# Systemy operacyjne dla urządzeń wbudowanych

**dr hab. Dariusz Makowski**

# Systemy operacyjne dla urządzeń wbudowanych (1)

## Odróbka zajęć:

## Czwartek 21.05.2020: 8.15-10.00

# Systemy operacyjne dla urządzeń wbudowanych (1)

## Dlaczego potrzebny jest system operacyjny ?

- Rosnący poziom złożoności zadań wykonywanych przez współczesne systemy wbudowane
- Implementacje skomplikowanych algorytmów przetwarzania danych
- Programowanie współczesnych urządzeń peryferyjnych ze względu na złożoność sprzętu stało się zadaniem trudnym i czasochłonnym.
- Brak zunifikowanych interfejsów oraz różnorodność udostępnianych przez producentów funkcji sprawia, że kod źródłowy nie jest bezpośrednio przenośny nawet pomiędzy różnymi platformami tego samego producenta.

# System operacyjny

**System Operacyjny OS (Operating System)** - oprogramowanie, które zarządza sprzętem oraz aplikacjami komputera (procesora). Podstawą wszystkich systemów operacyjnych jest wykonywanie podstawowych zadań takich jak: zarządzanie pamięcią, przydział czasu procesora, obsługa urządzeń, ustalanie połączeń sieciowych oraz zarządzanie plikami

- Systemy operacyjne oparte na architekturze z jądrem - Linux
- Systemy operacyjne oparte na architekturze z mikrojądrem (QNX, FreeRTOS). Mikrojądro dostarcza minimalną ilość usług wymagana do pracy wielozadaniowej

# System operacyjny czasu rzeczywistego

„…The ability of the operating system to provide a required level of service in a bounded response time…”

**- POSIX Standard 1003.1**

- Systemem czasu rzeczywistego określa się taki system, którego wynik przetwarzania zależy nie tylko od jego logicznej poprawności, ale również od czasu, w jakim został osiągnięty
- System czasu rzeczywistego odpowiada w sposób przewidywalny na bodźce zewnętrzne napływające w sposób nieprzewidywalny
- Poprawność pracy systemu czasu rzeczywistego zależy zarówno od wygenerowanych sygnałów wyjściowych jak i spełnionych zależności czasowych
- Z pojęciem „czasu rzeczywistego” wiąże się wiele nadużyć. Terminu tego używa się potocznie dla określenia obliczeń **wykonywanych bardzo szybko**, co nie zawsze jest prawdą.

# System czasu rzeczywistego

- **Czas reakcji systemu** – przedział czasu potrzebny systemowi operacyjnemu na wypracowanie decyzji (sygnału wyjściowego) w odpowiedzi na zewnętrzny bodziec (sygnał wejściowy)
- **Czas reakcji systemu** może wahać się w granicach od ułamków sekund (np.: system akwizycji danych z kamery) do kilkudziesięciu godzin (np.: system sterowania poziomem wody w zbiorniku retencyjnym)
- Charakterystyka różnych zadań aplikacji musi być znana *a priori*
- Systemy czasu rzeczywistego, w szczególności systemy dynamiczne, muszą być szybkie, przewidywalne, niezawodne i adoptowalne


*Dr hab. Dariusz Makowski (dmakow@dmcs.pl)*

# Podział systemów czasu rzeczywistego

- **Systemy o ostrych ograniczeniach czasowych** (ang. *hard real-time*) – przekroczenie terminu powoduje katastrofalne skutki (zagrożenie życia lub zdrowia ludzi, uszkodzenie lub zniszczenie urządzenia). Nie jest istotna wielkość przekroczenia terminu, a jedynie sam fakt jego przekroczenia

- **Systemy o miękkich lub łagodnych ograniczeniach czasowych** (ang. *soft real-time*) - gdy przekroczenie terminu powoduje negatywne skutki. Skutki są tym poważniejsze, im bardziej termin został przekroczony

- **Systemy o mocnych ograniczeniach czasowych** (ang. *firm real-time*) - gdy fakt przekroczenia terminu powoduje całkowitą nieprzydatność wypracowanego przez system wyniku. Fakt niespełnienia wymagań czasowych nie stanowi jednak zagrożenia dla ludzi lub urządzenia (bazy danych czasu rzeczywistego)

# Cechy systemu czasu rzeczywistego

- Źródła niedeterminizmu czasowego w systemach operacyjnych
  - Cache miss
  - Page miss
  - Task scheduling
- Rezygnacja z obsługi pamięci wirtualnej
- Sposób szeregowania zadań – popularne systemy operacyjne stosują bardzo „sprawiedliwe” metody
  - Procesy czasu rzeczywistego muszą być uprzywilejowane
  - Brak faworyzowania użytkowników terminalowych oraz programów interaktywnych
  - Brak wywłaszczania procesów jądra
  - Kompromis pomiędzy wielozadaniowością a spadkiem wydajności związanej z przełączaniem zadań
- Zdefiniowane mechanizmy komunikacji międzyprocesowej oraz synchronizacji
  - Semafory binarne, semafory ogólne, muteksy,
  - Sygnały, kolejki komunikatów, rejestry zdarzeń


*Dr hab. Dariusz Makowski (dmakow@dmcs.pl)*

# Jądro i mikrojądro

- Jądro (kernel) systemu operacyjnego dostarcza moduły realizujące trzy podstawowe funkcje: planowanie (scheduling), dyspozycja (dispatching) i usługi komunikacji i synchronizacji
- Planista (scheduler) realizuje algorytm określający które zadanie zostanie uruchomione oraz w jakiej kolejności
- Dyspozytor (dispatcher) zarządza strukturami niezbędnymi do uruchamiania zadań. Mechanizmy komunikacji i synchronizacji obejmują: semafory, monitory, kolejki komunikatów i inne.
- Mikrojądrem nazywamy jądro o funkcjonalności ograniczonej do modułu planowania i dyspozytora, mikrojądro realizuje wielozadaniowość na poziomie procesów/zadań
- Nanojądro obsługuje jedynie zarządzanie wątkami


*Dr hab. Dariusz Makowski (dmakow@dmcs.pl)*

# Dlaczego Linux nie jest systemem czasu rzeczywistego

- Zastosowany algorytm szeregowania z podziałem czasu
- Niska rozdzielczość zegara systemowego
- Nie wywłaszczalne jądro (nie dotyczy wersji > 2.6)
- Wyłączanie obsługi przerwań w sekcjach krytycznych
- Zastosowanie pamięci wirtualnej
- Optymalizacja wykorzystania zasobów sprzętowych

# System czasu rzeczywistego RTLinux

- System czasu rzeczywistego zaprojektowany na początku lat 90
- Występuje w dwóch odmianach:
  - RTLinux GPL
  - RTLinux Pro
- Zrezygnowano z poprawek czasu rzeczywistego wprowadzanych do jądra systemu Linux
- Jądro systemu operacyjnego traktowane jest jak proces działający w ramach jądra czasu rzeczywistego. Jest ono wykonywane wtedy gdy nie ma żadnych zadań czasu rzeczywistego
- Blokowanie przerwań oferowane przez jądro Linuxa zostało zablokowane – jeżeli przerwanie nie posiadające procedury obsługi działającej na poziomie jądra czasu rzeczywistego ma być zablokowane, zostaje przechwycone przez jądro czasu rzeczywistego i zakolejkowane do wykonania w momencie gdy jądro Linuxa odblokuje przerwania


*Dr hab. Dariusz Makowski (dmakow@dmcs.pl)*

# System czasu rzeczywistego RTLinux

- Zadania czasu rzeczywistego uruchamiane są w trybie jądra RT, działają we wspólnej przestrzeni adresowej, mogą komunikować się za pomocą zmiennych globalnych, mają największy priorytet i zmniejszony narzut na przełączanie. Działają jak sterowniki (moduły) jądra.

- Planista działa w oparciu o priorytety – w przypadku wystąpienia dwóch zadań o takich samych priorytetach wybierane jest to które zostało znalezione wcześniej, nie ma podziału czasu procesora. Planista uruchamiany jest tylko wtedy gdy pojawi się proces o wyższym priorytecie lub proces działający przechodzi do stanu blokowania

- Możliwe działanie planisty w trybie dynamicznego przydzielania priorytetów

- Wymiana danych pomiędzy zadaniami RT a zwykłymi za pomocą:
  - Nieblokujących kolejek czasu rzeczywistego
  - Pamięci dzielonej


*Dr hab. Dariusz Makowski (dmakow@dmcs.pl)*

# System czasu rzeczywistego QNX

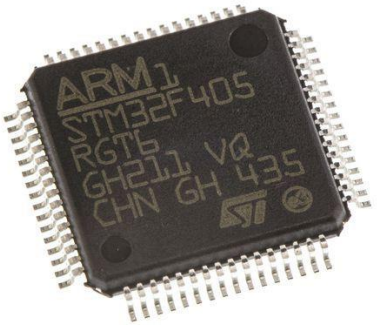
- Opracowany na początku lat 80
- Pierwszy wielozadaniowy system dla komputerów klasy x86
- Jego interfejs graficzny mieścił się w 1 MB pamięci
- Oparty o model klient-serwer
- Oparty o architekturę mikrojądra (rozmiar jądra 8 kB)
  - Kolejkowanie zadań
  - Sterowanie przekazywaniem wiadomości – lokalne jak i w obrębie sieci lokalnej
  - Przechwytywanie przerwań i przekierowywanie ich do odpowiednich procesów
- Obsługa systemu plików, przerwań, sieci, administrowanie procesami, gospodarka pamięcią operacyjną wykonywana jest za pomocą dodatkowych aplikacji działających w odrębnych przestrzeniach adresowych
- Procesy systemu operacyjnego działają tak samo jak procesy użytkownika – mają jednak inne poziomy uprzywilejowania

# System czasu rzeczywistego QNX

- Szeregowanie zadań:
  - Realizowane na podstawie priorytetu
  - W przypadku wystąpienia wielu zadań o takim samym priorytecie:
    - Wybierany jest pierwszy dostępny i wykonywany do momentu zakończenia działania, lub
    - Każdy ma przydzielany kwant czasu
    - Każdy ma przydzielany kwant czasu a po jego upłynięciu priorytet zadania jest obniżany

# System operacyjny FreeRTOS

- FreeRTOS (Free Real Time Operating System)
- System operacyjny dedykowany dla systemów wbudowanych wykorzystujących mikrokontrolery lub niewielkie procesory
- Zaprojektowany pod kątem minimalnego zapotrzebowania na zasoby sprzętowe z myślą o niewielkich mikrokontrolerach
- Dostępny w ramach licencji GPLv2 (do 2018)
- Opracowany przez firmę Real Time Engineers Ltd.
- W 2017 roku przejęty przez Amazon, włączony do projektu AWS (Amazon Web Services) open source i dostępny na licencji MIT
- Bardzo popularny – dostępny prawie na każda platformę sprzętową



# System operacyjny FreeRTOS (2)

- FreeRTOS oparty jest na mikrojądrze czasu rzeczywistego (real-time scheduler)
- Spełnia wymagania systemów o twardych ograniczeniach czasowych
  - Prosty
  - Przenośny
  - Darmowy
  - Zwięzły
- Głównie napisany w języku C
- Niewielka liczba funkcji asemblera
- Wspiera tryb Thumb procesorów ARM

# System operacyjny FreeRTOS (2)

Cechy systemu FreeRTOS:

- Planista pracuje w trybie podziału czasu procesora (Pre-emptive) lub współpracujących zadań (co-operative multitasking)
- Możliwość łatwego definiowania priorytetów zadań
- Kolejki (queues)
- Semafory binarne (binary semaphores)
- Semafory zliczające (counting semaphores)
- Semafory rekurencyjn (recursive semaphores)
- Muteksy (Mutexes)
- Funkcja Tick hook
- Wsparcie dla zadania idle (idle hook functions)
- Kontrola przepełnienia stosu (stack overflow checking)
- Makra pomocne podczas śledzenia oraz debugowania programu (trace hook macros)


*Dr hab. Dariusz Makowski (dmakow@dmcs.pl)*

# System operacyjny FreeRTOS (2)

Cechy systemu FreeRTOS:

- Niewielkie zużycie pamięci (typowa implementacja wykorzystuje ok. 6 kB pamięci programu i kilkaset bajtów pamięci danych)
- Każde zadanie wykorzystuje pamięć danych na przechowywanie kontekstu oraz stos
- Wykorzystuje przerwanie timera PIT (Periodic Interval Timer) do generacji tików systemowych SysTick

# Czego nie zapewnia system FreeRTOS

- FreeRTOS nie oferuje
- Wsparcia dla mechanizmu sterowników urządzeń
- Zaawansowanych mechanizmów zarządzania pamięcią
- Zaawansowanych algorytmów pracy schedulera
- Obsługi sieci (obsługa sieci może być zrealizowana za pomocą dodatkowych bibliotek)
- Obsługi kont użytkowników w jakiejkolwiek postaci

# Licencja

- Licencja z otwartym kodem źródłowym (Open source licence)
- Może zostać wykorzystany do aplikacji komercyjnych
- Użytkownicy końcowy są właścicielem opracowanego IP
- Więcej informacji na: http://www.FreeRTOS.org/license

# FreeRTOS - wybrane platformy sprzętowe

- Atmel: AVR, AVR32–SAM3, SAM4, SAM7, SAM9, SAM D20, SAM L21
- ARM: ARM7, ARM9, ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M3, ARM Cortex-M4, ARM Cortex-A
- Intel: X86, 8052
- IBM: PPC405, PPC404
- Microchip: PIC18, PIC24, dsPIC, PIC32
- NXP: LPC1000, LPC2000, LPC4300
- STMicroelectronics: STM32, STR7
- Xilinx: MicroBlaze, ARM

# FreeRTOS - Scheduler

- Scheduler (wywłaszczający lub niewywłaszczający) - odpowiada za kontrolę wykonywanych zadań i zarządza przydziałem czasu procesora
- Obsługuje priorytety zadań
- Podstawową jednostką czasu jest tick (wynoszący zwykle 1-10 ms)
- Operacje na zadaniach wpierane przez scheduler
  - Utworzenie i usunięcie
  - Zawieszenie i wznawianie
  - Opóźnienie wykonywania

# Kluczowe komponenty systemu FreeRTOS

- Zarządzania zadaniami
- Zarządzania kolejkami
- Zarządzania przerwaniami
- Zarządzania zasobami

- Więcej: patrz FreeRTOS manual:
  https://www.freertos.org/Documentation/FreeRTOS_Reference_Manual_V9.0.0.pdf

# Typy stałych oraz zmiennych

- Stałe zdefiniowane w pliku ProjDefs.h

| Definition | Value |
|---|---|
| pdTRUE[a] | 1 |
| pdFALSE | 0 |
| pdPASS | 1 |
| pdFAIL | 0 |

- Typy wykorzystywane do konfiguracji portów

| Definition | Value |
|---|---|
| portCHAR | char (type) |
| portLONG | long (type) |
| portSHORT | short (type) |
| portBASE_TYPE | Port-dependent [a]– defined to be the most efficient data type for the architecture |
| portMAX_DELAY | Port-dependent |
| portTickType | Port-dependent |


*Dr hab. Dariusz Makowski (dmakow@dmcs.pl)*

# Konwencja nazewnictwa

- Stałe zdefiniowane w pliku ProjDefs.h

| Parameter Name | Type | Prefix |
|---|---|---|
| Variables | portCHAR | c |
| | portSHORT | s |
| | portLONG | l |
| | portBASE_TYPE | x |
| | structures, and so on | x |
| | void | v |
| Pointers | — | p |
| Unsigned variables | — | u |

- psXXX – wskaźnik do typu short
- pvXXX - wskaźnik do typu void
- usXXX - wskaźnik do typu unsigned short

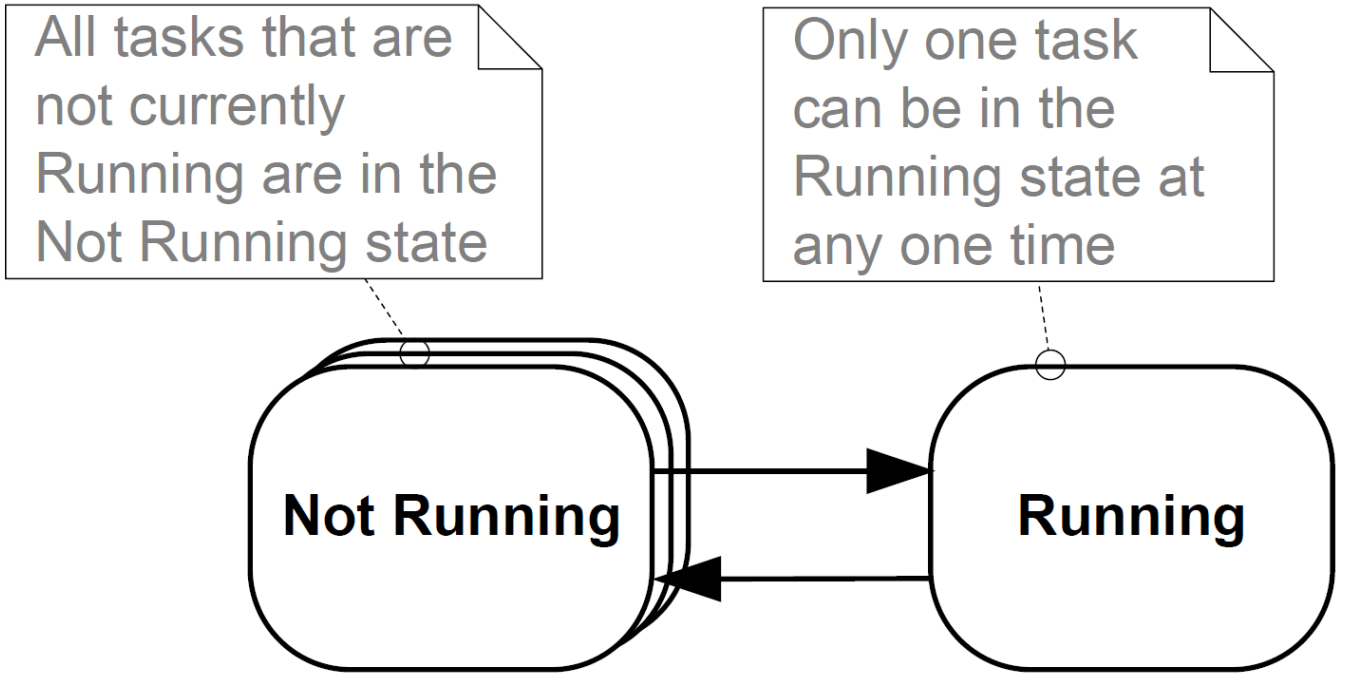
- Wartości funkcji również posiadają prefiksy

# FreeRTOS - Zadania

- Podstawowym elementem systemu są zadania
- Zbudowane zwykle z blokujących funkcji
- Zadania opisane są przy pomocy struktury kontekstu
- Każde zadanie posiada:
  - Indywidualny stos – konfigurowalny rozmiar dla każdego zadania
  - Priorytet – decyduje o kolejności wykonywanych zadań
  - Opcjonalny uchwyt do zdania (handler) – pozwala na zarządzanie zadaniem przez inne elementy systemu
- Zadanie tworzone domyślnie – IDLE Task (pomiar zapasowej mocy obliczeniowej), oszczędność energii: low power mode

# FreeRTOS – Zadania (2)

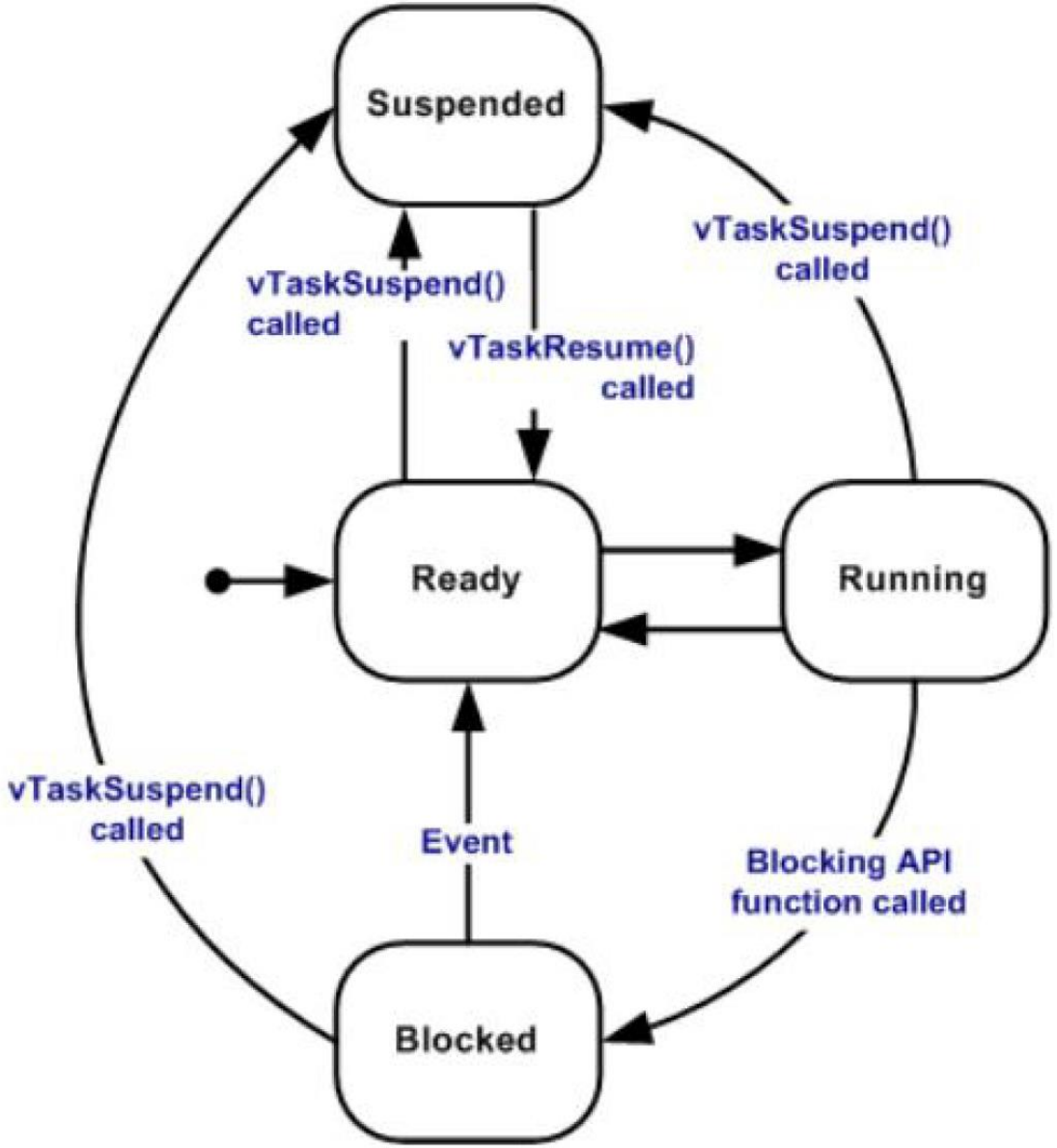
- Procesor wyposażony w 1 rdzeń może wykonywać tylko 1 zadanie w danym czasie



# FreeRTOS – Stan Zadania

- **Running** – zadanie wykonywane, przydzielony czas procesora
- **Ready** – zadanie gotowe do wykonywania, czeka na czas procesora
- **Blocked** – zadanie zablokowane - oczekuje na zdarzenie zewnętrzne lub czasowe, np. vTaskDelay() jest 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** – zadanie oczekujące, nieobsługiwane przez scheduler, sterowanie zadaniami przez funkcje: vTaskSuspend() oraz vTaskResume()

# FreeRTOS – Podsumowanie



# Tworzenie zadania

```
portBASE_TYPE xTaskCreate (
  pdTASK_CODE pvTaskCode,
  const portCHAR* const pcName,
  unsigned portSHORT usStackDepth,
  void*pvParameters,
  unsigned portBASE_TYPE uxPriority,
  xTaskHandle *pvCreatedTask
);
```

# Tworzenie zadania (2)

- **pvTaskCode** - wskaźnik do kontekstu (struktury) opisującego zadanie
- **pcName** - nazwa zadania (opcjonalna)
  - Makro configMAX_TASK_NAME_LEN określa długość nazwy
- **usStackDepth** - wielkość stosu (long int => 4 * 100 Bajtów)
  - Makro configMINIMAL_STACK_SIZE określa min. rozmiar stosu
- **pvParameters** - opcjonalne parametry (void*), które zostaną przekazane do kodu zadania
- **uxPriority** - priorytet zadania (0 - najmniejszy)
  - Makro configMAX_PRIORITIES określa max. Liczbę priorytetów
- **pvCreatedTask** - opcjonalny wskaźnik do uchwytu zadania

- Stałe umieszczone w pliku: FreeRTOSConfig.h

# Tworzenie zadania (3)

- Zwracane wartości
  - **pdPASS** – zadanie utworzone z sukcesem
  - **pdFAIL** – problem podczas tworzenia zadania, np. brak pamięci na stercie, powielona nazwa, etc.

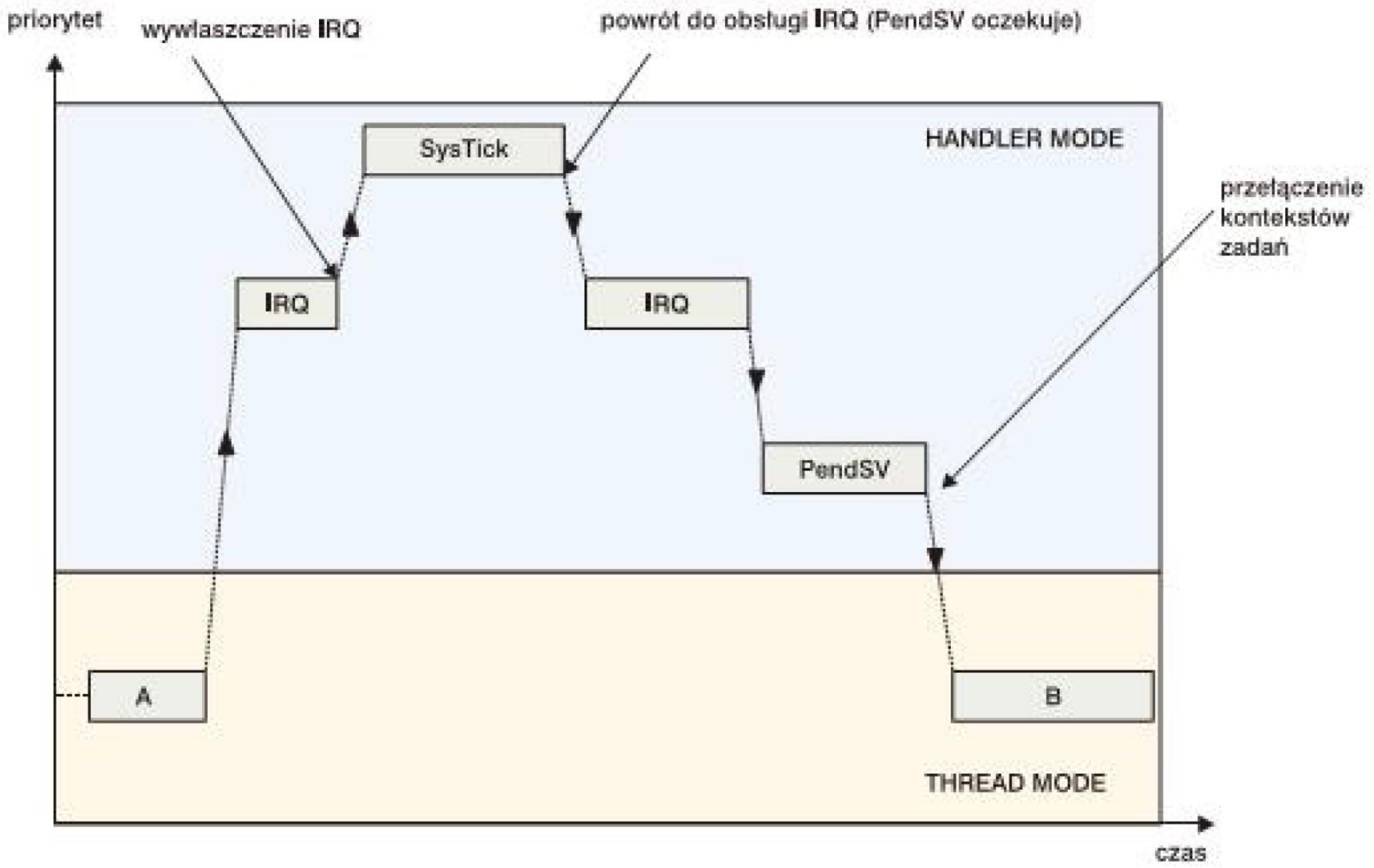
# Tworzenie zadania - przykład

```
TaskHandle_t handlTask1, handlTask2 =  null;
int main( void )
{
  /* Create one of the two tasks. Check returned value */
  xTaskCreate ( vTask1, /* Pointer to the function that implements the task */
                "Task 1", /* Text name for the task */
                1000, /* Stack depth */
                NULL, /* This example does not use the task parameter */
                1, /* This task will run at priority 1 */
                &handlTask1 ); /* This example does not use the task handle. */
  /* Create the other task in exactly the same way and at the same priority. */
          xTaskCreate ( vTask2, "Task 2", 1000, NULL, 1, handlTask2 );
  /* Start the scheduler so the tasks start executing. */
          vTaskStartScheduler ();
  /* If all is well then main() will never reach here */
  for( ;; );
}
```

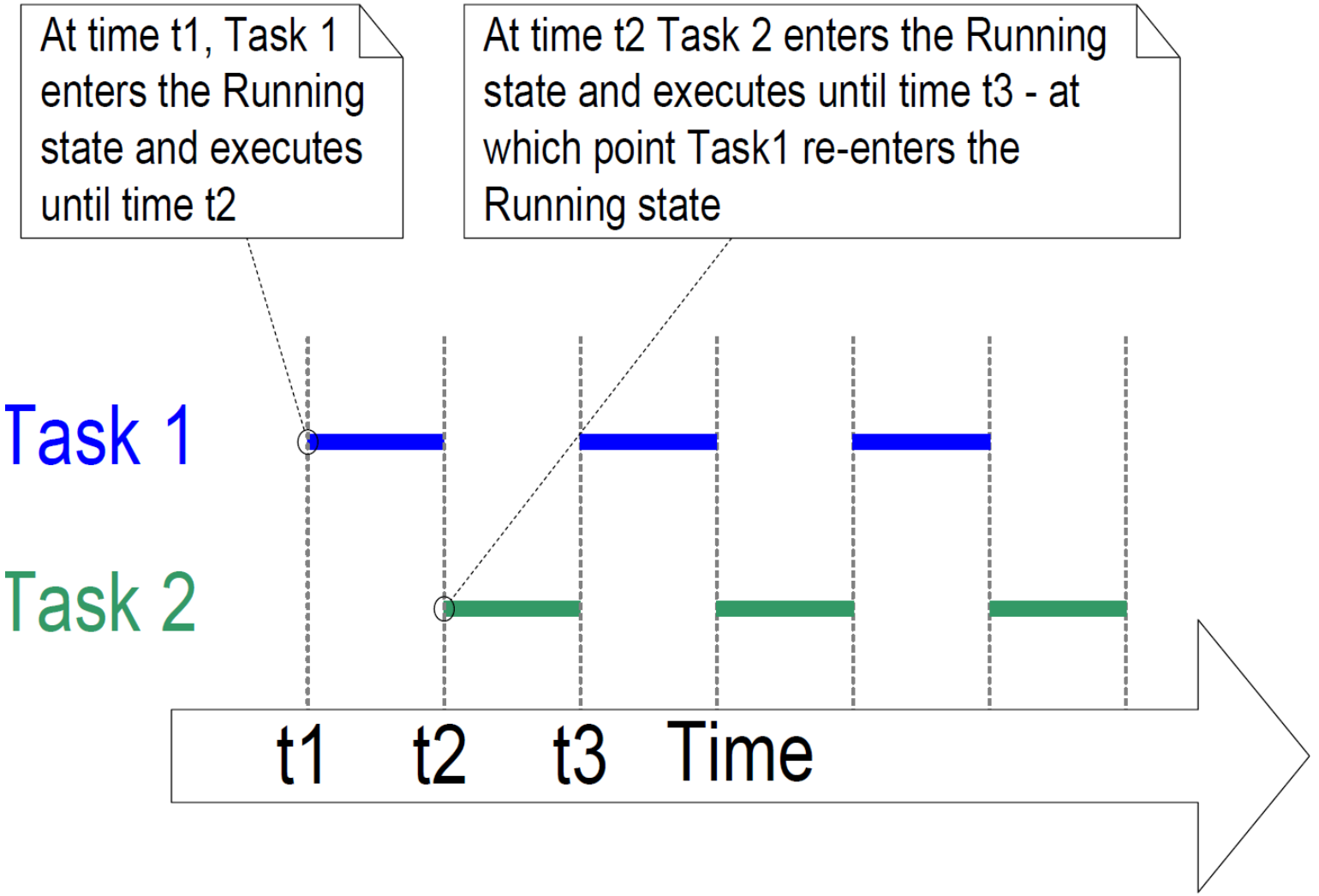
# Tworzenie zadania – przykład (2)

```
void vTask1 ( void *pvParameters )
{
        const char *pcTaskName = "Task 1 is running\r\n";
        volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

        /* As per most tasks, this task is implemented in an infinite loop. */

        for( ;; )
        {
                /* Print out the name of this task. */
                vPrintString( pcTaskName );

                /* Delay for a period. */
                for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
                {
                /* Wait and waites procesor time */
                }
        }
}
```

# Przełączenie kontekstu



# Wykonywanie zadań



# Usuwanie zadania

xTaskCreate() – tworzy zadnie

vTaskDelete() – usuwa zadnie

Przykład:

vTaskDelete( prtToTask ) – usuwa zadanie wskazywane przez prtToTask

vTaskDelete( NULL ) – usuwa aktualne zadanie

# Modyfikacje parametrów zadania

vTaskPrioritySet() – modyfikuje priorytet zadnia
configMAX_PRIORITIES – maksymalna liczba priorytetów określona wFreeRTOSConfig.h

configUSE_PORT_OPTIMISED_TASK_SELECTION =1 – liczba priorytetów ograniczona do 32, wsparcie asemblera

# Usypianie zadania

Funkcja vTaskDelay() pozwala na zmianę stanu zadania: Blocked state na określoną liczbę tików systemowych

```
voidvAnotherTask( void * pvParameters )
{
        for( ;; )
        // Do some work here
        vTaskDelay( 20 ); // Enter the Blocked state for 20 millisecond

        // pdMS_TO_TICKS() makro pozwala na określenie czasu uśpienia
           niezależnie od parametrów tiku

        vTaskDelay( pdMS_TO_TICKS( 20));
}
```

# Inne funkcje…

Funkcja uxTaskGetNumberOfTasks() zwraca liczbę zadań

Funkcja vTaskGetRunTimeStats() zwraca statystyki dot. zadań

```
Task            Abs Time      % Time
****************************************


uIP             12050         <1%
IDLE            587724        24%
QProdB2         2172          <1%
QProdB3         10002         <1%
QProdB5         11504         <1%
QConsB6         11671         <1%
PolSEM1         60033         2%
PolSEM2         59957         2%
IntMath         349246        14%
MuLow           36619         1%
GenQ            579715        24%
```

Funkcja UBaseType_tuxTaskGetStackHighWaterMark( TaskHandle_txTask ) zwraca pozostały wolny rozmiar stosu

# Inne funkcje…

Funkcja vTaskList() zwraca listę zadań

```
Name          State   Priority  Stack   Num
*******************************************
Print           R       4       331     29
Math7           R       0       417     7
Math8           R       0       407     8
QConsB2         R       0       53      14
QProdB5         R       0       52      17
QConsB4         R       0       53      16
SEM1            R       0       50      27
SEM1            R       0       50      28
IDLE            R       0       64      0
Math1           R       0       436     1
Math2           R       0       436     2
```

taskYIELD() – oddaje czas do innego zadania o takim samy priorytecie

# Zadanie IDLE

- Zadanie bezczynności IDLE – tworzone automatycznie w momencie
- uruchomienia schedulera
- Posiada najniższy priorytet
- Wykorzystywane do zwalnianie pamięci alokowanej przez system do wykonania zadań, które zostały skasowane poleceniem vTaskDelete().
- Używane do wprowadzania mikrokontrolera do stanu oszczędzającego energię

# Zadanie - podsumowanie

**Tworzenie zadań:**
xTaskCreate
xTaskCreateStatic
vTaskDelete

**Sterowanie zadaniami:**
vTaskDelay
vTaskDelayUntil
uxTaskPriorityGet
vTaskPrioritySet
vTaskSuspend
vTaskResume
xTaskResumeFromISR
xTaskAbortDelay

**Sterowanie z poziomu jądra SO:**
taskYIELD
taskENTER_CRITICAL
taskEXIT_CRITICAL
taskENTER_CRITICAL_FROM_ISR
taskEXIT_CRITICAL_FROM_ISR
taskDISABLE_INTERRUPTS
taskENABLE_INTERRUPTS
vTaskStartScheduler
vTaskEndScheduler
vTaskSuspendAll
xTaskResumeAll
vTaskStepTick

**Inne funckcje:**
uxTaskGetSystemState
vTaskGetInfo
xTaskGetCurrentTaskHandle
xTaskGetIdleTaskHandle
uxTaskGetStackHighWaterMark
eTaskGetState
pcTaskGetName
xTaskGetHandle
xTaskGetTickCount
xTaskGetTickCountFromISR
xTaskGetSchedulerState
uxTaskGetNumberOfTasks
vTaskList
vTaskStartTrace
ulTaskEndTrace
vTaskGetRunTimeStats
vTaskGetIdleRunTimeCounter
vTaskSetApplicationTaskTag
xTaskGetApplicationTaskTag
xTaskCallApplicationTaskHook
pvTaskGetThreadLocalStoragePointer
vTaskSetThreadLocalStoragePointer
vTaskSetTimeOutState
xTaskCheckForTimeOut

# Zadanie – podsumowanie (2)

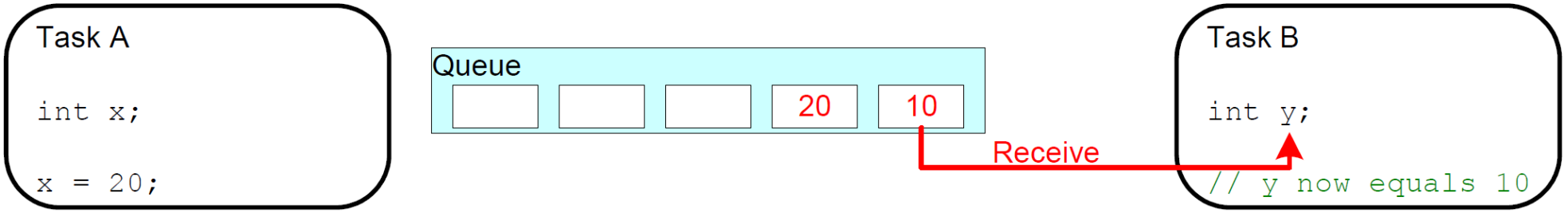
Komunikacja, zdarzenia

xTaskNotifyGive()
vTaskNotifyGiveFromISR()
ulTaskNotifyTake()
xTaskNotify()
xTaskNotifyAndQuery()
xTaskNotifyAndQueryFromISR()
xTaskNotifyFromISR()
xTaskNotifyWait()
xTaskNotifyStateClear()

# Kolejki

**Kolejki (FIFO)** - podstawowy mechanizm wykorzystywane do komunikacji między zadaniami (wymiana danych) oraz przerwaniami (komunikacja zadanie – przerwanie, przerwanie - zadanie)

- Unikamy zmiennych globalnych – zanieczyszczają program
- Przekazywanie danych wykonywanie jest przez kopiowanie
- Wiele zadań może pisać do kolejki, wiele zadań może z niej czytać
- Zadania mogą zostać automatycznie uśpione jeżeli nie ma danych w kolejce




Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration).

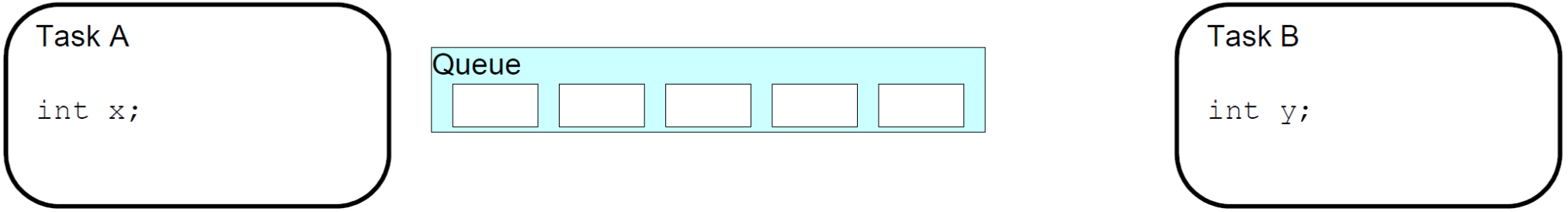
# Obsługa kolejek

xQueueHandle **xQueueCreate**(
unsigned portBASE_TYPE uxQueueLength,
unsigned portBASE_TYPE uxItemSize
);

**uxQueueLength** – długość kolejki

**uxItemSize** – wielkość pojedynczego elementu

xQueue = xQueueCreate( 5, sizeof( int32_t ) );




A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.

# Przesyłanie komunikatów

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

**xQueue** – uchwyt do kolejki

**pvItemToQueue** – element do wysłania

**xTicksToWait** – maksymalny czas (timeout) oczekiwania na umieszczenie zadania w kolejce w tikach systemowych

**xQueueSendFromISR() – bezpieczna funkcja do komunikacji z ISR**

**xQueueSendToBack()**

**xQueueSendToFront()**

# Przykład

Zwracane wartości:

pdPASS – dana zapisana poprawnie

errQUEUE_FULL – brak miejsca  w kolejce

```
xQueue1 = xQueueCreate( 10, sizeof( unsigned long ) );
if( xQueue1 != 0 )
   {
      /* Zapisz daną unsigned long.  Poczekaj 10 tików ąż będzie dostępne miejsce w kolejce*/
      if( xQueueSend( xQueue1,
                      ( void * ) &ulVar,
                      ( TickType_t ) 10 ) != pdPASS )
      {
         /* Nie udało się wysłać kominikatu */
      }
   }
```

# Odbieranie komunikatów

portBASE_TYPE **xQueueReceive**(
xQueueHandle xQueue,
void *pvBuffer,
portTickType xTicksToWait
);

**xQueue** – uchwyt do kolejki
**pvBuffer** – bufor, w którym zostanie umieszczony odebrany element
**xTicksToWait** – maksymalny czas oczekiwania (timeout) na pojawienie się elementu gotowego do odbioru (ustawienie portMAX_DELAY powoduje blokujący odczyt)

Zwracane wartości:
pdPASS – dana odczytana poprawnie
errQUEUE_EMPTY – brak danych do odczytu

# Przekład (1)

```
QueueHandle_t xQueue;

int main( void )
{
        xQueue = xQueueCreate( 5, sizeof( int32_t ) );
        if( xQueue != NULL )
        {
                xTaskCreate( vSenderTask, "Sender1", 1000, ( void * ) 100, 1, NULL );
                xTaskCreate( vSenderTask, "Sender2", 1000, ( void * ) 200, 1, NULL );

                xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2, NULL ); // wyższy priorytet
                vTaskStartScheduler();
        }
        else
        {
                /* The queue could not be created. */
        }
        for( ;; );
}
```

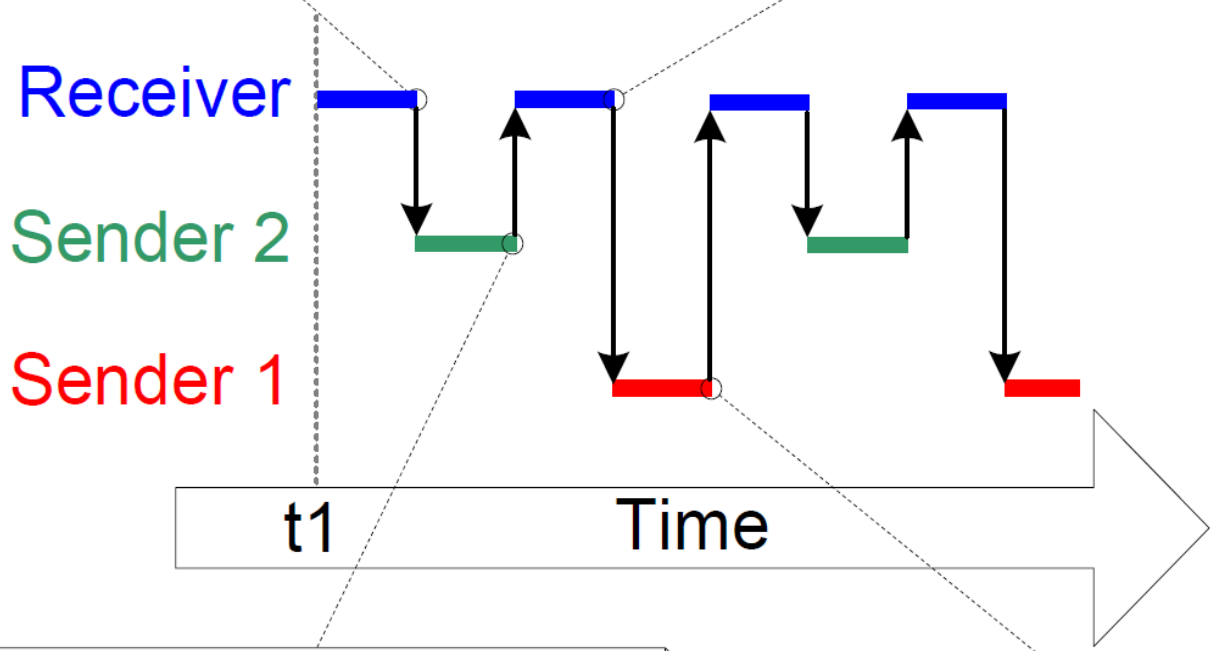
# Przekład (2)

```
static void vReceiverTask( void *pvParameters )
{
        int32_t lReceivedValue;
        BaseType_t xStatus;
        const TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );
        for( ;; )
        {
                if( uxQueueMessagesWaiting( xQueue ) != 0 )
                {
                        vPrintString( "Queue should have been empty!\r\n" );
                }
                xStatus = xQueueReceive( xQueue, &lReceivedValue, xTicksToWait );
                if( xStatus == pdPASS )
                {
                        vPrintStringAndNumber( "Received = ", lReceivedValue );
                }
                else
                {
                        vPrintString( "Could not receive from the queue.\r\n" );
                }
        }
}
```

# Przekład (3)

1 - The Receiver task runs first because it has the highest priority. It attempts to read from the queue. The queue is empty so the Receiver enters the Blocked state to wait for data to become available. Sender 2 runs after the Receiver has blocked.

3 - The Receiver task empties the queue then enters the Blocked state again. This time Sender 1 runs after the Receiver has blocked.




2 - Sender 2 writes to the queue, causing the Receiver to exit the Blocked state. The Receiver has the highest priority so pre-empts Sender 2.

4 - Sender 1 writes to the queue, causing the Receiver to exit the Blocked state and pre-empt Sender 1 - and so it goes on ……..

# Kolejki dedykowane dla przerwań

- FreeRTOS dostarcza funkcje do obsługi kolejek, które mogą być bezpiecznie używane w funkcji obsługującej przerwanie, nie blokując działania funkcji
  - xQueueSendFromISR
  - xQueueReceiveFromISR

# Kolejki - podsumowanie

**Zarządzanie kolejkami**

xQueueCreate
xQueueCreateStatic
vQueueDelete
xQueueSend
xQueueSendFromISR
xQueueSendToBack
xQueueSendToBackFromISR
xQueueSendToFront
xQueueSendToFrontFromISR
xQueueReceive
xQueueReceiveFromISR
uxQueueMessagesWaiting
uxQueueMessagesWaitingFromISR
uxQueueSpacesAvailable
xQueueReset
xQueuePeek
xQueuePeekFromISR
vQueueAddToRegistry
pcQueueGetName
vQueueUnregisterQueue
xQueueIsQueueEmptyFromISR
xQueueIsQueueFullFromISR
xQueueOverwrite
xQueueOverwriteFromISR

**Zarządzanie zdarzeniami**

xQueueCreateSet
xQueueAddToSet
xQueueRemoveFromSet
xQueueSelectFromSet
xQueueSelectFromSetFromISR


*Dr hab. Dariusz Makowski (dmakow@dmcs.pl)*

# Thank you for your attention

# Narzędzia

# STM 32

# STM32 IDE

**Procesor z rdzeniem ARM Cortex M4 firmy STM:** STM32L496ZGT6

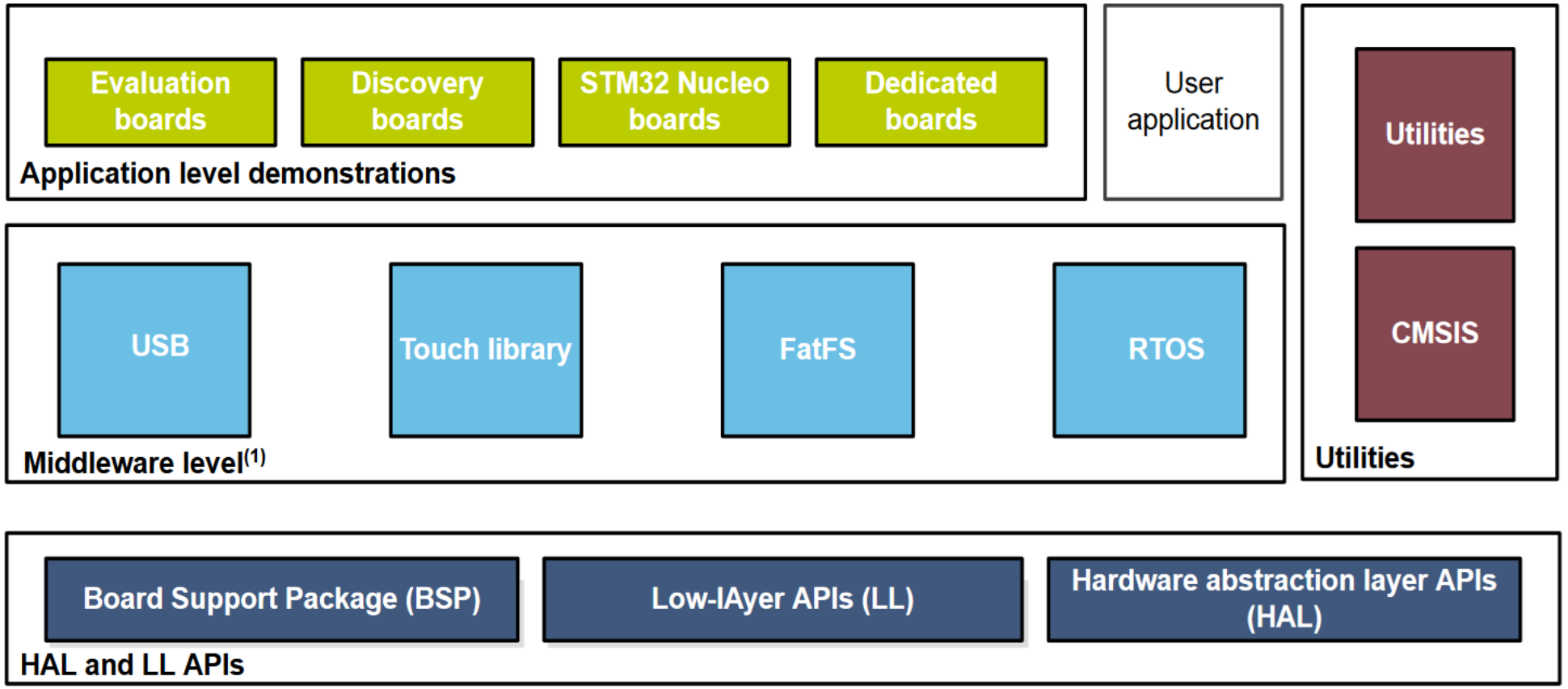
**Cube MX IDE**

- https://my.st.com/content/my_st_com/en/products/development-tools/software-development-tools/stm32-software-development-tools/stm32-ides/stm32cubeide.html#get-software
- https://www.st.com/content/ccc/resource/technical/document/user_manual/10/c5/1a/43/3a/70/43/7d/DM00104712.pdf/files/DM00104712.pdf/jcr:content/translations/en.DM00104712.pdf

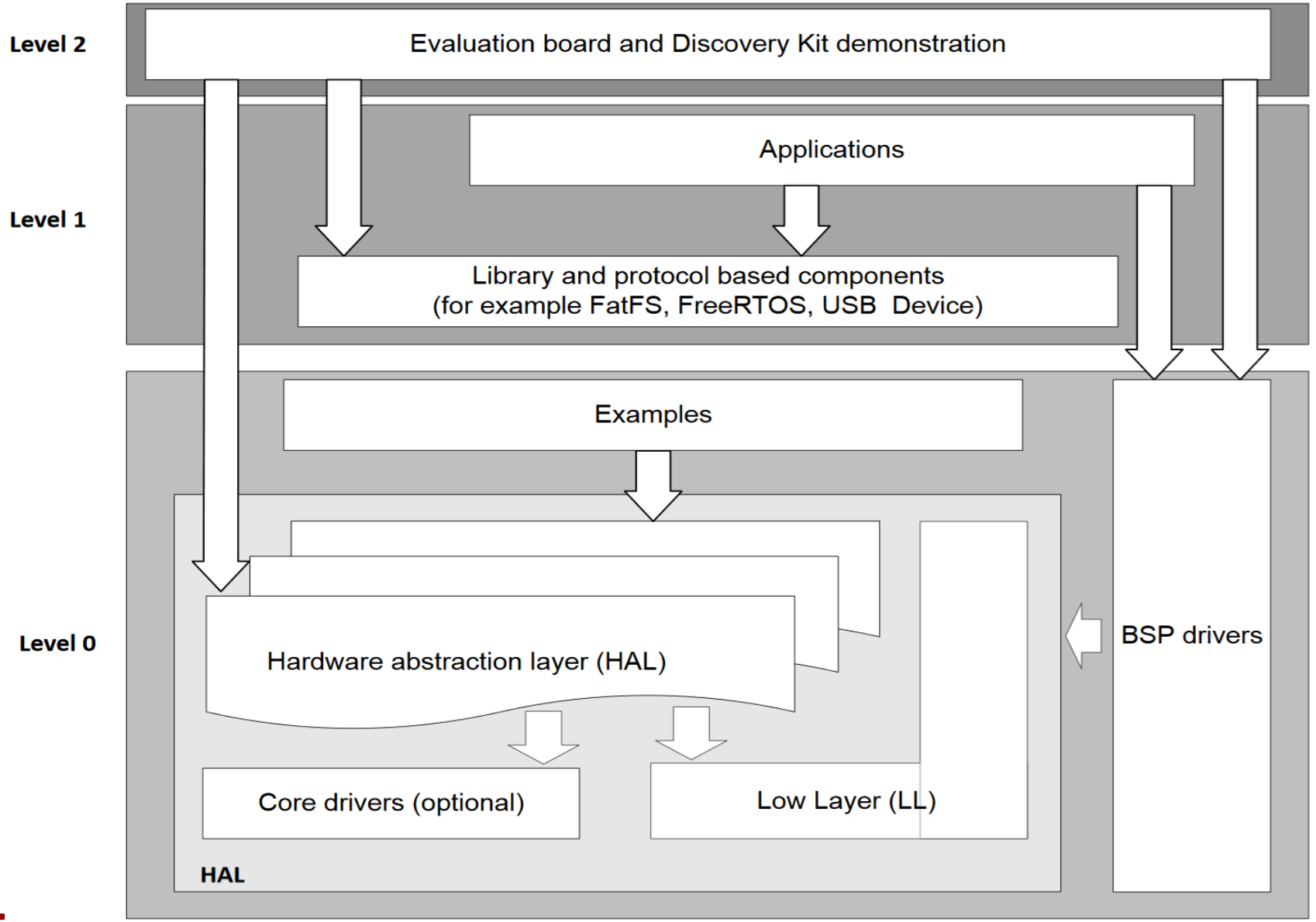
**HAL**

- https://www.st.com/content/ccc/resource/technical/document/user_manual/41/78/8e/63/f9/32/44/12/DM00113898.pdf/files/DM00113898.pdf/jcr:content/translations/en.DM00113898.pdf
- https://www.st.com/content/ccc/resource/technical/document/user_manual/e0/bc/bf/94/24/99/49/81/DM00114438.pdf/files/DM00114438.pdf/jcr:content/translations/en.DM00114438.pdf

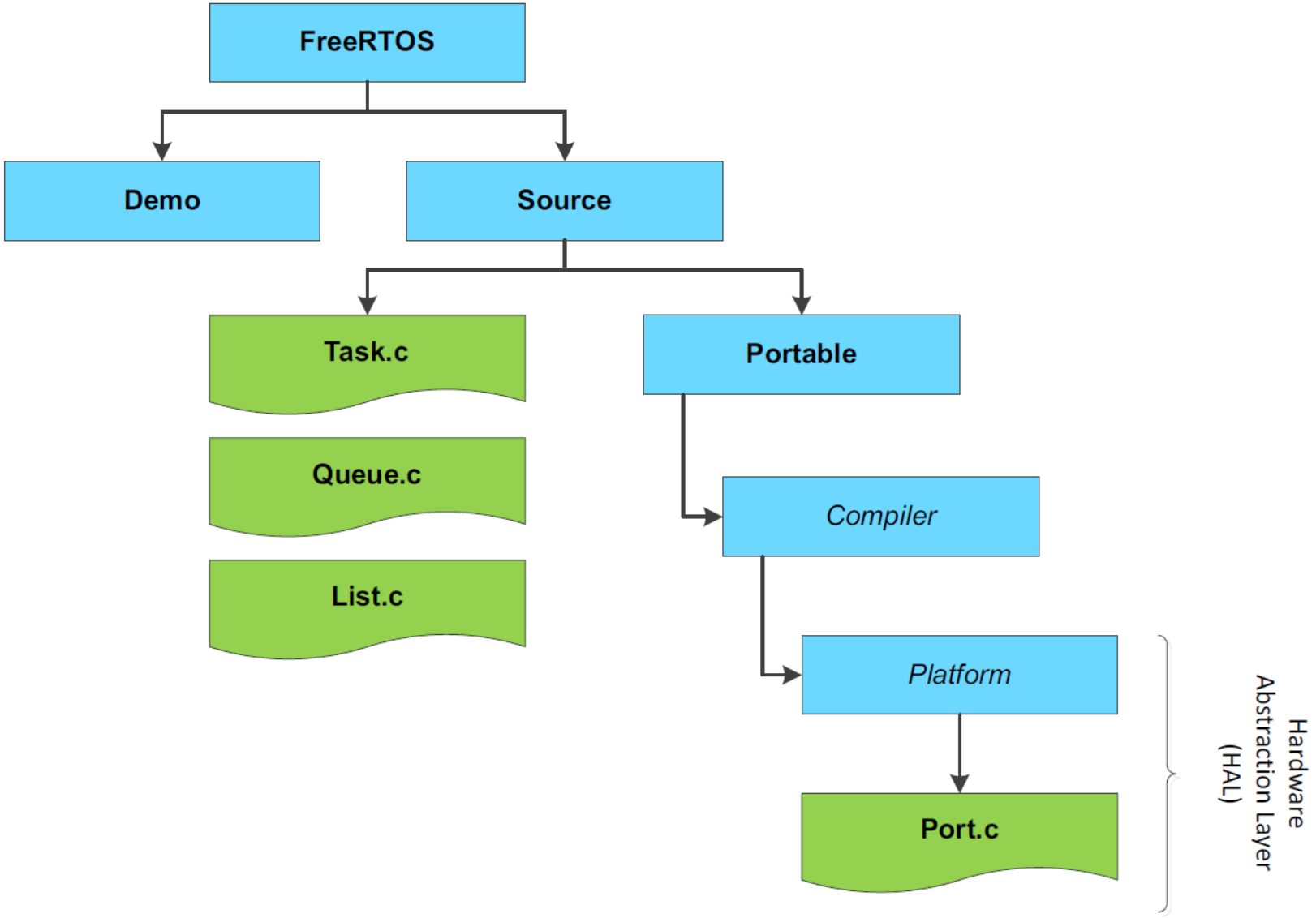
# STM32 L0 Software



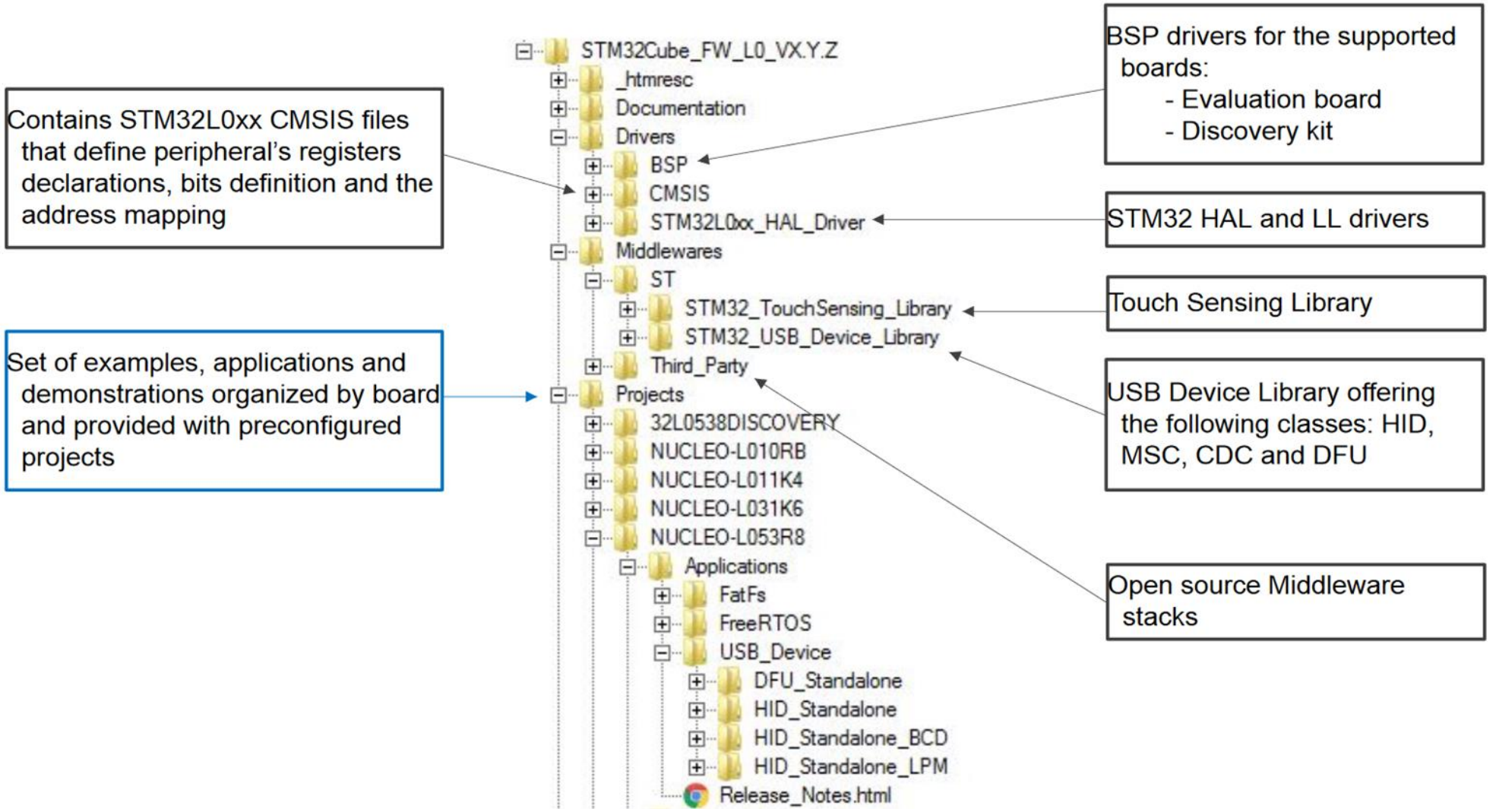
# Architektura oprogramowania STM32CubeL0



# FreeRTOS on STM32



# Struktura pakietu STM32CubeL0



# FreeRTOS

- Praktyczny tutorial dostępny na stronie FreeRTOS:

https://www.freertos.org/FreeRTOS-Plus/BSP_Solutions/ST/STM32CubeMX.html#walk-through

## STM32CubeMX Practical Walk-through

The time to market benefits of using STM32CubeMX are best demonstrated by way of a practical example, so this page provided links to a step-by-step guide to creating an IAR Embedded Workbench for ARM project in STM32CubeMX, including the STM32 pin assignments, and various middleware and peripheral driver components (including FreeRTOS!).

1. Downloading, Installing and Running STM32CubeMX
2. Downloading a Driver and Middleware Package From STM32CubeMX
3. Selecting an STM32 ARM Cortex-M MCU and Creating a New STM32MX Project
4. Selecting Peripherals and Creating an STM32 Pinout
5. Configuring Peripheral Drivers and Middleware
6. Visual Clock Tree Configuration
7. Generating an IDE Ready Embedded C Source Code Project
8. STM32 MCU Power Consumption Estimates