INSTYTUT INFORMATYKI
Wydział IEiT AGH

Ćwiczenie laboratoryjne lub pokaz zdalny

Zastosowanie systemu operacyjnego czasu rzeczywistego – na przykładzie FreeRTOS

Nazwa kodowa: **`rtos`**. Wersja **`20231031.0`**

Ada Brzoza
ada.brzoza@agh.edu.pl

1 Wstęp

1.1 Cel ćwiczenia

Celem ćwiczenia jest:

- nabycie umiejętności w posługiwaniu się prostym systemem operacyjnym czasu rzeczywistego,
- poznanie mechanizmów działania i sposobu implementacji podstawowych elementów systemu operacyjnego czasu rzeczywistego.

1.2 Wymagania wstępne

Wymagania wstępne do przeprowadzenia ćwiczenia są następujące:

- umiejętność posługiwania się językiem C,
- rozumienie pojęcia kompilacji skrośnej (*cross-compiling*) i umiejętność posługiwania się narzędziami do tego rodzaju kompilacji,
- umiejętność rozróżnienia typowych interfejsów zewnętrznych komputera PC,
- umiejętność rozróżniania rejestrów wewnętrznych mikroprocesora oraz rejestrów układów peryferyjnych mikrokontrolera lub systemu mikroprocesorowego,
- umiejętność posługiwania się narzędziami i elementami projektu poznanymi na laboratorium nr 2, 3 i 4.

1.3 Stanowisko testowe i plan pracy

Ćwiczenie przeprowadzone jest z wykorzystaniem platformy sprzętowej **STM32F429 Nucleo-144** z mikrokontrolerem z rdzeniem ARM Cortex-M4. Od strony programowej użyjemy następujących narzędzi:

- zintegrowanego środowiska STM32CubeIDE,
- programu terminalowego, np. **puTTY**.

1.4 Materiały pomocnicze

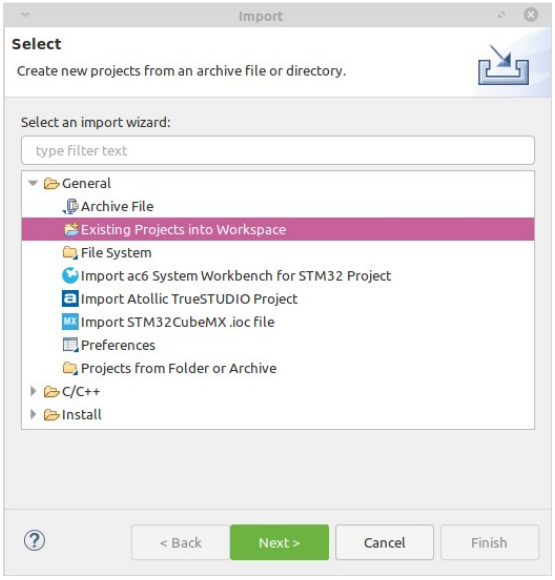
Do wykonania ćwiczenia niezbędne lub przydatne są następujące materiały:

- zmodyfikowany w ramach ćwiczenia nr 4 projekt startowy ***nucleo-FreeRTOS***,
- dokumentacja systemu operacyjnego FreeRTOS: https://www.freertos.org/ ,
- nota katalogowa (*datasheet*) mikrokontrolera STM32F429ZI,
- instrukcja obsługi (*reference manual*) mikrokontrolera STM32F429ZI,
- instrukcja obsługi (user manual) zestawu STM32F429 Nucleo-144,

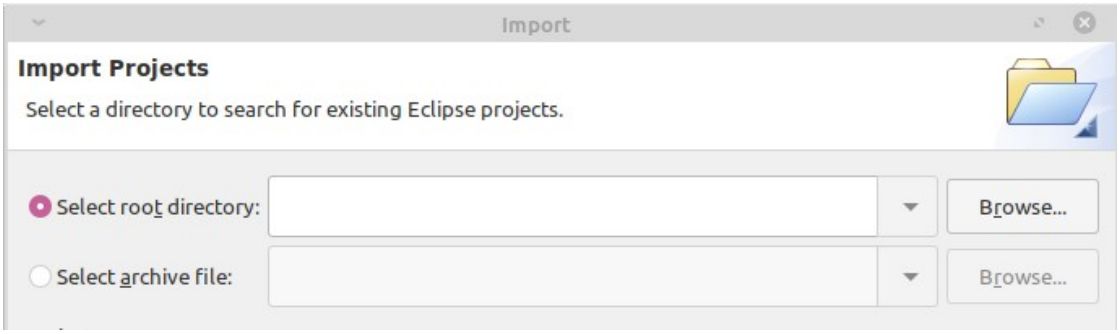
- zestaw przykładów dla STM32F429 Nucleo-144.

2 Import i otwarcie projektu

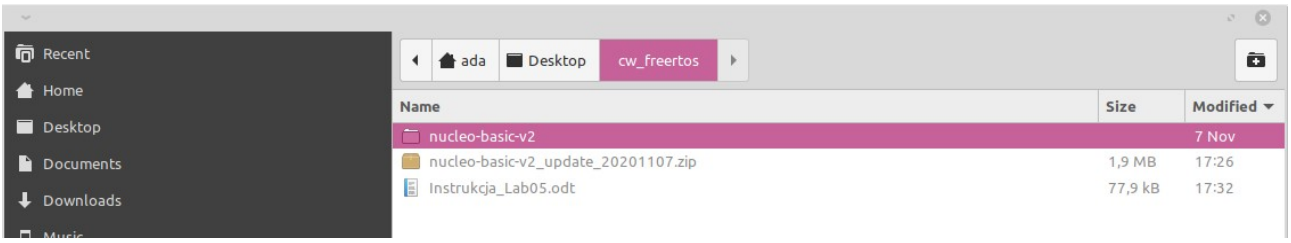
Pobrane projekt startowy nucleo-basic-v2 należy rozpakować i importować w środowisku STM32CubeIDE przy pomocy polecenia ***File → Import…*** a następnie rozwijamy sekcję ***General → Existing Projects into Workspace → Next***.



W kolejnym oknie dialogowym *Import*, zaznaczamy ***Select root directory*** i wciskamy ***Browse…***

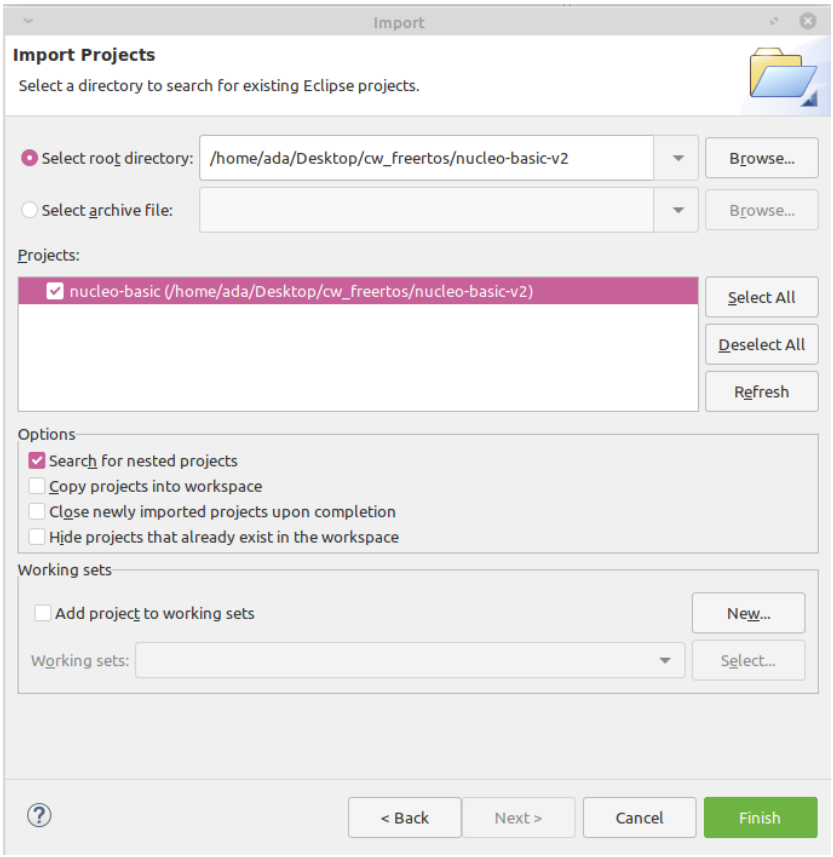


Wskazujemy katalog z pobranym projektem:



Następnie zatwierdzamy przyciskiem ***Finish***.

Po wybraniu katalogu wciskamy przycisk ***Open***. Powinniśmy zobaczyć okienko podsumowania, np. takie jak poniżej. Jeśli wszystko się zgadza, wciskamy przycisk ***Finish***.



Po zaimportowaniu projektu warto przeprowadzić jego próbną kompilację np. poprzez klikniecie na ikonę „młotek” na pasku głównego okna STM32Cube IDE lub wybranie odpowiedniej pozycji z menu kontekstowego projektu. Następnie możemy przeprowadzić uruchomienie programu na platformie sprzętowej np. przez rozpoczęcie debugowania, klikając ikonkę ikonka „żuczek”. Przed próbnym uruchomieniem dobrze jest nawiązać połączenie z płytką Nucleo przez port szeregowy. W systemie Linux port szeregowy z płytki STM32 Nucleo zwykle jest widoczny jako urządzenie **`/dev/ttyACM0`**. Jako program terminalowy można wykorzystać np.: PuTTY. Parametry transmisji są następujące: prędkość 115200, 8 bitów danych, 1 bit stopu, brak bitu parzystości i wyłączona kontrola przepływu.

Jeśli proces próbnego uruchomienia przebiegnie prawidłowo to po jego zakończeniu znajdująca się na płytce STM Nucleo a po chwili zielona dioda oznaczona jako LD1 będzie błyskać. W oknie programu terminalowego wyświetlone zostaną komunikaty:

```
Nucleo-144 F429ZI FreeRTOS project
Funkcja xprintf dziala.
Zwykly printf dziala.
MX_LWIP_Init, be patient…
entering StartDefaultTask main loop
```

3 System operacyjny FreeRTOS

FreeRTOS jest prostym systemem operacyjnym czasu rzeczywistego (*Real Time Operating System*, RTOS) przeznaczonym głównie do zastosowań w urządzeniach mikroprocesorowych, gdzie istnieją

silne ograniczenia na ilość dostępnych zasobów: głównie mocy obliczeniowej i pamięci operacyjnej. Podstawowa wersja systemu FreeRTOS rozprowadzana jest na licencji MIT (jednej z najmniej restrykcyjnych licencji).

FreeRTOS jest szeroko stosowany w urządzeniach przemysłowych, gdzie istotnymi czynnikami są: niezawodność, stabilność działania i czas reakcji na zdarzenia. Pod koniec 2017 roku system operacyjny FreeRTOS stał się częścią *AWS open source project*

https://aws.amazon.com/freertos/

przez co stał się lepiej rozpoznawalny jako element urządzeń Internetu Rzeczy (IoT).

W odróżnieniu od typowych systemów operacyjnych na platformy aplikacyjne, FreeRTOS jest kompilowany razem z kodem, który będzie na nim wykonywany. Najważniejsze funkcje tego systemu sprowadzają się przede wszystkim do:

- nadzorowania wykonywania zadań (*tasks*) przez moduł szeregujący (*scheduler*)
- zapewnienia komunikacji pomiędzy zadaniami oraz pomiędzy zadaniami a funkcjami obsługi przerwań przez wbudowane mechanizmy kolejkowania (*queue*),
- dostarczenia programiście mechanizmów nadzorowania dostępu do zasobów współdzielonych przy pomocy semaforów (*sempahore*) oraz mutexów.

Jedynymi fragmentami systemu FreeRTOS specyficznymi dla danej platformy (mikrokontrolera) są funkcje i makrodefinicje odpowiedzialne za obsługę układu czasowo-licznikowego (timera) oraz ewentualnie przerwań programowych. Warstwa abstrakcji dla pozostałych układów peryferyjnych nie jest nadzorowana przez system, dzięki czemu nie ma dodatkowego narzutu na komunikację pomiędzy warstwą sprzętową a kodem odpowiadającym za funkcjonalność. Ma to duże znaczenie głównie w prostych mikrokontrolerach o małych ilościach pamięci operacyjnej i stosunkowo niewielkiej mocy obliczeniowej.

Jednostką czasu w systemie operacyjnym FreeRTOS jest ***tick***, co w wolnym tłumaczeniu oznacza „tyknięcie”. Czas wszystkich opóźnień i oczekiwania liczony jest w liczbie tyknięć zegara systemowego. Typowy interwał czasu pomiędzy poszczególnymi *tickami* systemu przyjmuje się w przedziale 1-10 ms co odpowiada częstotliwościom 100-1000 Hz. W projekcie, na którym wykonywane jest ćwiczenie, *tick* liczony jest z częstotliwością 1000 Hz, a zmiana tej częstotliwości może być przeprowadzona przez edycję makra **`configTICK_RATE_HZ`** w pliku **`FreeRTOSConfig.h`**. Należy pamiętać, że im wyższa częstotliwość tykania zegara systemu FreeRTOS, tym wyższa będzie rozdzielczość czasowa zliczanych opóźnień i czasów oczekiwania, lecz także tym większy będzie narzut czasowy wnoszony przez system operacyjny. Bierze się to stąd, że przy domyślnych ustawieniach każde tyknięcie zegara systemowego oznacza obsługę przerwania od układu peryferyjnego odmierzającego czas.

3.1 Zadania

Uwaga: pełna dokumentacja API tworzenia i kontroli zadań znajduje się na stronie www.freertos.org.

Podstawowymi elementami systemu operacyjnego FreeRTOS są zadania (***tasks***). Zadania są

w rzeczywistości funkcjami zawierającymi najczęściej nieskończone pętle programowe. Ich wykonywanie nadzorowane jest przez ***scheduler***. Zadania można dynamicznie (tj. w czasie działania systemu operacyjnego) m.in.: tworzyć (*create*), usuwać (*delete*) oraz zawieszać i wznawiać ich działanie (*suspend*, *resume*). Oprócz tego, można wstrzymywać ich działanie na określony czas (*delay*) w taki sposób, by nie zajmowały czasu mikroprocesora.

Każde zadanie ma własny stos, którego rozmiar definiujemy przy tworzeniu zadania. Stos dla zadań alokowany jest na stercie (*heap*) systemu operacyjnego FreeRTOS. Zależnie od konfiguracji może to być albo ta sama sterta, na której alokujemy pamięć funkcją *malloc* lub, częściej, osobno zarządzana sterta systemu FreeRTOS. Warto pamiętać, że każda zmienna lokalna wewnątrz funkcji stanowiącej zadanie oraz każde wywołanie funkcji, szczególnie z dużą liczbą argumentów, korzysta ze stosu zadania.

Zadania mają obligatoryjnie przypisany **priorytet** określany liczbą naturalną. Zero oznacza najniższy priorytet przypisany domyślnie do zadania bezczynności (*idle task*) wykonywanego zawsze wtedy, gdy wszystkie inne utworzone zadania albo są wstrzymane, albo ich działanie jest zawieszone. Zadania o wyższym priorytecie mogą przerywać wykonywanie zadań o niższym priorytecie – domyślnie system operacyjny FreeRTOS działa z wywłaszczaniem (*preemption*). Wysokie priorytety oczywiście przypisuje się zadaniom, których wykonywanie musi ściśle mieścić się w zadanych przedziałach czasowych.

Dodatkowo, zadanie może mieć przypisany własny ***handler***. Jest to obiekt typu ***xTaskHandle*** reprezentujący zadanie. Jest on przydatny m.in. przy zawieszaniu oraz usuwaniu zadania „z zewnątrz” czyli przez inne zadanie.

Uwaga! Zadanie nie może kończyć się przez wyjście ze stanowiącej je funkcji (np. przez `return` lub zakończenie funkcji). Aby zakończyć zadanie należy zastosować funkcję `vTaskDelete`. Podanie do funkcji `vTaskDelete` argumentu NULL spowoduje zakończenie tego zadania, które ją wywołało. Alternatywnie w argumencie do `vTaskDelete` możemy zastosować *handler* zadania. Próba zakończenia zadania przez wyjście z funkcji spowoduje zawieszenie się oprogramowania.

3.1.1 Tworzenie zadania

Podstawowym sposobem tworzenia zadań działającym od najstarszych wersji FreeRTOSa jest wywołanie funkcji ***xTaskCreate***. Jej deklaracja przedstawiona jest poniżej.

```
BaseType_t xTaskCreate(    TaskFunction_t pvTaskCode,
                           const char * const pcName,
                           unsigned short usStackDepth,
                           void *pvParameters,
                           UBaseType_t uxPriority,
                           TaskHandle_t *pxCreatedTask
                       );
```

Parametry, które do niej podajemy są następujące:

- ***pvTaskCode***: wskaźnik do funkcji stanowiącej zadanie,
- ***pcName***: wskaźnik do początku łańcucha znakowego z nazwą zadania (można przekazać NULL, jeśli nie używamy nazwy zadania),

- ***usStackDepth***: rozmiar stosu zadania. Minimalny rozmiar stosu zdefiniowany jest w makrodefinicji *configMINIMAL_STACK_SIZE*,
- ***pvParameters***: opcjonalne parametry, które możemy przekazać do kodu zadania,
- ***uxPriority***: priorytet zadania w zakresie od 0 (najniższy) do *configMAX_PRIORITIES* (makrodefinicja z *FreeRTOSConfig.h*),
- ***pxCreatedTask***: wskaźnik do *handlera* zadania (jeśli rezygnujemy z *handlera*, przekazujemy tutaj *NULL*).

Jeśli zadanie zostanie poprawnie utworzone, funkcja zwróci wartość *pdPASS*. Typową przyczyną niepowodzenia w tworzeniu zadania jest próba alokacji zbyt dużego stosu dla tego zadania.

Najprostszym przykładem wywołania funkcji *xTaskCreate* może być

```
xTaskCreate(myTask,NULL,configMINIMAL_STACK_SIZE+20,NULL,2,NULL);
```

tworzący zadanie z funkcji o nazwie *myTask*, którego rozmiar stosu jest o 20 bajtów większy od minimalnego, priorytet tego zadania wynosi 2 oraz nie ma ono ani swojego *handlera* ani przypisanej nazwy.

Funkcja stanowiąca zadanie musi zwracać *void* oraz pobierać wskaźnik do typu *void*, który możemy przeznaczyć do przekazania opcjonalnych parametrów dla zadania. W omawianym przypadku, deklaracja *myTask* może być następująca:

```
void myTask (void *parameters);
```

3.1.2 Wstrzymywanie wykonywania zadania

FreeRTOS oferuje dwie funkcje służące do wstrzymywania wykonywania zadania na określony czas.

Funkcja ***vTaskDelay*** realizuje najprostszy rodzaj opóźnienia. Jego czas jest liczony od chwili wywołania funkcji i trwa tyle tyknięć zegara systemowego, ile przekażemy w jedynym argumencie do tej funkcji. Np.

```
vTaskDelay(500);
```

wstrzyma wykonywanie zadania na 500 tyknięć systemu operacyjnego. W projekcie, którym będziemy się posługiwać przy realizacji ćwiczenia, oznacza to opóźnienie wynoszące 500 milisekund, ponieważ interwał pomiędzy tyknięciami ustawiony został na 1 ms.

Druga funkcja realizująca opóźnienie nosi nazwę ***vTaskDelayUntil*** i typowo używa się jej do wykonania pewnego zadania co określony czas. Jej deklaracja jest następująca:

```
void vTaskDelayUntil(
      portTickType *pxPreviousWakeTime,
      portTickType xTimeIncrement
      );
```

gdzie:

- ***pxPreviousWakeTime*** to wskaźnik do zmiennej typu *portTickType* przechowującej

moment, w którym zadanie było ostatnio wznowione. Zmienna ta musi zostać zainicjalizowana przed pierwszym użyciem. Typowo wartość inicjalizująca jest aktualną wartością czasu systemowego,

- ***xTimeIncrement*** oznacza okres wznawiania działania zadania mierzony w tickach systemu operacyjnego.

Rozważmy dla przykładu hipotetyczną sytuację, że w pewnym zadaniu o nazwie *sensorTask* chcemy pobierać dokładnie co 1 sekundę dane z czujników dołączonych do mikrokontrolera i do pobierania tych danych służy funkcja o nazwie *readSensors*. Powiedzmy, że czas jej wykonywania nie jest stały i waha się w przedziale 100-400 ms. Aby zapewnić jej wywołania co dokładnie 1 s, trzeba użyć funkcji *vTaskDelayUntil* np. w taki sposób jak w poniższym przykładowym kodzie.

```
void sensorTask(void *params)
{
	portTickType xLastWakeTime;
	xLastWakeTime = xTaskGetTickCount(); //pobieramy aktualny czas
	while(1)
	{
		vTaskDelayUntil( &xLastWakeTime, 1000 );
		readSensors();
	}
}
```

Omówione tutaj funkcje realizują jawne opóźnienie.

Wstrzymanie wykonywania zadania następuje także w wielu innych sytuacjach takich jak np. oczekiwanie na:

- zwolnienie semafora,
- odbiór elementu z kolejki (gdy kolejka jest pusta, a my oczekujemy jak z jej „drugiej strony” zostaną wpisane nowe dane),
- wpisanie elementu do kolejki (gdy kolejka jest pełna i musimy poczekać aż z jej „drugiej strony” zostaną odczytane dane tym samym zwalniając miejsce).

Z punktu widzenia systemu operacyjnego, zarówno zadanie będące w trakcie oczekiwania wywołanego funkcjami *vTaskDelay* lub *vTaskDelayUntil* jak i czekającego na zwolnienie kolejki czy semafora praktycznie nie zajmuje czasu mikroprocesora i pozwala w tym czasie na działanie innych zadań. Jeśli zadanie nie będzie realizowało żadnego opóźnienia (będzie bez przerwy przetwarzało dane), może ono „zagłodzić” zadania o niższym priorytecie, w tym *idle task*. Jest to częsta przyczyna niepoprawnego działania programu, na którą warto zwracać uwagę tworząc własne aplikacje.

Uwaga. Używając systemu operacyjnego FreeRTOS typowo nieopłacalne jest używanie funkcji opóźniającej **`HAL_Delay`**. Funkcja **`HAL_Delay`** zabiera czały dostępny czas procesora przeznaczony

dla zadania (tak jak obliczenia) i nie oddaje go dyspozycji innych zadań. Może to nawet prowadzić do „zagłodzenia” (zaprzestania wykonywania) zadań o niższych priorytetach.

3.2 Kolejki

Uwaga: pełna dokumentacja API tworzenia i kontroli kolejek znajduje się na stronie www.freertos.org w dziale API Reference → Queues.

Wbudowane w FreeRTOS mechanizmy kolejkowania są jednym z podstawowych sposobów efektywnej i bezpiecznej wymiany danych pomiędzy zadaniami. Tworzenie kolejki odbywa się przez wywołanie funkcji ***xQueueCreate*** o następującej deklaracji:

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength,
                            UBaseType_t uxItemSize );
```

gdzie:

- ***uxQueueLength***: oznacza maksymalną ilość elementów, jaką może pomieścić kolejka,
- ***uxItemSize***: określa rozmiar pojedynczego elementu. W praktyce do określenia rozmiaru elementu najczęściej posługujemy się tutaj operatorem *sizeof*.

Funkcja zwraca wartość dla obiektu typu *xQueueHandle* reprezentującego kolejkę. Jeśli po wywołaniu *xQueueCreate* wartość w inicjalizowanym obiekcie typu *xQueueHandle* będzie *NULL*, oznacza to błąd w tworzeniu kolejki (np. brak wystarczającej ilości pamięci).

Gdy kolejka jest już utworzona, można dodawać do niej elementy oraz odbierać je. Najbardziej podstawową makrodefinicją służącą do umieszczania elementu w kolejce jest ***xQueueSend***:

```
portBASE_TYPE xQueueSend(
                              xQueueHandle xQueue,
                              const void * pvItemToQueue,
                              portTickType xTicksToWait
                          );
```

gdzie:

- ***xQueue*** pobiera wartość z obiektu reprezentującego kolejkę i zainicjalizowanego w momencie jej tworzenia,
- ***pvItemToQueue*** to wskaźnik do obiektu, który chcemy umieścić w kolejce,
- ***xTicksToWait*** to czas liczony w *tickach*, który dajemy funkcji *xQueueSend* na umieszczenie nowego elementu do kolejki; jeśli kolejka będzie pełna, to tyle czasu będzie trwało oczekiwanie na zwolnienie miejsca na nowy element, natomiast jeśli będzie pusta, to element zostanie umieszczony w niej od razu, bez oczekiwania.

Makrodefinicja *xQueueSend* zwróci wartość *pdTRUE*, jeśli nowy element zostanie poprawnie dodany do kolejki.

W parametrze *xTicksToWait* można przekazywać także wartości „specjalne”. Wartość 0 będzie oznaczała, że jeśli kolejka będzie pełna, kod makra ma od razu rezygnować z umieszczania nowego

elementu w kolejce. Inna wartość specjalna to *portMAX_DELAY* oznaczająca nieskończony czas oczekiwania (do skutku) na wolne miejsce w kolejce.

Odbieranie elementów z kolejki można przeprowadzić przy pomocy makra ***xQueueReceive***,

```
BaseType_t xQueueReceive(
                              QueueHandle_t xQueue,
                              void *pvBuffer,
                              TickType_t xTicksToWait
                            );
```

gdzie:

- ***xQueue*** to wartość z obiektu reprezentującego kolejkę,
- ***pvBuffer*** to wskaźnik do bufora, w którym znajdzie się odebrany element,
- ***xTicksToWait*** to czas oczekiwania na pojawienie się elementu gotowego do odbioru.

Jeśli wartością zwracaną będzie *pdTRUE*, to element został poprawnie odebrany w wyznaczonym czasie. Podobnie jak w przypadku *xQueueSend*, tutaj także możemy zdefiniować czas oczekiwania *xTicksToWait* wynoszący dowolną wartości od 0 (jeśli element czekał w kolejce pobieramy go, a jeśli nie – przechodzimy dalej) do *portMAX_DELAY* (wtedy czekamy do skutku aż pojawi się element do odebrania).

Omówione tutaj makrodefinicje i funkcje służące do dodawania i odbierania elementów z kolejek przewidziane są do używania wewnątrz funkcji stanowiących zadania. W praktyce jednak często zachodzi potrzeba wysyłania bądź odbierania danych wewnątrz funkcji obsługi przerwań (ISR). Odmiany makrodefinicji odpowiednio dodającej element do kolejki i odbierającej element z kolejki w bezpieczny sposób wewnątrz funkcji ISR mają nazwy ***xQueueSendFromISR*** i ***xQueueReceiveFromISR***. Ze specyfiki funkcji ISR wynika, że np. nie mogą one wstrzymywać działania systemu operacyjnego np. wprowadzając oczekiwanie na przyjście elementu. Oznacza to także nieco inny sposób ich wywoływania.

Omówione w tym punkcie funkcje i makrodefinicje nie wyczerpują listy wszystkich dostępnych narzędzi przewidzianych do posługiwania się kolejkami w systemie FreeRTOS (ich pełną listę i dokumentację można znaleźć na oficjalnej stronie projektu).

3.3 Semafory

Uwaga: pełna dokumentacja API tworzenia i kontroli semaforów znajduje się na stronie www.freertos.org w dziale API Reference → Semaphore / Mutexes.

Semafory w systemie operacyjnym FreeRTOS utworzone są przy pomocy makrodefinicji „opakowujących” istniejący system kolejek.

Tworzenie semafora odbywa się przez wywołanie makra ***xSemaphoreCreateBinary***.

```
SemaphoreHandle_t xSemaphoreCreateBinary( void );
```

Jedynym przekazywanym do niego argumentem jest obiekt typu *xSemaphoreHandle* reprezentujący semafor.

Ustawienie semafora w stan „zajęty” można przeprowadzić używając makra ***xSemaphoreTake***.

```
xSemaphoreTake( SemaphoreHandle_t xSemaphore,
                TickType_t xTicksToWait );
```

Tym razem podajemy dwa argumenty:

- ***xSemaphore*** to wartość ukryta w obiekcie reprezentującym semafor, który chcemy ustawić,
- ***xTicksToWait*** to czas oczekiwania liczony w tyknięciach systemu operacyjnego (jest to maksymalny czas, jaki przewidujemy na oczekiwanie na zwolnienie semafora).

Jeśli w ciągu wybranego czasu uzyskamy semafor (zwolni się dostęp do zasobu), to makro zwróci *pdTRUE*.

Mając dostęp do zasobu, możemy wykonać na nim operacje. Po ich zakończeniu możemy zwolnić semafor, by poinformować inne zadania, że już nie używamy tego zasobu. Semafor zwalniamy wywołaniem kodu z makra ***xSemaphoreGive***

```
xSemaphoreGive( xSemaphoreHandle xSemaphore );
```

do którego przekazujemy jedynie wartość obiektu reprezentującego zwalniany semafor. To makro ma także swoją odmianę, której można używać w funkcjach obsługi przerwań – nosi ono nazwę ***xSemaphoreGiveFromISR***. Wymaga ono dodatkowego parametru przekazywanego przez wskaźnik i mówiącego, czy po zakończeniu funkcji obsługi przerwania należy wykonać przełączenie zadań. Szczegóły można znaleźć w oficjalnej dokumentacji.

Uwaga: jeśli utworzony przez **`xSemaphoreCreateBinary`** semafor zaraz po utworzeniu jest „zajęty”, to wtedy należy zaraz po utworzeniu zwolnić ten semafor wywołując dla niego **`xSemaphoreGive`**.

4 Zadania do wykonania

Fragmenty zaznaczone w ten sposób stanowią wytyczne, co należy zrobić jako efekt realizacji ćwiczenia.

4.1 Tworzenie i kontrola nad zadaniami

1. W module main.c proszę napisać dwie funkcje mogące działać jako zadania systemu operacyjnego; funkcje powinny mieć nazwy **`messageTask`** i **`ledTask`** .
2. Proszę utworzyć jako zmienną globalną **`messageTaskHandler`** będącą uchwytem (*handler*) do zadania **`messageTask`**.
3. W zadaniu **`StartDefaultTask`** należy utworzyć zadania, których kod będą stanowiły funkcje **`messageTask`** i **`ledTask`**. Zadanie **`messageTask`** powinno mieć priorytet równy 1, a **`ledTask`** 3. Tworzenie zadań można przeprowadzić na początku sekcji USER CODE 5 oznaczonej dodatkowo komentarzem:

```
/* USER CODE BEGIN 5 */

/*
 * ===============================================
 * This is a nice place to create additional tasks,
 * queues, and semaphores for this lab exercise :)
```

```
 * ============================================
 */
```

4. W zadaniu **`ledTask`** proszę zaimplementować w nieskończonej pętli włączanie i wyłączanie diody LD2 podłączonej do wyprowadzenia PB7 (kontroler GPIOB, wyprowadzenie 7). Tutaj, dla wygody, można użyć makrodefinicji reprezentujący port GPIO oraz wyprowadzenie m.in. diody LD2 – można zadziałać analogicznie, jak obsługiwana jest dioda LD1 w zadaniu *StartDefaultTask*. Czas włączenia LD2 ma trwać 100 ms, a wyłączenie 400 ms. Uzyskamy przez to dwa błyśnięcia diody LED w każdej sekundzie, co będzie sygnalizowało obiegi pętli głównej zadania *ledTask*.
5. W zadaniu **`messageTask`** umieszczamy kod wypisujący na terminalu komunikat z aktualnym czasem systemowym oraz pętlę symulującą wykonywanie obliczeń. Dioda LD3 sygnalizuje wykonywanie "obliczeń" i w takiej postaci zadania jak poniżej, świeci prawie cały czas.

```
void messageTask(void* p){
	srand(xTaskGetTickCount());
	while(1){
		xprintf("Current systime is: %05d\n",(int)xTaskGetTickCount());
		HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_SET);

		/*Udajemy poważne obliczenia,
		  tutaj losowanie i instrukcja no-operation
		  powtórzona trudno powiedzieć, ile razy;)
		*/
		const int MASK = 0x007FFFFF;
		volatile int i = rand() & MASK;
		while(i--) asm("nop");

		HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_RESET);

	}
}
```

6. Następnie, nie modyfikując części udającej "poważne obliczenia", w pętli głównej zadania **`messageTask`** należy zaimplementować takie opóźnienie, aby zmiana stanu diody LED oraz wypisywanie komunikatu następowały dokładnie co 1000 tyknięć systemu operacyjnego.

Proszę umieścić na UPEL zrzut ekranu fragmentu terminala, pokazujący, że komunikat wyświetla się dokładnie co 1000 tyknięć.

4.2 Zastosowanie kolejek

W tej części należy utworzyć pętlę *loopback* działającą w następujący sposób:

- W zadaniu **`ledTask`** z portu szeregowego odczytujemy znak odebrany z terminala (np. przy pomocy funkcji **`inkey`** z **`main.c`**). **Wskazówka**: Funkcja **`inkey`** działa w ten sposób, że zwraca kod ASCII znaku ostatnio odebranego z portu szeregowego. Jeśli od ostatniego wywołania nie pojawił się żaden nowy znak, to funkcja zwróci 0. Dodatkowo warto zmniejszyć opóźnienia w **`ledTask`** do łącznej wartości nie przekraczającej 10 tyknięć.
- Dalej w zadaniu **`ledTask`** znak powinien być wysłany do kolejki o nazwie **`loopQueue`**

(przyjmijmy, że początkowo kolejka loopQueue powinna mieć pojemność 5 elementów). Jeśli kolejka będzie przepełniona i nie da się dodać do niej nowego elementu, generujemy komunikat informujący o tym.

- W zadaniu **`messageTask`** sprawdzamy, czy w kolejce *loopQueue* nie pojawił się nowy element.
- Jeśli pojawił się nowy element, to należy go odesłać z powrotem na terminal z odpowiednim komunikatem, np. „**`odebrano znak: <znak>`**”.

Po wstępnym sprawdzeniu działania, można wykonać praktyczny eksperyment. Co się stanie, gdy:

1. wyślemy z terminala kilka znaków w krótkim czasie – czy znaki „przepadną” czy może zostaną tymczasowo przechowane w kolejce?
2. zmienimy priorytet zadania **`messageTask`** na 3 a **`ledTask`** na 2?

4.3 Zawieszanie i wznawianie wykonywania zadań

Na kolejnym etapie wykonywania ćwiczenia należy dodać mozliwość zawieszania (*suspend*) i wznawiania (*resume*) zadania **`messageTask`** w reakcji na nadejście konkretnych znaków z programu terminalowego. Jeśli odebrany w zadaniu **`ledTask`** znak będzie miał kod litery **`'s'`**, należy zawiesić wykonywanie zadania **`messageTask`**. Jeśli z kolei przyjdzie znak **`'r'`**, należy wznowić wykonywanie zadania.

Pewnym ułatwieniem w obserwacji działania programu może być dobranie innych (nawet mniej precyzyjnych) funkcji realizujących opóźnienie w zadaniu **`messageTask`**.

Proszę umieścić na UPEL plik *main.c* z nazwą zmienioną na ***main_43.c*** i treścią w stanie po zakończeniu realizacji tego punktu.

4.4 Zastosowanie semaforów

Aby nie doszło do konfliktu w dostępie do pewnego zasobu stosuje się m.in. semafory. W ostatniej części ćwiczenia należy utworzyć dwa zadania o nazwach **`semTaskA`** i **`semTaskB`**, które będą symulowały blokowanie pewnego zasobu – w rzeczywistości będą realizowały jedynie opóźnienie *vTaskDelay* sygnalizując zajętość tego „sztucznego” zasobu świeceniem diody LED niebieskiej lub czerwonej (LD2 lub LD3). Najlepiej, gdy jedno z utworzonych zadań będzie miało priorytet o 1 wyższy niż drugie.

1. Tworzymy zadania **`semTaskA`** i **`semTaskB`**.
2. W każdym z tych zadań umieszczamy kod
   - włączający „swoją” diodę LED, np. LD2 w *semTaskA*, i LD3 w *semTaskB*,
   - czekający chwilę,
   - wyłączający „swoją” diodę LED,
   - czekający ponownie przy wyłączonej „swojej” diodzie LED.

Uwaga: warto zadać zróżnicowane czasy opóźnień przy włączonych i wyłączonych diodach LED, a

dodatkowo efekt działania/nie działania semafora będzie lepiej widoczny, gdy szybkości błyskania obu diod LED będą wyraźnie różne. Przykładowe czasy pozwalające na względnie łatwą obserwację, to: 100 ms i 200 ms dla jednego zadania oraz 4 s i 1 s dla drugiego.

3. Tworzymy semafor o nazwie **`labSemaphore`** i inicjalizujemy go.
4. Kod odpowiedzialny za cykl włącz-czekaj-wyłącz dla diod LD2 i LD3 w zadaniach „zabezpieczamy" semaforem **`labSemaphore`** wg opisu z punktu 3.3 i przykładów z oficjalnej dokumentacji. Następnie porównujemy działanie programu sprzed i po zabezpieczeniu semaforem. W ramach relacji z wykonania zadania komentujemy różnicę, a komentarz umieszczamy w materiałach sprawozdania. Możemy także zamienić priorytety zadań i zaobserwować efekt.

Proszę umieścić na UPEL plik *main.c* z nazwą zmienioną na ***main_44.c*** i treścią w stanie po zakończeniu realizacji tego punktu.

4.5 Archiwizacja katalogu projektu

Katalog ze zmodyfikowanym projektem należy zarchiwizować na własnym nośniku lub dostępnym serwerze, ponieważ będzie on przydatny przy realizacji następnego ćwiczenia laboratoryjnego. Dla zmniejszenia rozmiaru archiwum należy usunąć pliki wynikowe (***clean project***). Nie jest zalecane pozostawianie projektu na komputerze w pracowni, ponieważ dyski tych komputerów mogą zostać nadpisane pomiędzy zajęciami bez uprzedniego ostrzeżenia.

5 Zagadnienia

Wymienione w niniejszej sekcji zagadnienia proszę opracować samodzielnie, a następnie dla każdego punktu, odnieść się zwięźle w sprawozdaniu i/lub utrwalić zdobytą wiedzę i umiejętności zależnie od ogłoszonej przez osobę prowadzącą formy weryfikacji efektów uczenia dla tego zadania.

1. Funkcjonalność systemu operacyjnego FreeRTOS – oferowana (potencjalna) oraz wykorzystywana w ćwiczeniu, np.
   a. czy korzystamy oraz czy możemy korzystać z ochrony pamięci oferowanej przez jednostkę MPU?
   b. które mechanizmy są dostępne w ćwiczeniu oraz ogólnie, np. semafory, kolejki, mutexy?
2. Sprawowanie kontroli nad zadaniami:
   a. tworzenie zadań,
   b. czy można wyjść z funkcji stanowiącej zadanie?
   c. zakańczanie działania zadań (*vTaskDelete*),
   d. zawieszanie działania zadań (*vTaskSuspend*, *vTaskResume*),

e. zastosowanie obiektów reprezentujących zadanie („handlerów”),

f. funkcje realizujące opóźnienia: *vTaskDelay*, *vTaskDelayUntil* – czym się różnią, w jaki sposób używa się ich i jakich efektów możemy się spodziewać?

g. zadanie bezczynności (*idle task*): jaki ma priorytet, czy zawsze jest wykonywane?

3. Kolejki, semafory

   a. rozumienie mechanizmów działania,

   b. zastosowanie funkcji interfejsu API,

4. Rozumienie sensu stosowania systemu operacyjnego czasu rzeczywistego w urządzeniu mikroprocesorowym

5. Dla typowego lub wybranego, przykładowego kontrolera GPIO:

   a. Umiejętność narysowania mającego sens schematu blokowego kontrolera GPIO o możliwościach funkcjonalnych wg zadanych wytycznych, alternatywnie: umiejętność wskazania/wybrania, do czego są potrzebne poszczególne elementy na zadanym schemacie blokowym GPIO

   b. Znajomość funkcjonalności kontrolerów GPIO w nowoczesnych mikrokontrolerach

   c. Umiejętność wskazania zastosowań kontrolera GPIO, w tym możliwości implementacji różnych interfejsów przy jego pomocy

   d. ogólna/podstawowa wiedza na temat prostych interfejsów powszechnie używanych w systemach mikroprocesorowych:

      - SPI (Serial Peripheral Interface),
      - I2C (Inter-Integrated Circuit),
      - 1-Wire

   e. Świadomość ograniczeń w działaniu GPIO: szybkość, możliwości konfiguracyjne, reakcja na zdarzenia