Wydział Informatyki AGH

Ćwiczenie laboratoryjne

Zastosowanie systemu operacyjnego czasu rzeczywistego – na przykładzie FreeRTOS

Nazwa kodowa: **`rtos`**. Wersja **`20251106`**

Ada Brzoza-Zajęcka
abrzoza@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. W ćwiczeniu użyjemy następujących narzędzi:

- zintegrowanego środowiska **STM32CubeIDE w wersji 1.19.0**,
- programu terminalowego, np. **puTTY**,
- uniwersalnego modułu pomiarowego **Analog Discovery 3** produkcji Digilent wraz z aplikacją **WaveForms**.

1.4 Materiały pomocnicze

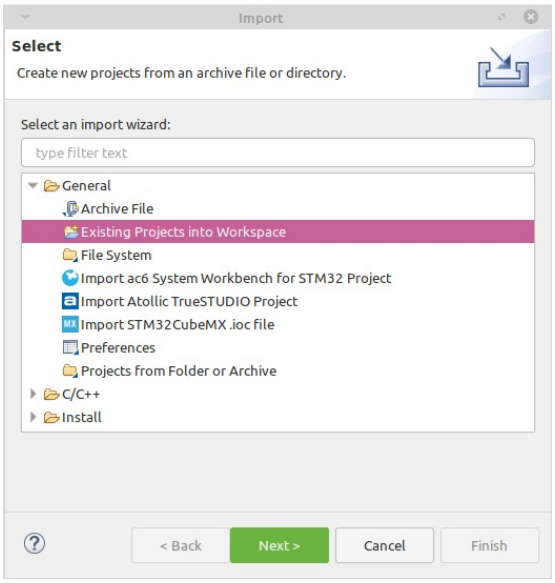
Do wykonania ćwiczenia niezbędne lub przydatne są następujące materiały:

- 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,

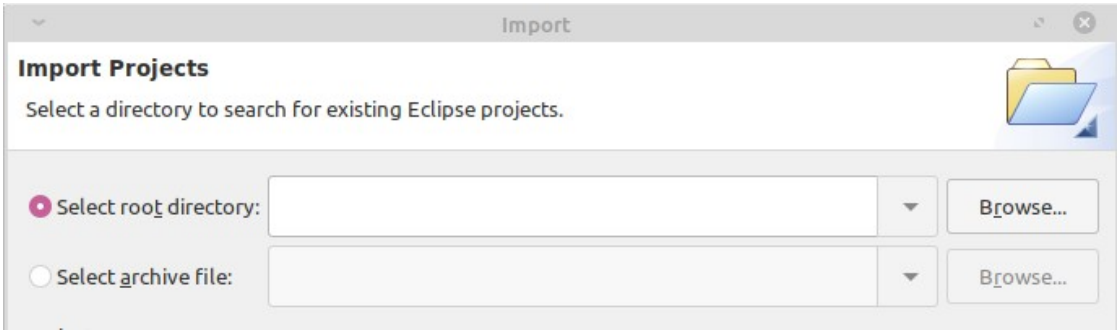
- instrukcja obsługi (user manual) zestawu STM32F429 Nucleo-144,
- zestaw przykładów dla STM32F429 Nucleo-144.

2 Import i otwarcie projektu

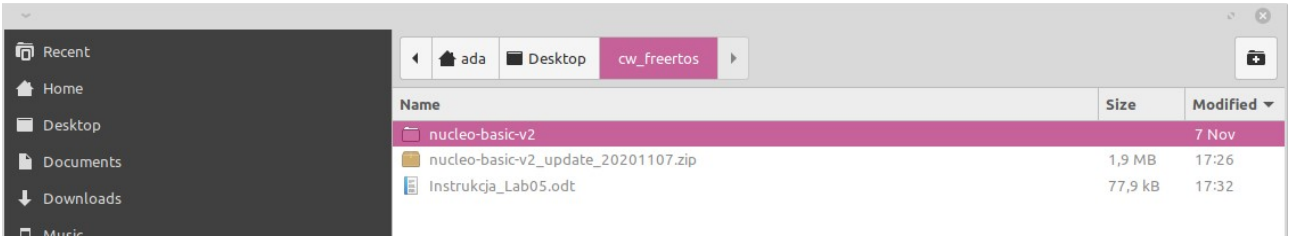
Pobrane projekt startowy nucleo-basic-v2 należy rozpakować i importować w środowisku STM32CubeIDE przy pomocy polecenia ***File → Import…*** a następnie rozwijamy sekcję ***General → Existing Projects into Workspace → Next***.



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

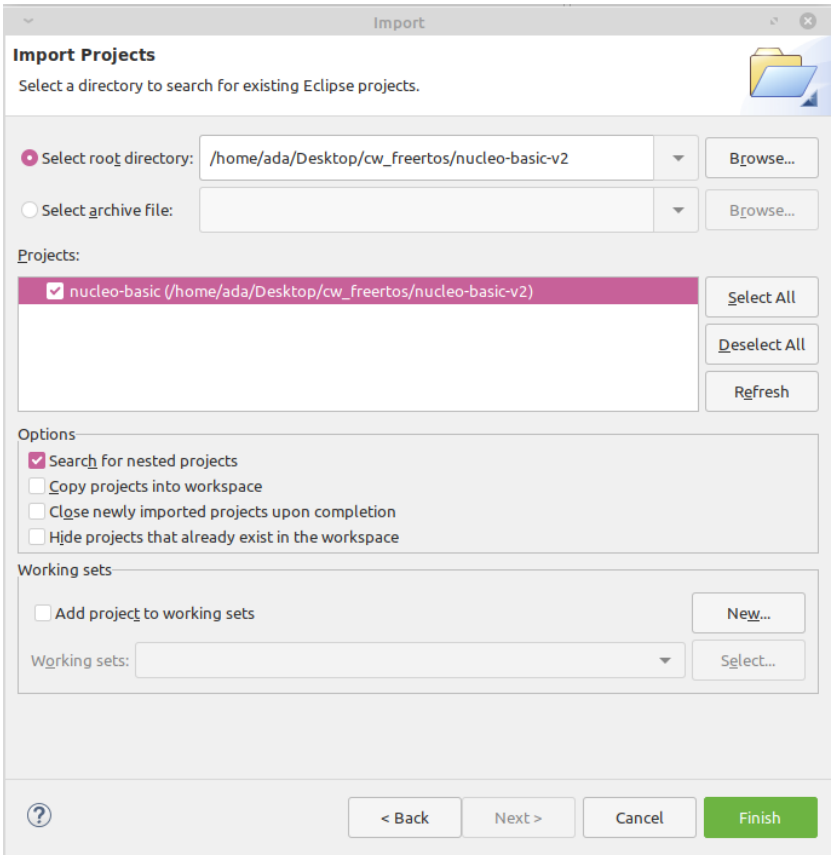


Wskazujemy katalog z pobranym projektem:



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 czerwona dioda LED oznaczona jako LD3 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***:

```
BaseType_t xQueueSend(
                      QueueHandle_t xQueue,
                      const void * pvItemToQueue,
                      TickType_t 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 uzyskiwanym efektem działania jest obiekt typu *xSemaphoreHandle* reprezentujący semafor. Jeśli obiekt ten ma wartość inną niż NULL, to znaczy, że prawdopodobnie został poprawnie utworzony.

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( SemaphoreHandle_t 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.

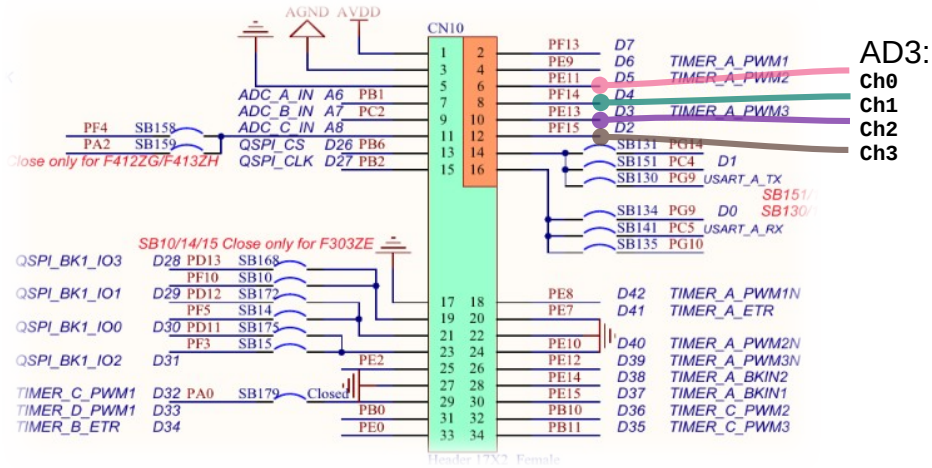
Uwaga: jeśli utworzony przez **xSemaphoreCreateBinary** semafor zaraz po utworzeniu jest „zajęty”, to wtedy należy zaraz po utworzeniu zwolnić ten semafor wywołując dla niego **xSemaphoreGive**.

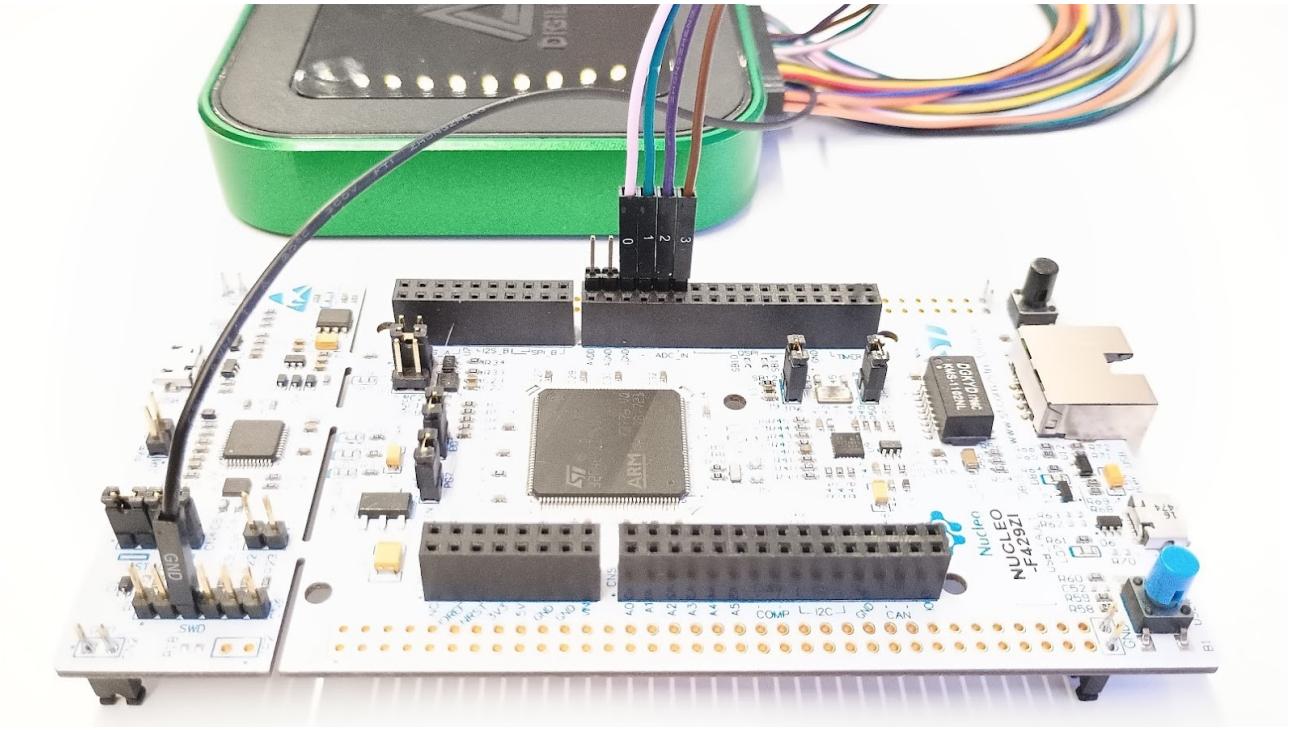
4 Przygotowanie analizatora stanów logicznych

Jako analizatora stanów logicznych użyjemy Analog Discovery 3 (AD3) produkcji Digilent.

4.1 Część sprzętowa analizatora

Sondy analizatora powinny zostać podłączone do płytki w sposób przedstawiony na poniższym schemacie i zdjęciu.





Wyprowadzenie GND powinno być podłączone do dowolnego wyprowadzenia masy płytki. Powyżej przedstawiono jedynie przykład.

Uwaga: po zakończeniu pracy nad ćwiczeniem proszę nie odłączać przewodów AD3. W razie jakichkolwiek wątpliwości z podłączeniem przewodów należy skonsultować się z osobą prowadzącą laboratorium.

Analizator będzie rejestrował napięcia (tutaj stany logiczne) na wyprowadzeniach GPIO mikrokontrolera. Wyprowadzenia będą ustawiane w programie w taki sposób, aby obrazować działanie wybranych fragmentów kodu. W tym ćwiczeniu wyprowadzenia mikrokontrolera będą aktywowane do stanu wysokiego, gdy system operacyjny przełączy się do wykonywania wybranego zadania i dezaktywowane, gdy zakończy wykonywanie zadania.

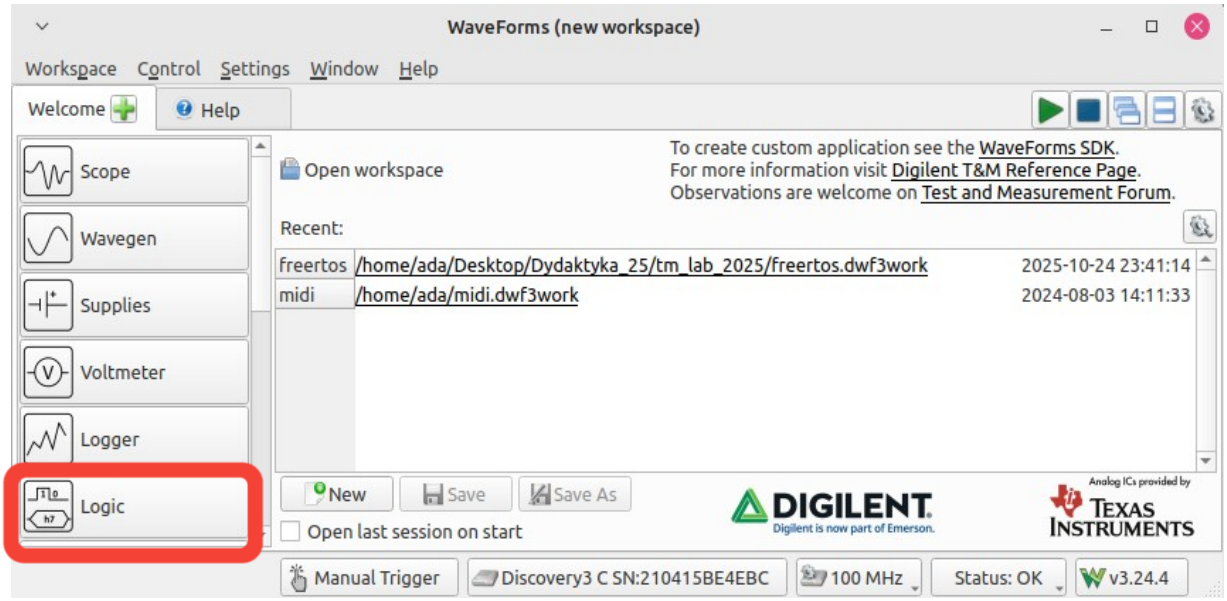
4.2 Oprogramowanie analizatora

Oprogramowanie analizatora nosi nazwę Digilent **WaveForms**. Powinno ono być zainstalowane na

komputerach w pracowni pod systemami **Linux**. Jest ono dość intuicyjne, a jego obsługa jest dość interaktywna. Najłatwiej będzie, jeśli przy uruchomieniu moduł AD3 będzie już podłączony do komputera przez port USB. Jeśli nie - pojawi się komunikat:

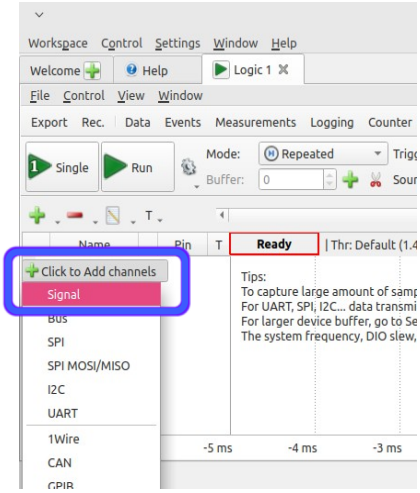
W takim przypadku należy podłączyć AD3 do komputera i wcisnąć ***Retry***.

Zobaczymy główne okno WaveForms, w którym wybieramy interfejs ***Logic***.

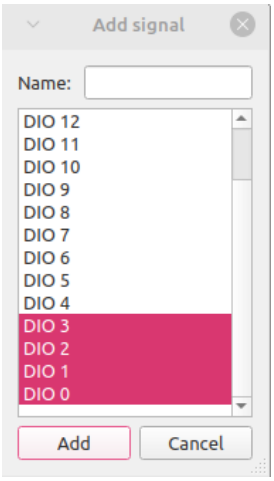


4.2.1 Dodawanie kanałów pomiarowych

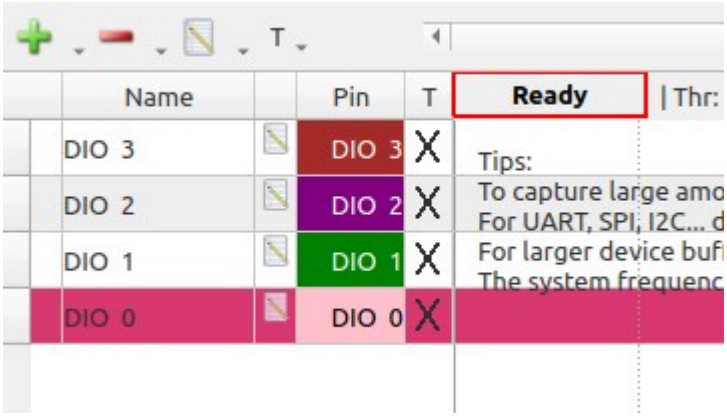
Logic zostanie otwarty w nowej zakładce aplikacji. Następnie dodajemy kanały pomiarowe przez ***Click to Add Channels***. Następnie wybieramy Signal.



W dialogu *Add signal* wybieramy kanały ***DIO 0, 1, 2 i 3***. Zatwierdzamy przyciskiem ***Add***.



Klikając dwa razy na poszczególne kanały możemy zmienić ich nazwy, co nie jest konieczne, lecz sugerowane dla przejrzystości na dalszych etapach ćwiczenia i przy sprawozdaniu.



Zalecana jest następująca zmiana nazw kanałów:

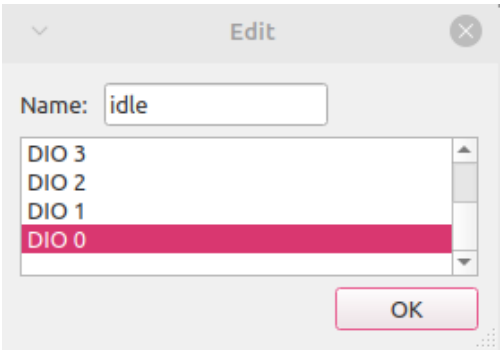
Kanał DIO 0 → ***idle*** (lub coś innego, sugerującego zadanie bezczynności),

Kanał DIO 1 → ***message*** (dla zadania wysyłania wiadomości),

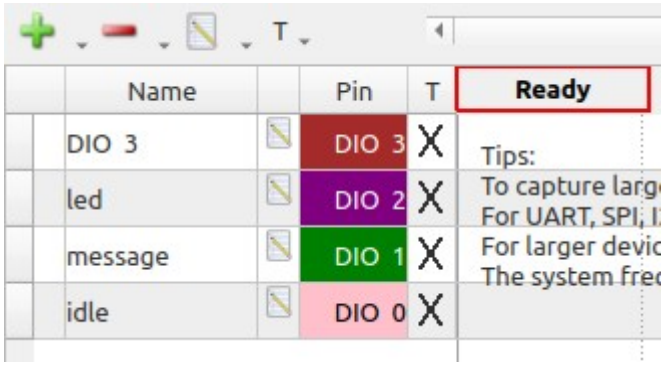
Kanał DIO 2 → ***led*** (dla zadania błyskającego diodą LED),

Kanał DIO 3 → możemy zostawić do własnych eksperymentów.

Np.

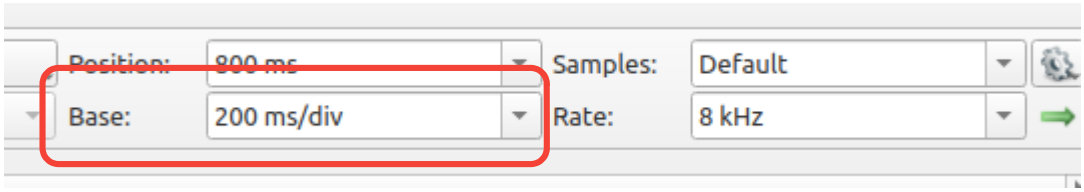


Na początek konfiguracja kanałów może wyglądać np. tak:



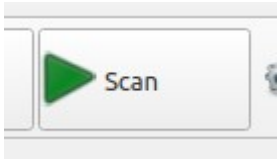
4.2.2 Konfiguracja podstawy czasu

Podstawę czasu (w polu ***Base***) ustawiamy na 200 ms. Jest to tylko przykładowa wartość - jak najbardziej możemy eksperymentować z różnymi podstawami czasu. Podstawę czasu możemy zmieniać przez **Ctrl + Rolka myszy**.

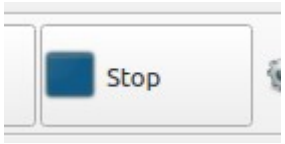


4.2.3 Rozpoczynanie i zatrzymywanie pomiarów

Przyciskiem ***Scan*** lub ***Start*** (zależnie od ustawionej podstawy czasu) rozpoczynamy rejestrację.

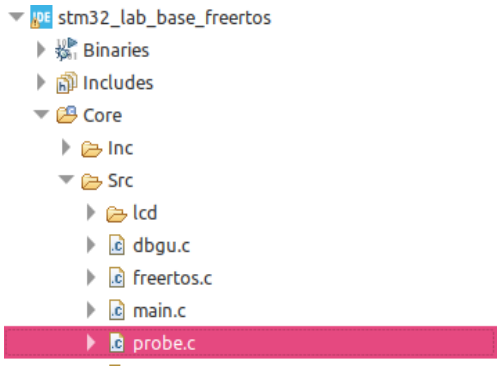


Rejestrację możemy zatrzymać przyciskiem Stop, który pojawi się w miejscu ***Scan*** lub ***Start***.



4.3 Część programowa - firmware STM32

Od strony programowej za generowanie sygnałów dla analizatora odpowiada ją funkcje z modułu **`Core/Src/probe.c`** oraz **`Core/Inc/probe.h`**:



Główną funkcją jest tutaj **`probe_output_set`**:

```
//state can be 0 for L or nonzero for H
//idx can be from 0 to 3
void probe_output_set(int idx, uint8_t state){
  switch(idx){
    case 0:
      if(state) PROBE_CH0_H; else PROBE_CH0_L;
      break;
    case 1:
      if(state) PROBE_CH1_H; else PROBE_CH1_L;
      break;
    case 2:
      if(state) PROBE_CH2_H; else PROBE_CH2_L;
      break;
    case 3:
      if(state) PROBE_CH3_H; else PROBE_CH3_L;
      break;
  }
}
```

Z kolei makrodefinicje **`PROBE_CHx_L/H`** znajdują się w **`main.h`**. Np. dla kanału 0:

```
#define PROBE_CH0_PIN   GPIO_PIN_11
#define PROBE_CH0_GPIO  GPIOE

#define PROBE_CH0_H     HAL_GPIO_WritePin(PROBE_CH0_GPIO, PROBE_CH0_PIN, GPIO_PIN_SET)
#define PROBE_CH0_L     HAL_GPIO_WritePin(PROBE_CH0_GPIO, PROBE_CH0_PIN, GPIO_PIN_RESET)
```

W ćwiczeniu aktywacja sygnałów przy pomocy funkcji **`probe_output_set`** odbywa się w pliku `Core/Inc/`**`freertos_custom_config.h`**. Zostały tam zdefiniowane makra **`traceTASK_SWITCHED_IN`** oraz **`traceTASK_SWITCHED_OUT`**.

```
#define traceTASK_SWITCHED_IN() { \
  uint32_t id;  \
  id = pxCurrentTCB->uxTaskNumber; \
  probe_output_set(id,1); \
}

#define traceTASK_SWITCHED_OUT() { \
  uint32_t id;  \
  id = pxCurrentTCB->uxTaskNumber; \
  probe_output_set(id,0); \
}
```

Są one wywoływane automatycznie przez system operacyjny FreeRTOS przy przełączaniu zadań: w momencie uruchomienia (kontynuacji) wykonywania zadania oraz wstrzymania wykonywania zadania. Informacja, które zadanie zostało uruchomione lub wstrzymane, może być odczytana z

danych **pxCurrentTCB->uxTaskNumber**. Zależnie od odczytanego stamtąd numeru zadania tworzony jest parametr id mówiący o tym, dla której sondy należy podać aktywny stan logiczny.

Przypisanie numerów zadań do **uxTaskNumber** odbywa się przez wywołanie funkcji **vTaskSetTaskNumber** z API systemu FreeRTOS. Podajemy w niej handler do zadania oraz przypisany numer. W programie testowym na laboratorium mogą to być przykładowo następujące wywołania:

```
  vTaskSetTaskNumber(xTaskGetIdleTaskHandle(), 0);
  vTaskSetTaskNumber(messageTaskHandle,1);
  vTaskSetTaskNumber(ledTaskHandle,2);
```

Tym sposobem:

- zadanie bezczynności będzie miało numer 0,
- zadanie przesyłające wiadomości będzie miało numer 1,
- a zadanie błyskające diodą LED będzie miało przypisany numer 2.

5 Zadania do wykonania

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

5.1 Tworzenie i kontrola nad zadaniami

W module **main.c** napiszemy dwie funkcje mogące działać jako zadania systemu operacyjnego. Funkcje będą miały nazwy **messageTask** i **ledTask** . W ramach przygotowania proszę utworzyć zmienne globalne: **messageTaskHandler** oraz **ledTaskHandler** będące uchwytami (*handler*) do tworzonych zadań. Można je utworzyć np. w sekcji ***USER CODE BEGIN PV***.

5.1.1 Zadanie messageTask

1. Tworzymy funkcję, która będzie stanowiła zadanie **messageTask** . Umieszczamy w niej kod wypisujący na terminalu komunikat z aktualnym czasem systemowym oraz opóźnienie **HAL_Delay**, które będzie udawało skomplikowane "obliczenia", ponieważ realizacja tego opóźnienia pochłania całą moc obliczeniową procesora. Z kolei użyta później funkcja **vTaskDelay** (wbudowana w system FreeRTOS) praktycznie całkowicie odciąża procesor od obliczeń. Dioda LD1 sygnalizuje wykonywanie "obliczeń".

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

    // Udajemy powazne obliczenia (ta czesc zajmuje czas procesora)
    LD1_ON;
    uint32_t t;
    HAL_RNG_GenerateRandomNumber(&hrng, &t); //generujemy liczbe losowa
    t = (t & 0x000001FF);                    //ograniczamy zakres maskujac jej bity
    HAL_Delay(t);                            //czekamy w funkcji zajmujacej czas procesora
    LD1_OFF;

    //a teraz zadanie przestanie zajmowac czas procesora
    vTaskDelay(100);
  }
}
```

2. W zadaniu **StartDefaultTask** należy wywołać funkcje **xTaskCreate** tworzącą zadanie **messageTask**. Zadanie **messageTask** powinno mieć priorytet równy **osPriorityNormal+1** oraz maksymalny rozmiar stosu wynoszący **configMINIMAL_STACK_SIZE+16**. Tworzenie

zadań można przeprowadzić na początku sekcji *USER CODE 5* , po wywołaniu funkcji **`probe_output_init`**.

Ważna uwaga: przy tworzeniu zadania należy w odpowiednim argumencie **`xTaskCreate`** podać wcześniej utworzony uchwyt tego zadania (handler **`messageTaskHandler`**). Handler należy koniecznie podać przez wskaźnik (referencję) czyli ze znakiem &.

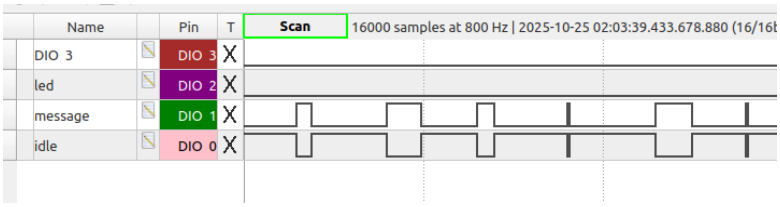
```
/* USER CODE BEGIN 5 */
probe_output_init();
//tutaj utworzymy zadanie przy pomocy xTaskCreate
```

3. Nieco dalej, pod wywołaniem funkcji przypisującej identyfikator 0 do zadania **`idleTask`**:

   `vTaskSetTaskNumber(xTaskGetIdleTaskHandle(), 0);`
   dodajemy analogiczne wywołanie **`vTaskSetTaskNumber`**. Jednak tym razem dla zadania **`messageTask`**. Korzystamy z utworzonego wcześniej uchwytu tego zadania (**`messageTaskHandler`**) i przypisujemy do tego zadania identyfikator **`1`**.

4. Kompilujemy kod, obserwujemy miganie zielonej diody LED: LD1. Włączamy rejestrację sygnałów w WaveForms. W ramach sprawozdania należy potrafić wskazać i wyjaśnić osobie prowadzącej na podstawie obserwowanych przebiegów czasowych w WaveForms: w których chwilach mikroprocesor wykonuje zadanie bezczynności, a w których **`messageTask`**. Screeshot z zarejestrowanymi przebiegami czasowymi umieszczamy na UPEL pod nazwą ***message_task_waveforms***.

   Zarejestrowane przebiegi powinny nieco przypominać przedstawione poniżej (tu przy podstawie czasu 2 s/div):



   Troubleshooting: co jeśli nie widzimy przebiegów? Przede wszystkim warto sprawdzić, czy poprawnie przekazaliśmy uchwyt zadania messageTask podczas jego tworzenia funkcją xTaskCreate. Ostrzeżenie kompilatora może sugerować nieprawidłowe przekazanie tego uchwytu.

5. Następnie, nie modyfikując części udającej "poważne obliczenia", w pętli głównej zadania **`messageTask`** należy zaimplementować takie opóźnienie, aby włączenie diody LED oraz wypisywanie komunikatu następowały dokładnie co 1 sekundę czyli co 1000 tyknięć systemu operacyjnego, np.

```
Current systime is: 01010
Current systime is: 02010
Current systime is: 03010
Current systime is: 04010
Current systime is: 05010
Current systime is: 06010
Current systime is: 07010
```

6. Proszę umieścić na UPEL zrzut ekranu fragmentu terminala, pokazujący, że komunikat wyświetla się dokładnie co 1000 tyknięć oraz zrzut ekranu z przebiegami zarejestrowanymi w aplikacji WaveForms. Zalecane nazwy plików to: ***message_task_1k_term*** oraz

message_task_1k_wave. Prezentujemy wynik również osobie prowadzącej.

5.1.2 Zadanie ledTask

5.1.2.1 Przygotowanie kodu zadania

Na samym początku kodu zadania proszę umieścić wywołanie funkcji **`vTaskDelay(100)`**. Dzięki temu główna pętla zadania rozpocznie się nieco później. Będzie to miało znaczenie na późniejszym etapie ćwiczenia.

Dalej w zadaniu **`ledTask`** proszę zaimplementować w nieskończonej pętli włączanie i wyłączanie diody LD2 podłączonej do wyprowadzenia PB7 (kontroler GPIOB, wyprowadzenie 7). W czasie, gdy dioda LED jest włączona, zadanie powinno zajmować czas procesora (opóźnienie realizowane przez **`HAL_Delay`**), natomiast przy wyłączonej diodzie LED zadanie nie powinno zajmować czasu procesora (zastosowanie **`vTaskDelay`**). W zadaniu **`ledTask`**, dla wygody, można użyć makrodefinicji dla diody LD2 z pliku **`main.h`** działając analogicznie, jak obsługiwana jest dioda LD1 w zadaniu **`messageTask`**. Czas włączenia LD2 powinien trwać 20 ms, a wyłączenia 60 ms. Uzyskamy przez to częste i krótkie błyśnięcia diody LED, co będzie sygnalizowało obiegi pętli głównej zadania **`ledTask`**, a jednocześnie będzie cyklicznie zajmowało i zwalniało czas procesora.

5.1.2.2 Uruchomienie zadania

Utworzenie (**`xTaskCreate`**), przypisanie roboczego numeru zadania (**`vTaskSetTaskNumber`**) oraz przypisanie uchwytu zdania **`ledTask`** powinno odbywać się w sposób analogiczny jak dla zadania **`messageTask`**. Inny będzie oczywiście uchwyt zadania - tym razem nazywamy go **`ledTaskHandle`**. Inne będą także następujące parametry zadania:

- wstępnie ustawiamy priorytet zadania **`ledTask`** na **`osPriorityNormal+2`**,
- zadaniu nadajemy nazwę **`"led"`**,
- w wywołaniu funkcji **`vTaskSetTaskNumber`** przypisujemy mu roboczy numer **2**.

5.1.2.3 Opcja: usunięcie zadania StartDefaultTask i "checkpoint" kodu

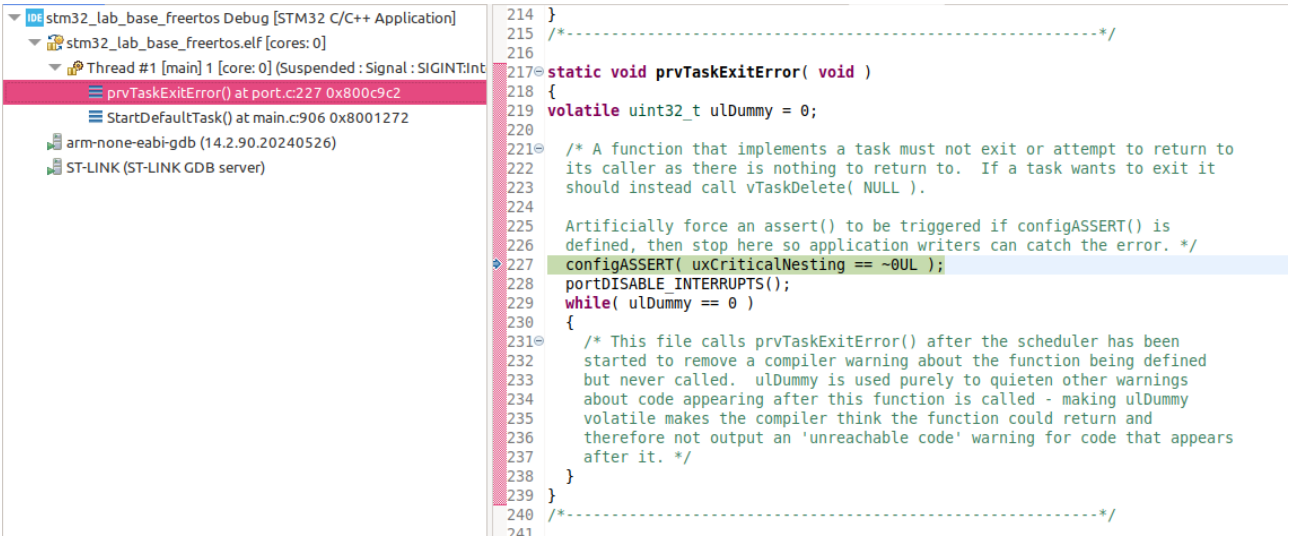
Z racji, że w tym momencie pętla główna zadania **`StartDefaultTask`** nie jest nam potrzebna, więc po wykonaniu potrzebnych inicjalizacji, możemy bez obaw usunąć wykonywanie zadania **`StartDefaultTask`** z kolejki zadań. Możemy to zrobić wywołując w jego kodzie **`vTaskDelete`** z parametrem **`NULL`** - będzie to oznaczało, że zadanie usunie samo siebie. Jeśli chcielibyśmy usunąć to zadanie z poziomu innego zadania, to zamiast **`NULL`** należałoby przekazać w argumencie uchwyt (handler) do zadania, które chcemy usunąć.

Po zastosowaniu **`vTaskDelete`**, na tym etapie **`StartDefaultTask`** mogłoby wyglądać np. tak jak na poniższym zrzucie ekranu (przy okazji możemy porównać kod z dotychczas napisanym). Po uruchomieniu pozostałych zadań i po nadaniu interesującym zadaniom identyfikatorów, zadanie **`StartDefaultTask`** przestanie zajmować zasoby systemu mikroprocesorowego.

```c
/* USER CODE END Header_StartDefaultTask */
void StartDefaultTask(void *argument)
{
  /* init code for USB_DEVICE */
  MX_USB_DEVICE_Init();
  /* USER CODE BEGIN 5 */
  probe_output_init();

  xTaskCreate(messageTask, "msg", configMINIMAL_STACK_SIZE+16, NULL, osPriorityNormal+1, &messageTaskHandle);
  xTaskCreate(ledTask, "led", configMINIMAL_STACK_SIZE+16, NULL, osPriorityNormal+2, &ledTaskHandle);

  vTaskSetTaskNumber(xTaskGetIdleTaskHandle(), 0);
  vTaskSetTaskNumber(messageTaskHandle,1);
  vTaskSetTaskNumber(ledTaskHandle,2);

  vTaskDelete(NULL);

  /* Infinite loop */
  for(;;)
  {
    LD3_TOGGLE;
    vTaskDelay(50);
  }
  /* USER CODE END 5 */
}
```

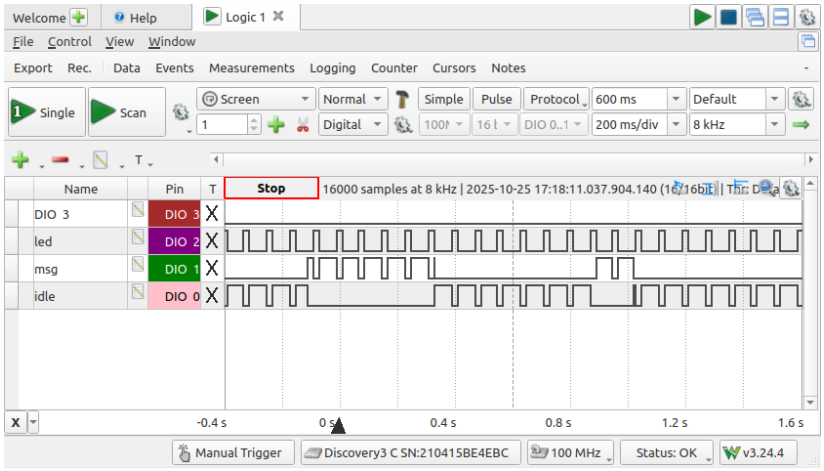
Uwaga: błędem byłoby wyjście z funkcji stanowiącej zadanie - próba wykonania takiego kodu skończyłaby się sytuacją, z której typowo jedynym "ratunkiem" przy aktualnym projekcie byłby sprzętowy reset. Gdybyśmy zatem zamiast **`vTaskDelete`** napisali w kodzie zadania **`return`**, to program zatrzymałby się na obsłudze sytuacji wyjątkowej **`prvTaskExitError`** w systemie FreeRTOS. Nieobowiązkowo, jeśli szacujemy, że wystarczy nam czasu na resztę ćwiczenia, można spróbować zrobić taki eksperyment w sesji debugowania a po "zawieszeniu się" programu wcisnąć przycisk pauzy

i sprawdzić, gdzie w rzeczywistości utknął:



5.1.3 Obserwacje

Przy projekcie zestawionym i uruchomionym jak dotychczas, zależnie od ustawionych parametrów czasowych na analizatorze stanów logicznych powinniśmy móc zaobserwować przebiegi podobne do poniższych (tutaj przykład dla podstawy czasu 200 ms/div):

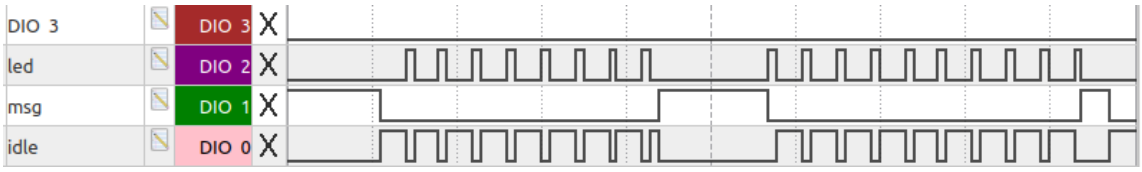


Co należy zrobić, aby udowodnić, że wiemy o co chodzi? (czyli to, co dawniej było nazywane "sprawozdaniem")

Dla każdego punktu poniżej należy wykonać osobny zrzut ekranu i kopię pliku **`main.c`** nadając im odpowiednie nazwy wg dalszego opisu. Dysponując własnymi screenshotami, na ich podstawie można udzielić odpowiedzi na polecenia z kolejnych podpunktów:

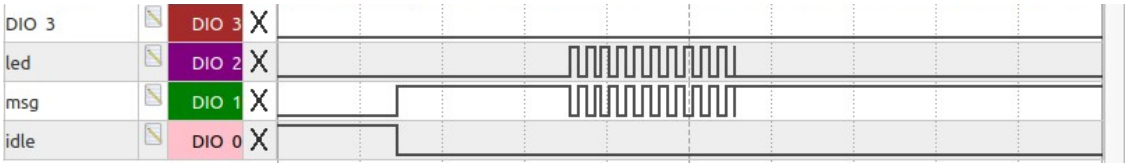
1. Aktualnie priorytet **`ledTask`** powinien być ustawiony na 2, a **`messageTask`** na 1. Wykonujemy screenshot ze stanu obecnego i nazywamy go ***led_task_1***. Wykonujemy kopię **`main.c`** i nadajemy jej nazwę **`main_led_task_1.c`** Screenshot umieszczamy na UPEL, a później opowiemy osobie prowadzącej co widzimy na przebiegach czasowych: w których chwilach aktywne jest zadanie bezczynności, w których aktywne jest zadanie **`messageTask`**, a w których **`ledTask`**. Odnosimy przebiegi czasowe do błyskania diod LED na płytce testowej Nucleo - jesteśmy w stanie bez problemu wskazać, która dioda LED odpowiada któremu zadaniu.

2. Zmieniamy priorytet zadania **`messageTask`** na wyższy niż **`ledTask`**. Możemy to zrobić np. ustawiając priorytet **`messageTask`** na **`osPriorityNormal+3`**. Programujemy mikrokontroler zmodyfikowanym w ten sposób kodem i obserwujemy przebiegi. Wykonujemy zrzut ekranu (**`led_task_2`**) oraz kopię pliku main.c (**`main_led_task_2.c`**). Zaobserwowane przebiegi mogą wyglądać podobnie do poniższych:



W ramach sprawozdania zalecane jest abyśmy potrafili opowiedzieć osobie prowadzącej: co się zmieniło, skąd takie zmiany i ogólnie... o co właściwie może tu chodzić?

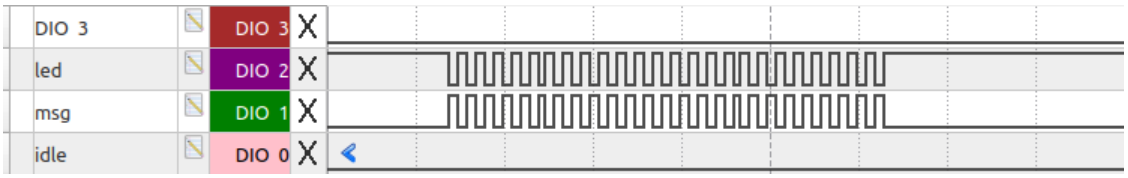
3. Zmieniamy priorytet zadania **`messageTask`** na **równy** priorytetowi **`ledTask`** - np. oba na **`osPriorityNormal+2`**. Wykonujemy zrzut ekranu (**`led_task_3`**) oraz kopię pliku **`main.c`** (**`main_led_task_3.c`**). Tym razem oczekujemy przebiegów podobnych do poniższych:



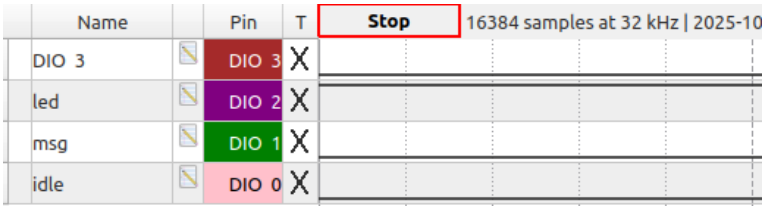
4. Tymczasowo zmieniamy funkcję opóźniającą w zadaniu **ledTask** z **vTaskDelay** na **HAL_Delay**, a także na samym początku zadania dodajemy krótkie opóźnienie vTaskDelay – zalecanym czasem opóźnienia jest tutaj 100 tyknięć, co odpowiada czasowi 100 ms. Do obserwacji warto ustawić priorytet zadania **ledTask** na niższy lub równy priorytetowi **messageTask**.

```
void ledTask(void* p){
  vTaskDelay(100);
  while(1){
    LD2_ON;
    //zajmujemy czas procesora
    HAL_Delay(20);
    LD2_OFF;
#if 0
    //zwalniamy czas procesora
    vTaskDelay(60);
#else
    //dalej zajmujemy czas procesora
    HAL_Delay(60);
#endif
  }
}
```

Tym sposobem zadanie powinno zajmować cały dostępny czas procesora od chwili wejścia do nieskończonej pętli. Wykonujemy zrzut ekranu (**led_task_4**) oraz kopię pliku **main.c** (**main_led_task_4.c**). Do sprawozdania umiemy wyjaśnić, co stało się z czasem aktywności zadania bezczynności *idle task*? Jak możemy nazwać taki stan? Czy taki stan jest pożądany czy nie i dlaczego?



5. Przy zadaniu **ledTask** z obiema funkcjami opóźniającymi ciągle jeszcze realizowanymi przez **HAL_Delay** zmieniamy priorytet tego zadania na wyższy niż **messageTask**, rejestrujemy przebiegi czasowe i - analogicznie - zastanawiamy się, co się stało i dlaczego?



6. Gdy skończymy rejestrację i obserwacje przebiegów czasowych w tym punkcie, przywracamy zmodyfikowane opóźnienie w zadaniu **ledTask** z powrotem na **vTaskDelay** (to ważne zanim przejdziemy dalej). Usuwamy także wywołanie funkcji **vTaskDelete** z zadania **StartDefaultTask**.

```
void ledTask(void* p){
  vTaskDelay(100);
  while(1){
    LD2_ON;
    //zajmujemy czas procesora
    HAL_Delay(20);
    LD2_OFF;
    //zwalniamy czas procesora
    vTaskDelay(60);
  }
```

}

5.2 Zastosowanie kolejek

Przy realizacji tej części ćwiczenia przyda się funkcja inkey. Przykład jej zastosowania możemy znaleźć w projekcie startowym w funkcji **`terminal_iface`**:

```
void terminal_iface(void){
  uint8_t key = inkey();
  if(key > 0){
    xprintf("odebrano znak\n");
    switch(key){
    case ' ':
      xprintf("space pressed!\n");
      break;
    case 't':
      cdc_transmit((uint8_t*)"abc ", 4);
      break;
    }
  }
}
```

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.

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`**). 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>`**”.

W ramach sprawozdania na podstawie przeprowadzonych prac powinniśmy być w stanie opowiedzieć osobie prowadzącej:

1. co się stanie, gdy wyślemy z terminala kilka znaków w krótkim czasie – czy znaki „przepadną” czy może zostaną tymczasowo przechowane w kolejce?
2. czy cokolwiek zmieni się, jeśli zmienimy priorytet zadania **`messageTask`** na **`osPriorityNormal+3`** a **`ledTask`** na **`osPriorityNormal+2`**?

Wykonujemy zrzut ekranu obrazujący działanie tej części programu (zapisujemy jako **`queue`**) oraz tworzymy kopię pliku **`main.c`** (zapisujemy jako **`main-queue.c`**). Oba pliki wysyłamy na UPEL.

5.3 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. Do zawieszania i wznawiania działania zadania służą funkcje **vTaskSuspend()** praz **vTaskResume()**. Znalezienie ich dokumentacji nie jest trudne ;-) Hint: przyda się utworzony poprzednio handler do do zadania **messageTask**.

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_sus.c** i treścią w stanie po zakończeniu realizacji tego punktu.

5.4 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). Na początek można ustawić takie same priorytety obu zadań np. na **osPriorityNormal+1**.

1. Tworzymy zadania **semTaskA** i **semTaskB**.
2. W każdym z tych zadań umieszczamy kod, który:
    - włączy „swoją" diodę LED, np. LD2 w *semTaskA*, i LD3 w *semTaskB*,
    - czeka chwilę,
    - wyłączy „swoją" diodę LED,
    - czeka ponownie, tylko tym razem 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** jako zmienną globalną w main.c 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_sem.c** i treścią w stanie po zakończeniu realizacji tego punktu.

5.5 Archiwizacja katalogu projektu

Katalog ze zmodyfikowanym projektem warto zarchiwizować na własnym nośniku lub dostępnym serwerze. 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.

6 Zagadnienia

Wymienione w niniejszej sekcji zagadnienia proszę opracować samodzielnie, a następnie dla każdego punktu utrwalić zdobytą wiedzę i umiejętności.

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. czym różni się *vTaskDelay* od *HAL_Delay*?
   h. 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. Możliwości śledzenia wykonywania zadań - zasada działania, rozumienie jak to działa, że

mogliśmy oglądać przebiegi czasowe z pracy systemu operacyjnego za pomocą analizatora stanów logicznych?

6. 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.