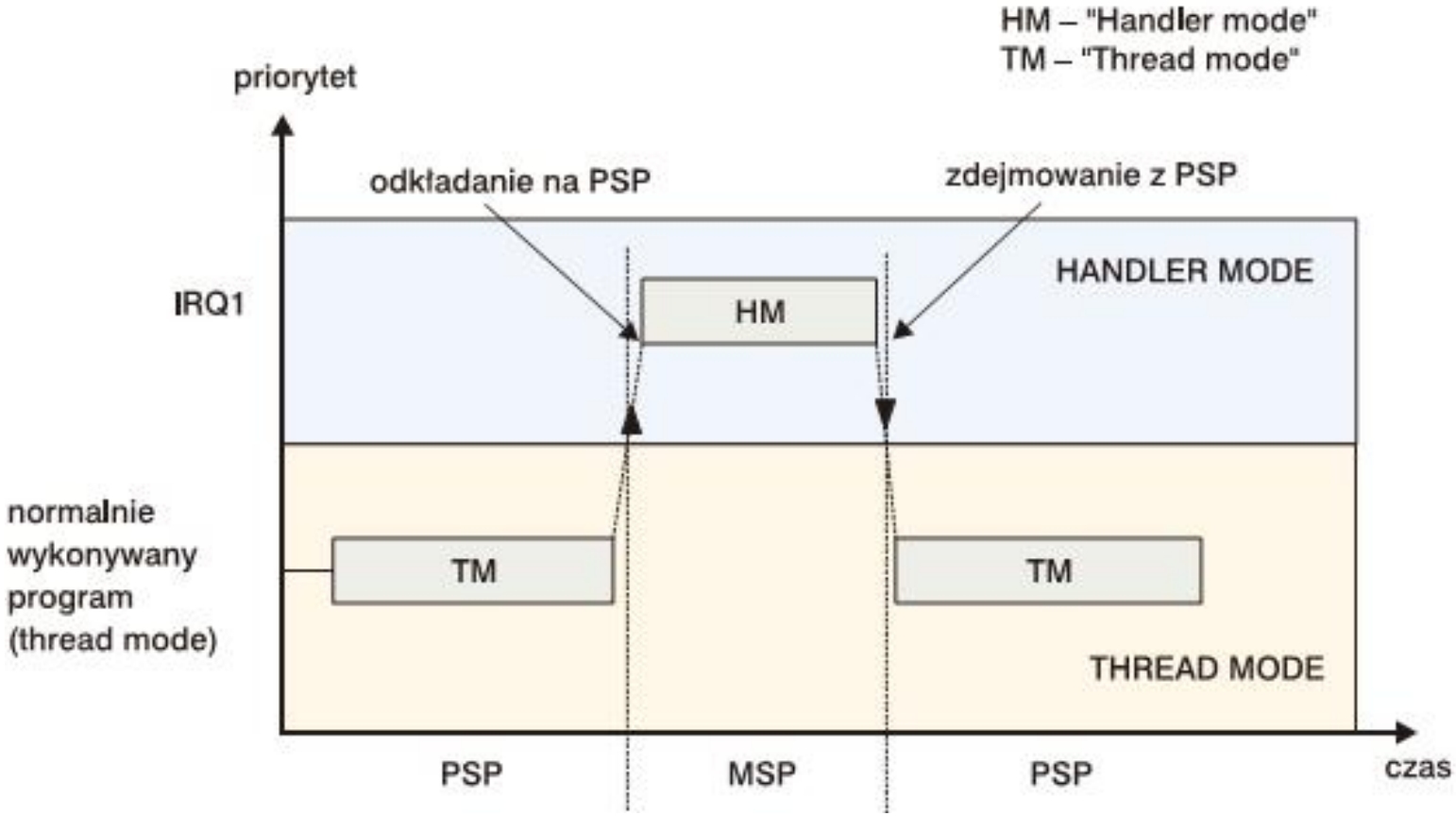
Ex – kolejne elementy stosu

Cechy rdzenia Cortex-M3 wspomagające implementację RTOS [5] c.d.

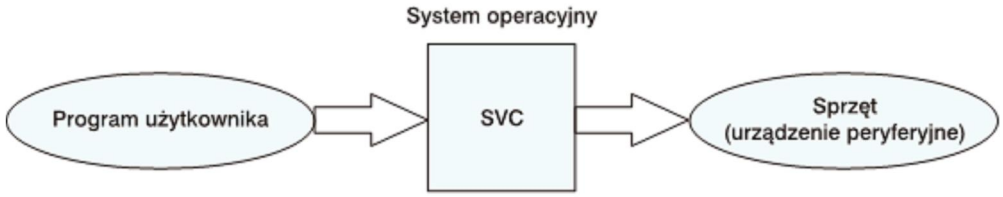


Bankowanie wskaźników stosów (MSP i PSP) w mikrokontrolerach z rdzeniem Cortex-M3

Cechy rdzenia Cortex-M3 wspomagające implementację RTOS [5] c.d.

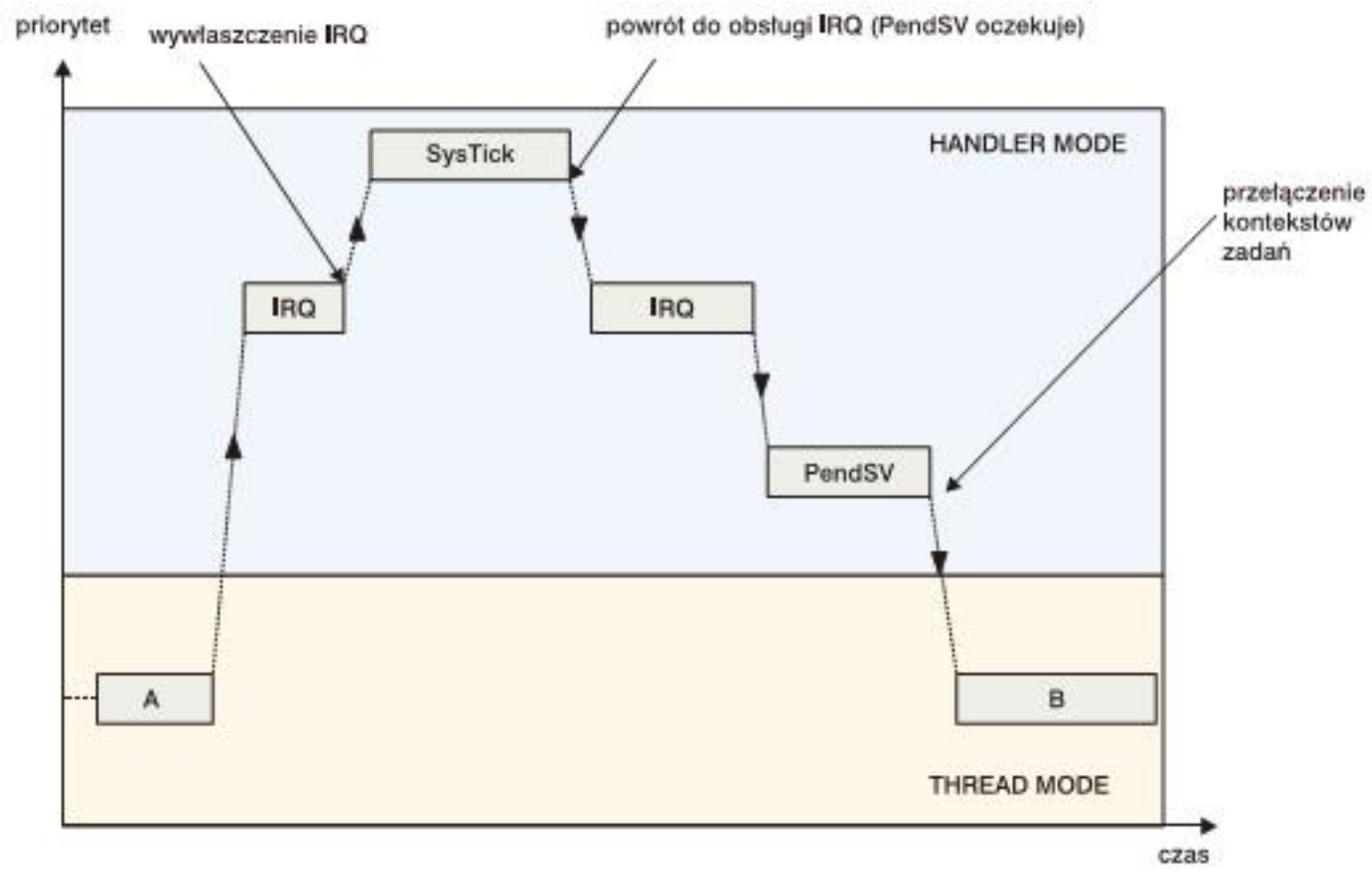


Cechy rdzenia Cortex-M3 wspomagające implementację RTOS [5] c.d.



Exception Number	Exception Type	Priority	Exception Table Address Offset	Function
			0x00	Starting value of R13 / MSP
1	Reset	-3 (highest)	0x04	Reset
2	NMI	-2	0x08	Non maskable interrupt
3	Hard fault	-1	0x0C	All classes of fault, when the corresponding fault handler cannot be activated because it is currently disabled or masked
4	MemManage	Programmable	0x10	Memory management fault, for example, instruction fetch from a nonexecutable region
5	Bus fault	Programmable	0x14	Error response received from the bus system; caused by an instruction prefetch abort or data access error
6	Usage fault	Programmable	0x18	Typically caused by invalid instructions or trying to switch to ARM mode
7 – 10	-	-	0x1C – 0x28	Reserved
11	SVC	Programmable	0x2C	System service call via SVC instruction
12	Debug monitor	Programmable	0x30	Debug monitor
13	-	-	0x34	Reserved
14	PendSV	Programmable	0x38	Deferred request for system service
15	SYSTICK	Programmable	0x3C	System tick timer
16 – 47	IRQ	Programmable	0x40 – 0x3FC	Peripheral interrupt request #0 - #31

Cechy rdzenia Cortex-M3 wspomagające implementację RTOS [5] c.d.



Najważniejsze cechy FreeRTOS: [3]

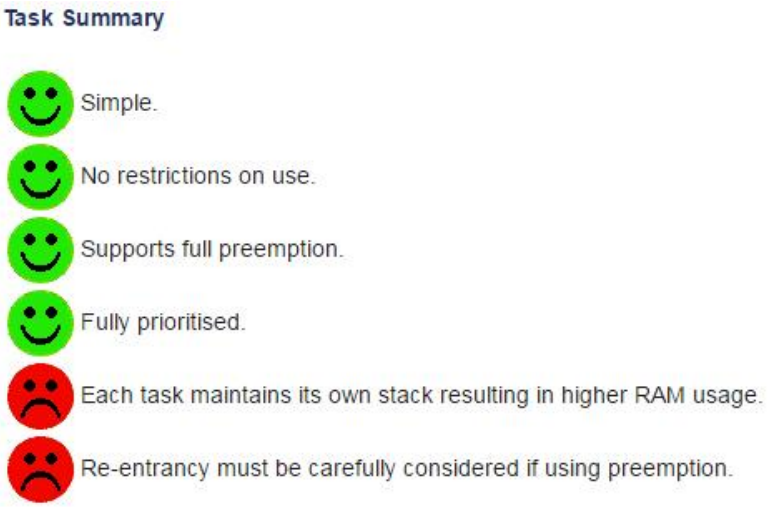
- Algorytm szeregowania (ang. scheduler) typu wywłaszczeniowego (ang. preemptive) jak też współdziałający (ang. cooperative) i hybrydowy (w zależności od konfiguracji).
- Obiekty RTOS: zadania (ang. task), semafory, muteksy, timery programowe, kolejki i zdarzenia grupowe - mogą być tworzone w statycznie lub dynamicznie alokowanej pamięci RAM.
- Oficjalne wsparcie dla ponad 30 typów rdzeni mikrokontrolerów, m.in.: ARM-CM3, ARM7, AVR ...
- Mały, prosty i łatwy w użyciu, zajmuje tylko 4k-9kB pamięci.
- Darmowy, Open Source License.

Zadania (ang. task) [3]:

• aplikacja używająca RTOSu może być podzielona na zestaw niezależnych zadań,

• każde zadanie jest wykonywane we własnym niezależnym kontekście,

• w danej chwili czasu tylko jedno zadanie jest wykonywane,

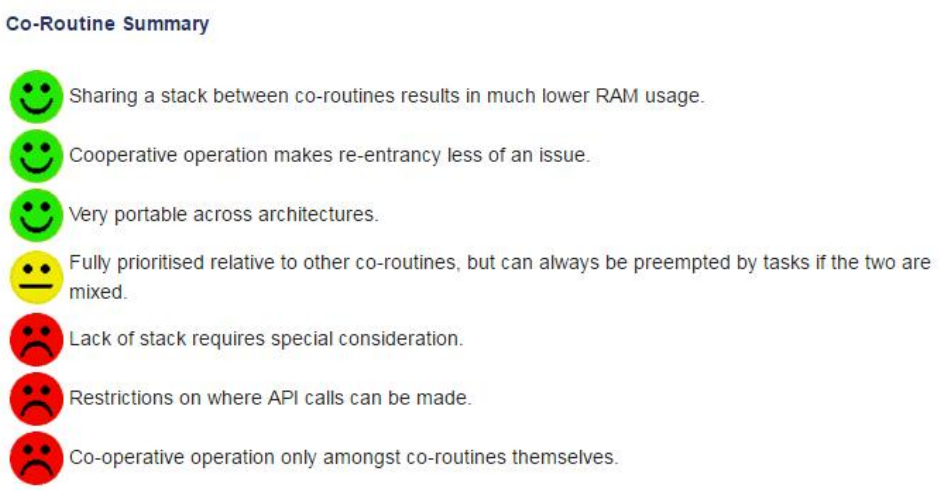


• algorytm szeregowania zadań (ang. scheduler) decyduje, które zadanie jest wykonywane w danej chwili czasu,

• zadania nie mają informacji o tym jak działa scheduler, nie wiedzą nawet czy były przełączone w międzyczasie na inne zadanie bo przełączanie zadań przywraca kontekst (kontekst – zawartość rejestrów procesora).

Współprogramy (ang. co-routine) [3]

- wszystkie współprogramy dzielą wspólny stos,
- były używane na bardzo małych mikrokontrolerach ze względu na niskie zapotrzebowanie na RAM,



- obecnie raczej używa się zadań niż współprogramów bo łatwiej jest zapewnić przełączanie z wywłaszczaniem,

Zadania w systemie FreeRTOS [3]:

Możliwe stany zadań:

• **Running** – zadanie jest wykonywane, w danej chwili czasu wykorzystuje procesor,

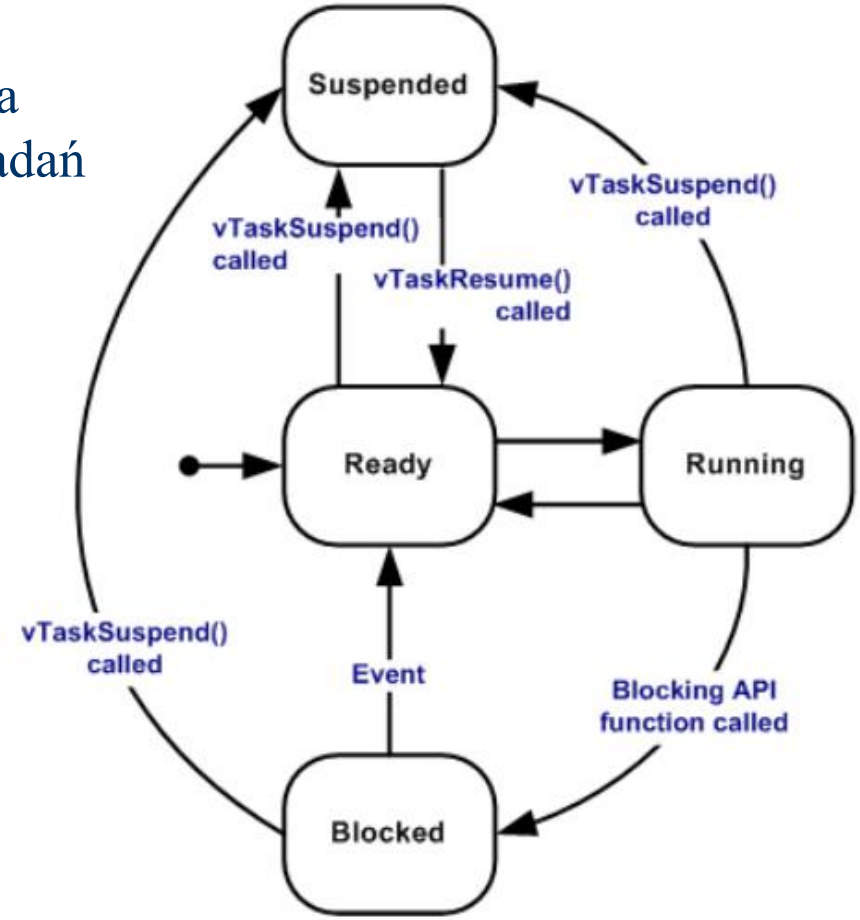
• **Ready** – zadanie, które może być wykonywane (nie jest ani w stanie **Blocked** ani w stanie **Suspended**) ale nie jest w danej chwili wykonywane bo inne zadanie o tym samym lub wyższym priorytecie jest w stanie **Running**,

• **Blocked** – zadanie które aktualnie oczekuje na zdarzenie zewnętrzne lub czasowe, na przykład jeśli zadanie wywoła ***`vTaskDelay()`*** będzie w stanie **Blocked** do czasu upłynięcia czasu blokady (zdarzenie czasowe), zadanie może także oczekiwać na zdarzenie w kolejce lub semaforze, zadanie w stanie **Blocked** zawsze ma czas upłynięcia blokady (timeout) po którym jest automatycznie odblokowywane, zadania w stanie **Blocked** nie są możliwe do umieszczenia w schedulerze,

• **Suspended** - nie są możliwe do umieszczenia w schedulerze, zadania są umieszczane lub wyjmowane ze stanu **Suspend** wyłącznie poprzez wywołanie poleceń odpowiednio ***`vTaskSuspend()`*** oraz ***`vTaskResume()`***, nie ma czasu upłynięcia stanu **Suspend**.

Zadania w systemie FreeRTOS, c.d. [3]:

Możliwe przejścia pomiędzy stanami zadań



Zadania w systemie FreeRTOS, priorytety [3]

Priorytety zadań:

• każdemu zadaniu przypisany jest priorytet liczony od 0 (najmniejszy priorytet) do wartości (*configMAX_PRIORITIES-1*), wartość *configMAX_PRIORITIES* jest zdeklarowana w pliku *FreeRTOSConfig.h*,

• algorytm kolejkowania zadań (scheduler) zapewnia to, że zadania w stanie **Ready** oraz **Running** zawsze dostaną czas procesora zamiast zadań w stanie **Ready** o niższych priorytetach, inaczej mówiąc zadanie przełączone do stanu **Running** jest zawsze zadaniem o najwyższym priorytecie, które można uruchomić,

• dowolna liczba zadań może mieć ten sam priorytet, jeśli nie zdefiniowano zmiennej *config_USE_TIME_SLICING* lub zdefiniowano ją na wartość równą 1 wówczas zadania o takim samy priorytecie będące w stanie **Ready** będą miały przydzielany czas procesora wg algorytmu karuzelowego (ang. round robin).

Zadania w systemie FreeRTOS, implementacja zadania [3]

Struktura tworząca zadanie przedstawiona jest poniżej. Zdefiniowany jest typ *`TaskFunction_t`* który jest funkcją zwracająca *`void`* a argumentem jest wskaźnik do *`void`*. Parametr służy do przekazywania do zadania informacji każdego typu, przykłady są zawarte w aplikacjach demonstracyjnych.

Zadanie nie powinno się nigdy zakończyć stąd jest implementowane jako nieskończona pętla. Zadania są tworzone przez wywołanie funkcji *`xTaskCreate()`* a kasowane przez *`vTaskDelete()`*

```
void vATaskFunction( void *pvParameters )
{
    for( ;; )
    {
        -- Task application code here. --
    }

    /* Tasks must not attempt to return from their implementing
    function or otherwise exit.  In newer FreeRTOS port
    attempting to do so will result in an configASSERT() being
    called if it is defined.  If it is necessary for a task to
    exit then have the task call vTaskDelete( NULL ) to ensure
    its exit is clean. */
    vTaskDelete( NULL );
}
```

Zadania w systemie FreeRTOS, implementacja zadania c.d. [3]

Zadanie można alternatywnie utworzyć z wykorzystaniem makr *portTASK_FUNCTION* oraz *portTASK_FUNCTION_PROTO*. Zadanie z poprzedniego slajdu można więc alternatywnie utworzyć poprzez:

Prototyp zadania:

```
void vATaskFunction( void *pvParameters );  zamienia się na
portTASK_FUNCTION_PROTO( vATaskFunction, pvParameters );
```

Zadanie:

```
portTASK_FUNCTION( vATaskFunction, pvParameters )
    {
        for( ;; )
        {
            -- Task application code here. --
        }
    }
```

Zadanie bezczynności – Idle [3]

Zadanie bezczynności – Idle jest tworzone automatycznie w momencie uruchomienia schedulera. Ma ono najniższy priorytet. Zadanie to jest odpowiedzialne za zwalnianie pamięci alokowanej przez RTOS do wykonania zadań które zostały skasowane poleceniem `vTaskDelete()`.

Zaczep do zadania bezczynności jest funkcją, która jest wywoływana w ciągu każdego cyklu zadania Idle. Jeśli użytkownik chce wykonywać jakąś część swojej aplikacji z najniższym priorytetem może to zrobić na 2 sposoby:

• stworzyć funkcjonalność z wykorzystaniem zaczepu bezczynności,

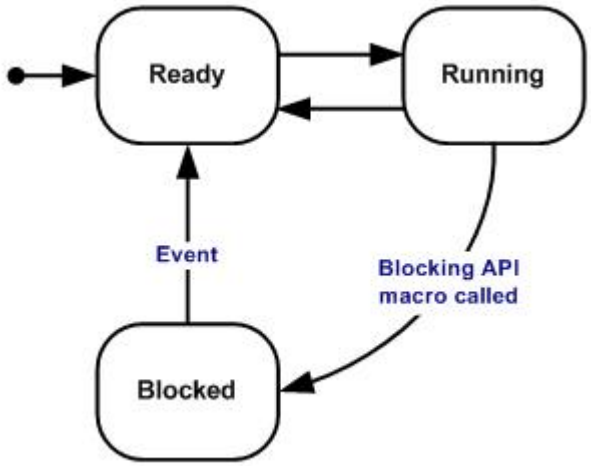
• stworzyć zadanie o najniższym priorytecie (równym priorytetowi bezczynności), ta wersja jest bardziej elastyczna ale używa więcej pamięci RAM .

Często używa się zaczepu bezczynności do wprowadzenia mikrokontrolera do stanu oszczędzającego energię.

Współprogramy (co-routines), możliwe stany [3]

Współprogramy są przewidziane do stosowania wyłącznie w niewielkich mikrokontrolerach i zazwyczaj nie są używanie w architekturach 32-bitowych. Możliwe stany współprogramu:

- **Running** – współprogram jest wykonywany,
- **Ready** - takie współprogramy, które są gotowe do działania (nie są zablokowane) ale nie są obecnie obsługiwane z bo:
  - inny współprogram o takim samym priorytecie jest wykonywany,
  - zadanie jest wykonywane, taki przypadek jest możliwy jedynie jeśli w aplikacji występują zarówno zadania jak i współprogramy,



Współprogramy (co-routines), możliwe stany, c.d. [3]

• **Blocked** - współprogram jest w stanie **Blocked** jeśli czeka na wystąpienie zdarzenia czasowego lub zewnętrznego, przykład – współprogram wywołuje funkcję ***crDELAY()***, która go blokuje aż do czasu upłynięcia czasu blokady, zablokowane współprogramy nie są możliwe do umieszczenia w schedulerze.

Współprogramy (co-routines), priorytety [3]

Współprogramy mogą mieć przypisany priorytet z zakresu od 0 do *`( configMAX_CO_ROUTINE_PRIORITIES - 1 )`*, wartość *`configMAX_CO_ROUTINE_PRIORITIES`* deklarowana jest w pliku *`FreeRTOSConfig.h`*. Niskie numery oznaczają niski priorytet.

Priorytety współprogramów są ważne tylko w stosunku do innych współprogramów, zadanie ma zawsze wyższy priorytet.

Implementacja współprogramu [3]

Współprogram powinien mieć następującą strukturę:

```
void vACoRoutineFunction( CoRoutineHandle_t xHandle, UBaseType_t uxIndex )
    {
        crSTART( xHandle );
        for( ;; )
        {
            -- Co-routine application code here. --
        }
        crEND();
    }
```

Współprogram jest zdefiniowany jako funkcja zwracająca `void` a biorąca jako argument typ `CoRoutineHandle_t`. Współprogramy są tworzone poprzez wywołanie funkcji `xCoRoutineCreate()`.

Implementacja współprogramu, c.d. [3]

Ważne uwagi:

• współprogram musi zaczynać się wywołaniem `crSTART()`,

• współprogram musi kończyć się wywołaniem `crEND()`,

• współprogram nie może nigdy się zakończyć więc jest zazwyczaj implementowany jako nieskończona pętla,

• wiele wpsółprogramów może być stworzonych w jednej funkcji, do rozróżniania ich służy parametr `uxIndex`.

Umieszczenie współprogramu w kolejce wykonania [3]

Aby wykonać współprogram należy wywołać `vCoRoutineSchedule()`, Najlepszym miejscem do wywołania ww. funkcji jest zaczep do zadania bezczynności.

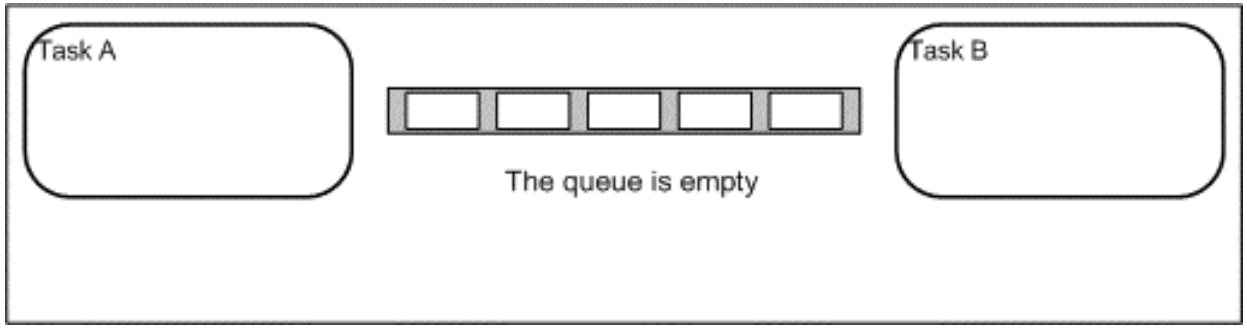
Synchronizacja i komunikacja pomiędzy zadaniami [3]

- Semafor binarny,
- Semafor liczący,
- Kolejka,
- Mutex,
- Rekursywny mutex,
- Komunikaty międzyzadaniowe (prostsza alternatywa niż semafory), FreeRTOS od v8.22
- Zdarzenia i grupy zdarzeń, FreeRTOS od v8.00.

Kolejki we FreeRTOS [3]

Kolejka jest podstawową formą przekazywania informacji między zadaniami. Jest to bufor typu FIFO. Do kolejek umieszcza się zazwyczaj kopię przekazywanej danej zamiast odnośnika do niej chociaż obie wersje są możliwe. Każda kolejka ma zdeklarowaną ilość oraz typ elementów.

Poprzez referencje do kolejki można wysyłać komunikaty o zmiennym rozmiarze. Wykonuje się to poprzez umieszczenie w kolejce wskaźnika do wiadomości oraz wskaźnika do rozmiaru wiadomości.



Kolejki we FreeRTOS, c.d. [3]

API kolejek umożliwia podanie czasu blokowania.

Kiedy zadanie próbuje dokonać odczyt z pustej kolejki przechodzi do stanu **Blocked** aż do czasu zapisu do kolejki lub do wygaśnięcia czasu blokady.

Podobnie kiedy zadanie próbuje zapisu do pełnej kolejki przechodzi do stanu **Blocked** aż będzie dostępne miejsce lub wygaśnie czas blokady.

Semafor binarny [3]

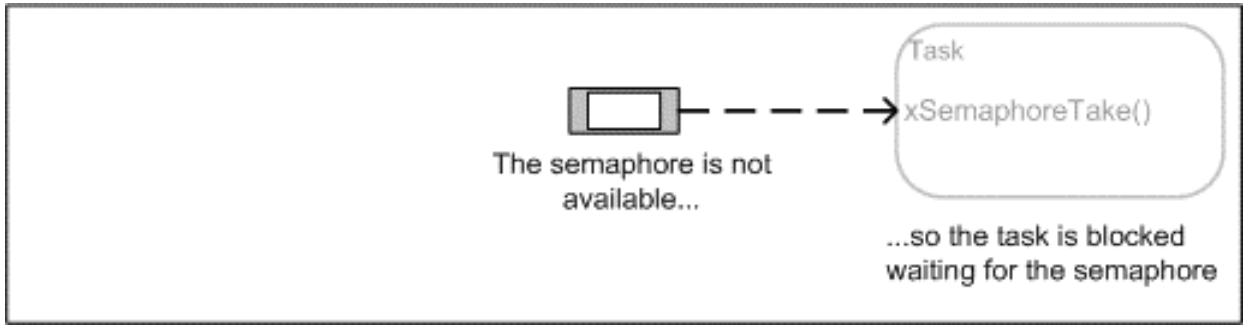
Semafor binarny jest wykorzystywany do synchronizacji oraz wzajemnego wykluczania zadań. Semafory binarne i mutexy są do siebie podobne z tą różnicą że mutexy zawierają mechanizm dziedziczenia priorytetów a semafory nie. Z tego powodu semafory lepiej nadają się do celów synchronizacji (zadań i przerwań) a mutexy do realizacji wzajemnego wykluczania.

API semaforów binarnych umożliwia zdeklarowanie czasu blokowania. Czas blokowania oznacza maksymalna liczbę tików czasowych zadania w stanie **Blocked** podczas próby przyjęcia semafora jeśli semafor nie jest dostępny natychmiast. Jeśli więcej niż jedno zadanie jest zablokowane oczekiwaniem na semafor wówczas zadanie o najwyższym priorytecie uruchomi się w momencie gdy semafor zostanie aktywowany.

Semafor binarny, c.d. [3]

Semafor binarny należy rozumieć jak kolejkę, która może przechowywać tylko jeden element, przy czym zawartość tego elementu nie ma znaczenia. Semafor może więc być tylko pełny albo pusty.

Przypadek użycia – zadanie ma obsługiwać element peryferyjny mikrokontrolera. Funkcja obsługi przerwania bloku peryferyjnego daje semafor np. w momencie pojawienia się danych co odblokowuje zadanie, które bierze semafor i przechodzi ze stanu **Blocked** do **Runing** w celu obsługi danego zdarzenia.



Semafor liczący [3]

Semafory liczące w odróżnieniu od binarnych mogą być traktowane jak kolejki o długości większej niż jeden których zawartość jest nieistotna. Typowo semafory liczące są używane w 2 następujących przypadkach:

• zliczanie zdarzeń, w tym przypadku funkcja obsługi zdarzenia daje semafor (zwiększając wartość semafora) za każdym razem kiedy zdarzenie wystąpi natomiast zadanie obsługi bierze semafor (zmniejszając wartość semafora) za każdym razem kiedy przetwarza dane zdarzenie,

• zarządzanie zasobami, w tym przypadku wartość wskazuje liczbę możliwych do użycia zasobów, aby uzyskać dostęp do zasobu zadanie musi najpierw otrzymać semafor (zmniejszając wartość semafora), kiedy wartość semafora będzie równa zeru oznacza że zasoby się wyczerpały, zadanie kończąc prace oddaje semafor i zwalnia zasoby.

Mutex [3]

Mutexy są binarnymi semaforami posiadającymi możliwość dziedziczenia priorytetów przez zadania wykorzystujące wspólny mutex.

Mutex jest tokenem który daje dostęp do zasobu. Kiedy zadanie chce skorzystać z zasobu musi najpierw dostać token, po zakończeniu używania zasobu token musi zostać zwrócony, co umożliwia dostęp do zasobu innym zadaniom.

API mutexu umożliwia zdefiniowanie czasu blokowania. Jest to czas liczony liczbą tików, kiedy zadanie pozostaje w stanie **Blocked** w przypadku próby wzięcia niedostępnego mutexu.

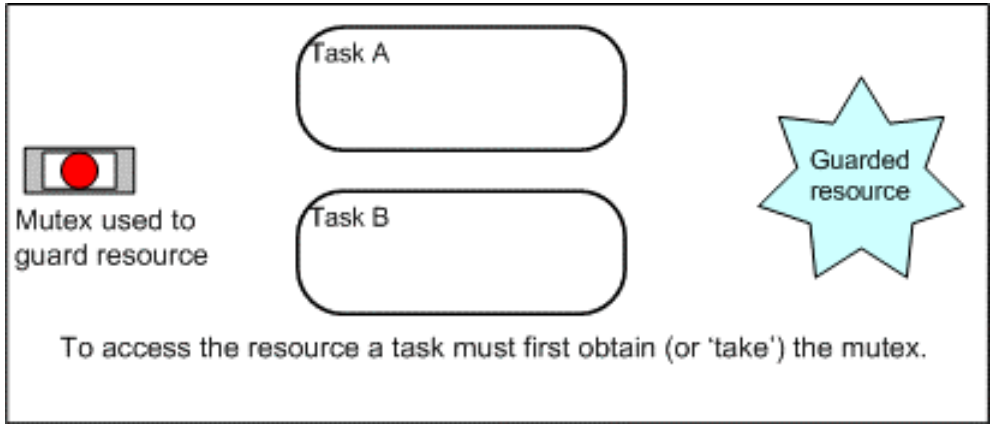
Inaczej niż w przypadku semafora binarnego, mutex wykonuje dziedziczenie priorytetów. To oznacza, że jeśli zadanie o wysokim priorytecie jest zablokowane przez zadanie o niższym priorytecie, które ma mutex wówczas priorytet zadania posiadającego mutex wzrasta do wartości priorytetu zadania zablokowanego oczekującego na ten sam mutex. Mechanizm dziedziczenia priorytetu jest wykonany w celu zapewnienia jak najszybszego wykonania zadań o wysokich priorytetach (przez odblokowanie zadań o niższych priorytetach blokujących zadania o wysokim priorytecie oczekujące na mutex).

Mutex, c.d. [3]

Mutexy nie mogą być używane z przerwań bo:

• zawierają mechanizm dziedziczenia priorytetów, który ma sens wyłącznie gdy mutex jest dawany z zadania a nie z przerwania,

• przerwanie nie może blokować się w czasie oczekiwania na chronione zasoby co akurat jest wykonywane przez mutex na który się oczekuje.



Mutex rekursywny [3]

Mutex używany rekursywnie może być brany wielokrotnie przez aktualnego właściciela. Taki mutex nie będzie dostępny ponownie aż do momentu oddania go tyle razy ile razy został wzięty. Taki mutex ma te same właściwości co zwykły mutex.

Bezpośrednie komunikaty międzyzadaniowe [3]

We FreeRTOS od wersji 8.22 możliwa jest uproszczona komunikacja między zadaniami z użyciem komunikatów bezpośrednich. Cechy takiej komunikacji są następujące:

• zwiększona szybkość obsługi oraz mniejsze zużycie pamięci RAM,

• komunikaty bezpośrednie mogą być używane jeśli jest tylko jedno zadanie będące odbiorcą tych komunikatów,

• w przypadku gdy komunikaty bezpośrednie są używane zamiast kolejek wówczas zadanie – odbiorca może przejść do stanu **Blocked** ale zadanie – nadawca nie może oczekiwać w stanie **Blocked** jeśli wysyłka nie może zostać zrealizowana natychmiastowo.

Timery programowe (ang. software timers) [3]

Timer programowy jest funkcjonalnością umożliwiająca wykonanie funkcji w pewnym ustawionym czasie w przyszłości. Czas pomiędzy startem timera a wywołaniem funkcji w przyszłości jest nazywany okresem timera. Funkcja po prostu zostanie wykonana po upływie okresu timera.

Timer programowy musi zostać najpierw utworzony a następnie dopiero można go używać.

Wywoływana funkcja jest wykonywana w kontekście zadania timera i dlatego:

• nie może przechodzić w stan zablokowania czyli nie może wywoływać np. vTaskDelay(),

• nie może mieć zerowego czasu blokowania przy dostępie do kolejek i semaforów.

Szczegóły w [3].

Bity zdarzeń (ang. Event Bits) [3]

Bity zdarzeń (które są też nazywane event flags) są używane do wskazywania, że dane zdarzenie wystąpiło. Aplikacja może na przykład:

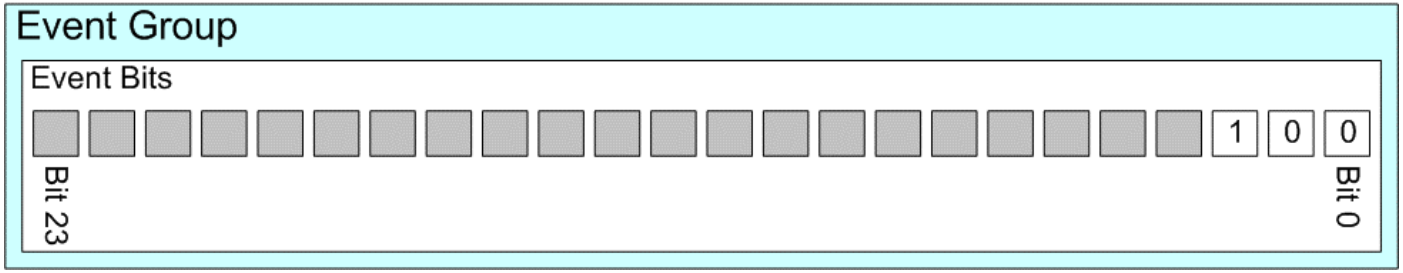
- zdefiniować bit który oznacza „otrzymano wiadomość” kiedy jest ustawiony na „1” oraz „nie ma wiadomości” kiedy bit jest ustawiony na „0”,
- zdefiniować bit który oznacza „aplikacja wstawiła do kolejki dane, które można wysłać do sieci” jeśli jest ustawiony na „1” oraz „nie ma wiadomości do wysłania” kiedy bit jest ustawiony na „0”,
- zdefiniować bit który oznacza „nadszedł czas wysłania informacji synchronizującej” jeśli jest ustawiony na „1” oraz „jeszcze nie upłynął czas do chwili synchronizacji” kiedy bit jest ustawiony na „0”,

Grupy zdarzeń (ang. Event Groups) [3]

Grupa zdarzeń jest zestawem bitów zdarzeń. Poszczególne, indywidualne bity są oznaczane numerami. Przykład z poprzedniego slajdu mógłby mieć numeracje od 0 do 2 dla kolejnych bitów zdarzeń.

Grupy zdarzeń są reprezentowane przez zmienne typu `EventGroupHandle_t`. Liczba bitów przechowywanych w grupie zdarzeń wynosi 8 jeśli zmienna `configUSE_16_BIT_TICKS` jest ustawiona na 1 lub 24 jeśli te zmienna jest ustawiona na 0.

Poszczególne, pojedyncze bity zdarzeń są typu `EventBits_t`.



API grup zdarzeń [3]

API grup zdarzeń umożliwiają między innymi to, że zadania mogą:

• ustawiać i zerować bity grup zdarzeń,

• wchodzić do stanu **Blocked** do czasu aż nie zostanie ustawiony dany bit zdarzenia.

Ze względu na powyższe właściwości grupy zdarzeń mogą służyć do synchronizacji zadań.

Dodanie systemu FreeRTOS do projektu [4]

Wykorzystanie sytemu we własnych projektach wykonywane jest poprzez:

- ściągnięcie systemu w wersji na daną rodzinę mikrokontrolera ze strony freertos.org,
- dodanie plików systemu FreeRTOS do projektu,
- dodanie ścieżek inkludowanych przez kompilator,
- ustawienie wielkości sterty (ang. heap) oraz stosu (ang. stack),
- dodanie kodu inicjalizującego (w main()),
- skonfigurowanie systemu w pliku FreeRTOSConfig.h
- stworzenie własnej aplikacji (utworzenie zadań, semaforów,....)

FreeRTOS we własnych projektach [3, 4]

Łatwiej jest jednak, co nawet jest zalecane przez autorów FreeRTOS, uruchomienie systemu poprzez:

- skopiowanie przykładu dla danego mikrokontrolera ze strony freetros.org,
- modyfikacja przykładu do wykorzystania we własnym projekcie.

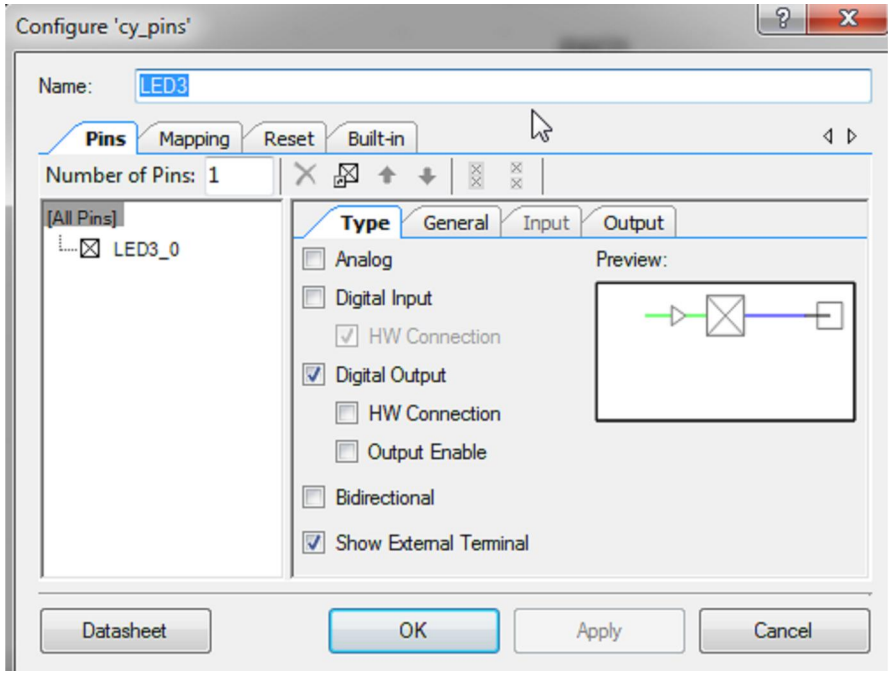
Przykład implementacji na podstawie [4]

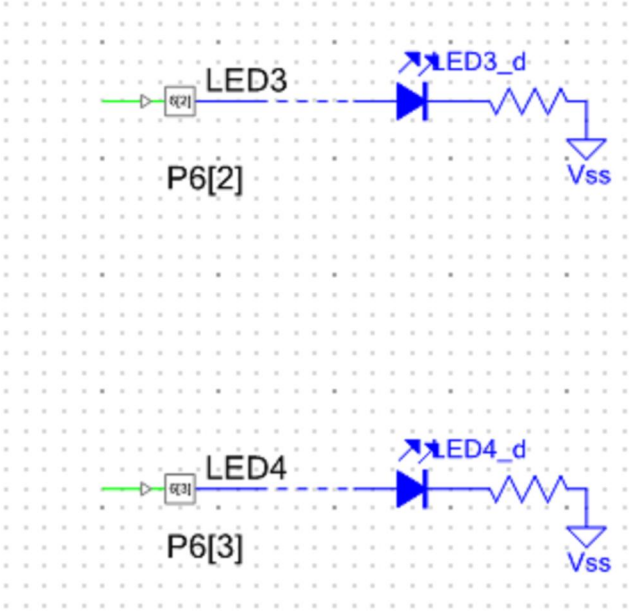
Czynności do wykonania:

1. Z linku **http://www.ue.eti.pg.gda.pl/~bpa/pusoc/freertos/** należy skopiować plik **CY8CKIT-050-GCC_v1_0.zip** i rozpakować w katalogu **d:\Designs\FreeRTOS**
2. Należy uruchomić **IDE PSOC Creator** i wczytać przestrzeń roboczą **D:\Designs\FreeRTOS\CY8CKIT-050-GCC\ CY8CKIT-050-GCC.cywrk**
3. W panelu **Workspace Explorer** klikamy prawym klawiszem myszki na projekt **'NewDesign'** i wybieramy **SetAsActiveProject**.
4. Zamykamy wszystkie otwarte dokumenty w oknie głównym a następnie otwieramy schemat projektu **NewDesign** – plik o nazwie **TopDesign.cysch**.
5. Ze schematu należy skasować wszystkie strony pozostawiając wyłącznie **Switches** oraz **LEDs**.

Przykład implementacji na podstawie [4], c.d.

5. Na schemacie **LEDs** pozostawiamy tylko wyjścia z diodami LED zmieniając jednocześnie typ wyjścia na nie posiadający połączenia sprzętowego:





Przykład implementacji na podstawie [4] , c.d.

6. Przy próbie zbudowania projektu (Shift-F6) pojawi się w pliku **MainTask.c** błąd jak na rysunku poniżej. Spowodowany jest on usunięciem bloku LED3_Control ze schematu. Aby usunąć błąd należy zamienne skorzystać z funkcji dostępu po wyprowadzenia I/O: **`LED3_Write(~LED3_Read());`**

```
71  void Main_Task( void *pvParameters )
72  {
73      /* Init the LED controller - enable software control */
74      LED3_Control_Set_Channel( LED3_Control_USE_SOFTWARE );
75
76      for( ; ; )
77      {
78          /* Delay 1s */
79          vTaskDelay( configTICK_RATE_HZ );
80
81          /* Toggle the LED3 on/off -
82          LED3_Control_Write( ! LED3_C
```

```
71  void Main_Task( void *pvParameters )
72  {
73      /* Init the LED controller - enable software control */
74
75      for( ; ; )
76      {
77          /* Delay 1s */
78          vTaskDelay( configTICK_RATE_HZ );
79
80          /* Toggle the LED3 on/off - creates a 0.5Hz frequency */
81          LED3_Write(~LED3_Read());
```

Przykład implementacji na podstawie [4] , c.d.

Po zmianach wprowadzonych wg poprzednich slajdów projekt można zbudować (Shift-F6) oraz wgrać na płytkę (Ctrl-F5). Efektem będzie mruganie diody LED3 co 2 sekundy. Należy zwrócić uwagę, że:

- w pliku **`main.c`** mamy stworzone zadanie **`Main_Task`** (funkcja **`xTaskCreate`**) a następnie uruchomiony jest algorytm kolejkowania (funkcja **`vTaskStartScheduler()`**),
- deklaracja zadania **`Main_Task`** jest umieszczona w pliku nagłówkowym **`Task_Defs.h`** natomiast definicja w pliku **`Main_Task.c`**,
- funkcja **MAIN_TASK** ma w nieskończonej pętli 2 polecenia:
  - oczekiwanie `vTaskDelay(configTICK_RATE_HZ);` oraz
  - zmianę stanu diody LED3 `LED3_Write(~LED3_Read());` ,

Przykład implementacji na podstawie [4] , c.d. – zadanie do wykonania

W kolejnym kroku należy dokonać następujących modyfikacji:

• dodać zadanie o nazwie **`Task_LED3_Fast`**, które ma mrugać diodą LED3 10 razy szybciej niż w zadaniu Main_Task czyli z częstotliwością 5Hz,

• w zadaniu **`Main_Task`** należy ustawić priorytet równy 1 a nowe zadanie ma mieć priorytet wyższy o 1 niż zadanie główne,

• co obserwujemy po zbudowaniu projektu?

Dalszy ciąg zmian:

• porządkujemy projekt poprzez:

  • stworzenie zadania **`TASK_LED3_Slow`** i przeniesienie funkcjonalności z zadania głównego,

  • w zadaniu **`Main_Task`** warunkowo zawieszamy i wznawiamy działanie zadań w zależności od wartości poru **`P6[1]`**,

Plik Task_LED3_Fast.c

```
#include "FreeRTOS.h"
#include "task.h"
#include <event_groups.h>

/* Include PSoC system and component APIs and defines */
#include <project.h>

/* Include application function declarations and defines */
#include <Task_Defs.h>
void Task_LED3_Fast( void *pvParameters )
{

   for( ; ; )
   {
       /* Delay 0.1s */
       vTaskDelay( configTICK_RATE_HZ/10 );
       /* Toggle the LED3 on/off - creates a 5Hz frequency */
       LED3_Write(~LED3_Read());
   } /* for( ; ; ) */
}
```

Plik Task_LED3_Slow.c

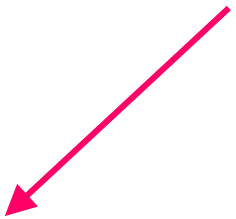
```
#include "FreeRTOS.h"
#include "task.h"
#include <event_groups.h>

/* Include PSoC system and component APIs and defines */
#include <project.h>

/* Include application function declarations and defines */
#include <Task_Defs.h>
void Task_LED3_Slow( void *pvParameters )
{
    for( ; ; )
    {
        /* Delay 0.5s */
        vTaskDelay( configTICK_RATE_HZ/2 );
        /* Toggle the LED3 on/off - creates a 1Hz frequency */
        LED3_Write(~LED3_Read());
    } /* for( ; ; ) */
}
```

Plik Main_Task.c

```
#include "FreeRTOS.h"
#include "task.h"
#include "event_groups.h"
/* Include PSoC system and component APIs and defines */
#include <project.h>
/* Include application function declarations and defines */
#include <Task_Defs.h>
extern TaskHandle_t TaskFast,TaskSlow;
void Main_Task( void *pvParameters )
{
    for( ; ; )
    {
        if (P6_1_Read())
        {
            vTaskSuspend(TaskSlow);
            vTaskResume(TaskFast);
         }
         else
         {
             vTaskSuspend(TaskFast);
             vTaskResume(TaskSlow);
         }
//vTaskDelay( configTICK_RATE_HZ/10 );
    } /* for( ; ; ) */
} /* Main Task */
```



Do późniejszego testowania zmiany wzajemne priorytetów zadań oraz wstawienie funkcji oczekiwania w zadaniu Main_Task

Zmiany w pliku main.c

```
...
TaskHandle_t TaskFast,TaskSlow;
int main( void )
...
taskCreated = xTaskCreate(
        Task_LED3_Fast,                 /* Task function */
        "Task_Fast",                      /* Task name (string) */
        MAIN_TASK_STACK_SIZE,           /* Task stack, allocated from heap */
        NO_TASK_ARG,                    /* Argument to task function */
        2,                              /* Priority */
        &TaskFast );                    /* Task handle */

taskCreated = xTaskCreate(
        Task_LED3_Slow,                 /* Task function */
        "Task_Slow",                      /* Task name (string) */
        MAIN_TASK_STACK_SIZE,           /* Task stack, allocated from heap */
        NO_TASK_ARG,                    /* Argument to task function */
        2,                              /* Priority */
        &TaskSlow );                    /* Task handle */
...
```

Do późniejszego testowania zmiany wzajemne priorytetów zadań oraz wstawienie funkcji oczekiwania w zadaniu Main_Task

Zmiany w pliku Task_Defs.h

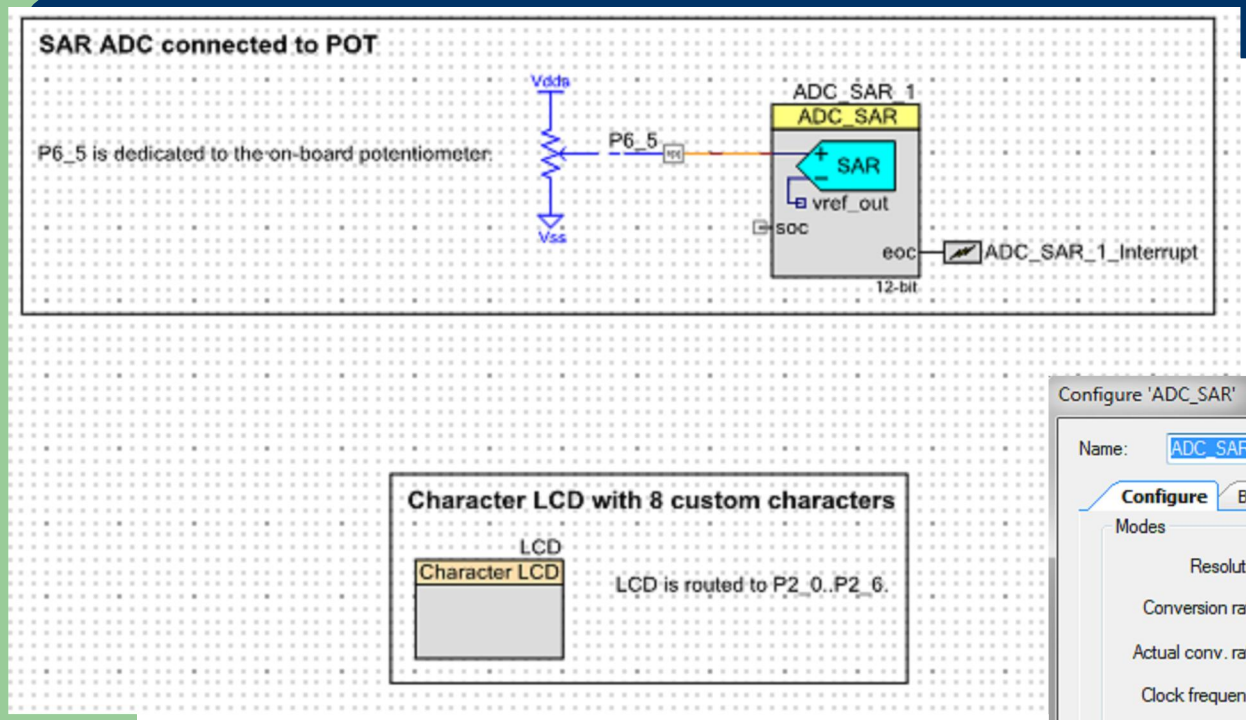
```
...
/*
Forward definitions of all task start functions.
*/
void Main_Task( void * );
void Task_LED3_Fast( void * );
void Task_LED3_Slow( void * );
```

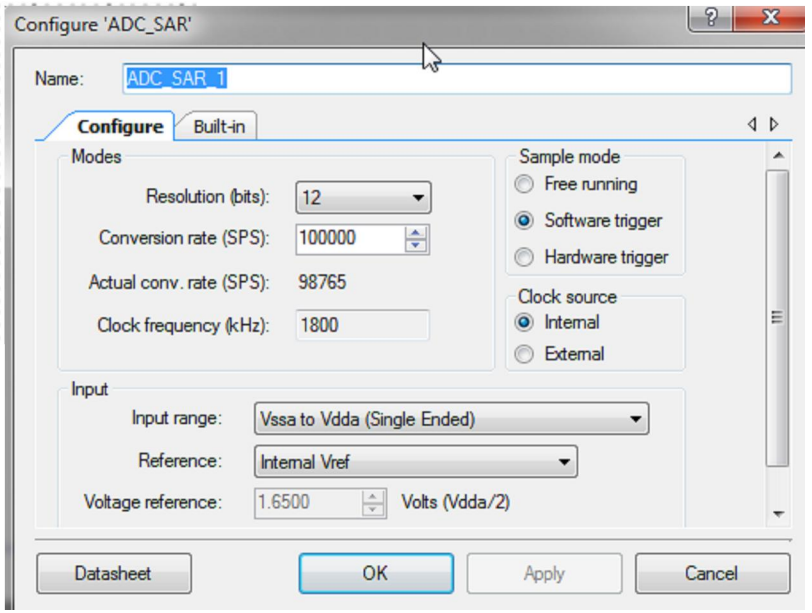
Wykorzystanie kolejki do przekazywania danych pomiędzy zadaniami – czynności do wykonania

Modyfikacja przykładu polega na dodaniu zadania odczytującego napięcie z potencjometru i umieszczenie wartości w kolejce a następnie dodaniu zadania, które tą wartość z kolejki odczytuje i wyświetla na wyświetlaczu LCD. Niezbędne do wykonania kroki:

- dodanie wyświetlacza oraz SAR ADC do schematu,
- dodanie definicji zadania Task_ADC,
- dodanie definicji zadania Task_LCD,
- utworzenie powyższych zadań w pliku main.c, dodatkowo zmiany w deklaracji zmiennych i i pliku nagłówkowym.

Dodany schemat





Plik Task_ADC.c

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include <project.h>
#include <Task_Defs.h>
extern QueueHandle_t xQueueADC;
void Task_ADC( void *pvParameters )
{
   int16 wynik_ADC;
   xQueueADC = xQueueCreate(2, sizeof(int16));
   //ADC start
   ADC_SAR_1_Start();
   for( ; ; )
   {
      ADC_SAR_1_StartConvert();
      wynik_ADC = ADC_SAR_1_IsEndConversion(1);
      wynik_ADC = ADC_SAR_1_GetResult16();
      wynik_ADC = ADC_SAR_1_CountsTo_mVolts(wynik_ADC);
      xQueueSend(
           xQueueADC,   // kolejka do której następuje przekazanie wartości,
           (void *) &wynik_ADC, // wskaźnik do zmiennej przekazywanej do kolejki,
           (portTickType) 100); // czas pozostania w stanie Blocked przy pełnej Q
       vTaskDelay(configTICK_RATE_HZ/10);
   }
}
```

Plik Task_LCD.c

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "stdio.h"
/* Include PSoC system and component APIs and defines */
#include <project.h>
/* Include application function declarations and defines */
#include <Task_Defs.h>
extern QueueHandle_t xQueueADC;
void Task_LCD( void *pvParameters )
{
   int16 wynik_ADC;
   char ciag[32];
   LCD_Init();
   for( ; ; )
   {
      xQueueReceive(
         xQueueADC,     // kolejka z której następuje przekazanie wartości,
         &wynik_ADC,    // wskaźnik do zmiennej do której zostaje przekazana wartość z kolejki,
         (portTickType) 100);// czas pozostania w stanie Blocked jeśli kolejka jest pusta,

      LCD_Position(0,0);
      sprintf(ciag,"Vpot = %d mV  ",wynik_ADC);
      LCD_PrintString(ciag);
      vTaskDelay( configTICK_RATE_HZ/10 );
   }
}
```

Zmiany w pliku main.c

```
...
QueueHandle_t xQueueADC;
...
taskCreated = xTaskCreate(
        Task_LCD,                   // Task function
        "Task_LCD",                 // Task name (string)
        256,                        // Task stack, allocated from heap
        NO_TASK_ARG,                // Argument to task function
        1,                          // Priority
        TASK_HANDLE_UNUSED);        // Task handle


taskCreated = xTaskCreate(
        Task_ADC,                    // Task function
        "Task_ADC",                  // Task name (string)
        256,                         // Task stack, allocated from heap
        NO_TASK_ARG,                 // Argument to task function
        1,                           // Priority
        TASK_HANDLE_UNUSED);         // Task handle
```

Zmiany w pliku Task_Defs.h

```
...
/*
Forward definitions of all task start functions.
*/
void Main_Task( void * );
void Task_LED3_Fast( void * );
void Task_LED3_Slow( void * );
void Task_ADC( void * );
void Task_LCD( void * );
```