Struktura jądra UNIX

Wykład 2: FreeRTOS

Wprowadzenie

1. Jądro dla systemów wbudowanych (licencja MIT, Amazon)
   (FreeRTOS dla ATmega328P – port Tilka)
2. Jądro czy raczej biblioteka wątków?
   a. zadania (wątki lub korutyny) i algorytm szeregowania
   b. komunikacja i synchronizacja
   c. obsługa czasomierzy
   d. minimalistyczne zarządzanie pamięcią
3. Ograniczenia sprzętowe systemów wbudowanych:
   a. mało RAM / ROM
   b. brak ochrony pamięci i trybu uprzywilejowanego
4. Nadaje się do budowania systemów czasu rzeczywistego
   (stosowanie algorytmów o stałym czasie odpowiedzi)

Planowanie zadań w RTOS

1. Dla zadań okresowych klasycznie stosowane:
   a. Rate Monotonic Scheduling: statyczny przydział priorytetów! (im krótszy okres, tym wyższy priorytet)
   b. Earliest Deadline First: dynamiczny przydział priorytetów! (im bliżej do terminu, tym wyższy priorytet)
2. Wielozadaniowość we FreeRTOS:
   a. kooperacyjna: FIFO + priorytety + taskYIELD
   b. z wywłaszczaniem: round-robin + priorytety
   c. konfigurowalne (FreeRTOSConfig.h)
   d. priorytet `0` (najniższy), `N-1` (najwyższy)

Zarządzanie pamięcią

1. Może być całkowicie statyczne!
2. Domyślnie korzysta z `pvPortMalloc` i `vPortFree`
   (Port = część zależna od architektury / platformy)
3. 5 domyślnych implementacji (Memory Management):
   a. `heap_1.c`: alokator stosowy, nie można zwalniać pamięci
   b. `heap_2.c`: listowy, best-fit, nie ma scalania bloków
   c. `heap_3.c`: j.w., ale thread-safe
   d. `heap_4.c`: j.w., ale first-fit i złączanie wolnych bloków
   e. `heap_5.c`: j.w., ale wspiera więcej niż jeden zarządzany obszar
4. Czemu taki wybór → porozmawiamy na ćwiczeniach!

Zadania FreeRTOS

1. Tworzymy z użyciem `xTaskCreate`:
   a. z grubsza odpowiadają wątkom,
   b. stos o ustalonym z rozmiarze (jak wykrywać przepełnienia?),
   c. kod realizuje nieskończoną pętlę.
2. Uniksowy `exit` ≈ `vTaskDelete(NULL)`
3. Zadania nie są zorganizowane w hierarchię – brak `join`.
4. TCB dostępne do odczytu dla wątku:
   a. `xTaskGetCurrentTaskHandle`
   b. `vTaskGetInfo`
   c. `TaskStatus->ulRunTimeCounter` (runtime ticks)
5. Inne ciekawe (?) operacje:
   `vTaskDelay`, `xTaskAbortDelay`, `vTaskPrioritySet`, `uxTaskPriorityGet`

Przykład: Tworzenie i niszczenie zadań

```
/* Function that creates a task. */
void vOtherFunction(void) {
  TaskHandle_t xHandle = NULL;

  /* Create the task, storing the handle. */
  BaseType_t xReturned =
      xTaskCreate(vTaskCode,        /* Function that implements the task. */
                  "NAME",           /* Text name for the task. */
                  STACK_SIZE,       /* Stack size in words, not bytes. */
                  (void *)1,        /* Parameter passed into the task. */
                  tskIDLE_PRIORITY, /* Priority at which the task is created. */
                  &xHandle); /* Used to pass out the created task's handle. */

  if (xReturned == pdPASS) {
    /* The task was created.  Use the task's handle to delete the task. */
    vTaskDelete(xHandle);
  }
}
```

Informacje o zadaniu

```
typedef struct xTASK_STATUS {
  ...

  /* The state in which the task existed when the structure was populated. */
  eTaskState eCurrentState;
  /* The priority at which the task was running (may be inherited) when the
  structure was populated. */
  UBaseType_t uxCurrentPriority;
  /* The priority to which the task will return if the task's current priority
  has been inherited to avoid unbounded priority inversion when obtaining a mutex. */
  UBaseType_t uxBasePriority;
  /* The total run time allocated to the task so far [...] */
  unsigned long ulRunTimeCounter;
  /* Points to the lowest address of the task's stack area. */
  StackType_t *pxStackBase;
  /* The minimum amount of stack space that has remained for the task since
  the task was created.  The closer this value is to zero the closer the task
  has come to overflowing its stack. */
  configSTACK_DEPTH_TYPE usStackHighWaterMark;
} TaskStatus_t;
```

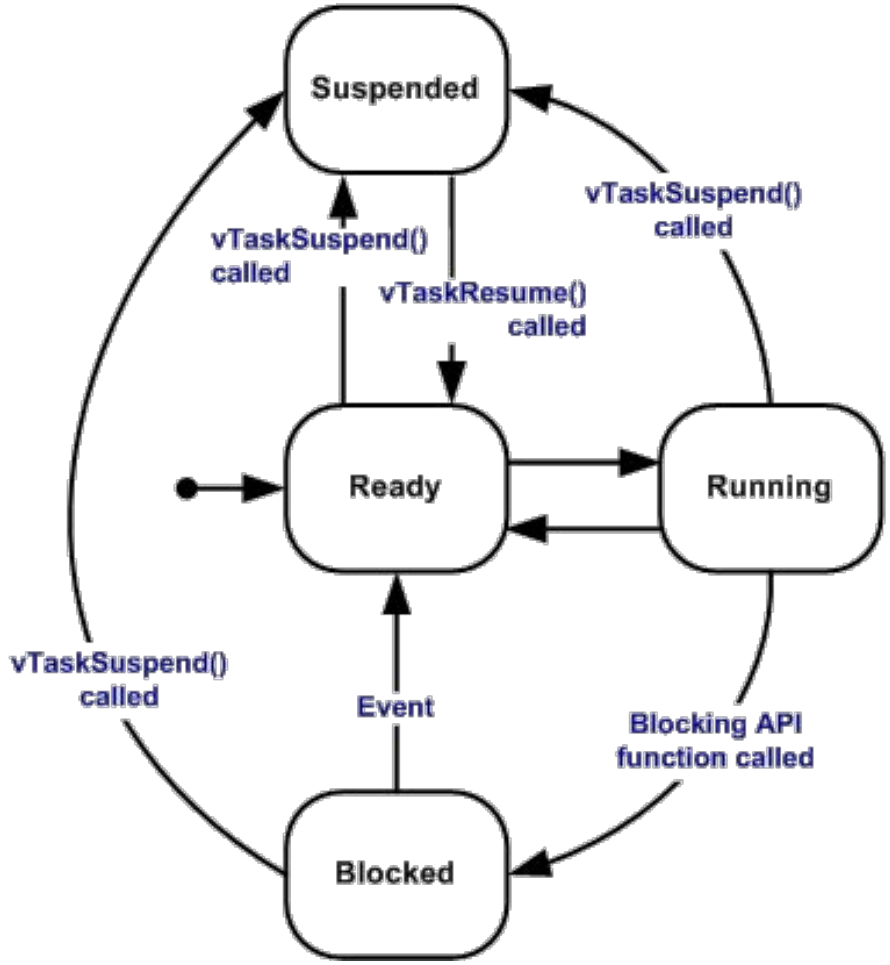
Stany zadań

Nic szokującego:
`READY`, `RUNNING`, `SUSPENDED`, `BLOCKED` (zawsze timeout!)

Oprócz `xTaskResume` jest też `xTaskResumeFromISR` (?!)

(ISR = Interrupt Service Routine)

Czemu potrzebujemy osobnych procedur, których można używać w ISR?



Co robi ISR?

1. Zapisuje kontekst zadania.
2. Jeśli wiele urządzeń podpiętych pod to samo przerwanie, to wywołanie odpowiedniej procedury obsługi.
3. W wyniku obsłużenia przerwania ISR może:
   a. wybudzić wątek o wyższym priorytecie, który czekał na odpowiedź,
   b. zadecydować, że upłynął kwant czasu bieżącego zadania.
4. Odnotowujemy fakt obsłużenia przerwania w sprzęcie: wysyłamy EOI (ang. End of Interrupt).
5. Przywracamy kontekst zadania.

Jak w takim razie zrobić wywłaszczanie?

Przykład: Wybudzanie z kontekstu przerwania

```
TaskHandle_t xTaskHandle;

void vTaskCode(void *pvParameters) {
  for (;;) {
    /* ... Perform some function here. */
    vTaskSuspend(NULL);
    /* The task is now suspended.
     * It won't reach here until the ISR resumes it. */
  }
}

void vAnExampleISR(void) {
  /* Resume the suspended task. */
  BaseType_t xYieldRequired = xTaskResumeFromISR(xTaskHandle);

  /* We should switch context so the ISR returns to a different task. */
  portYIELD_FROM_ISR(xYieldRequired);
}
```

Kontekst przerwania vs. kontekst zadania

1. Czasami mówi się też:
   a. część **a**synchroniczna (**bottom-half** w BSD)
   b. część synchroniczna (**top-half** w BSD)
2. Normalnie wykonuje się kod zadań. Przełączenie zadań odbywa się kooperacyjnie / synchronicznie: `yield`, `block`.
3. Procesor może w dowolnym momencie przerwać kod zadania i wskoczyć do ISR.
4. Na czyim stosie będzie się wykonywać ISR: prywatny? pożyczony od dowolnego wątku?

Jak synchronizować górną i dolną połówkę?

Sychronizacja połówek

1. Jeśli w obydwu połówkach używamy struktur danych (**przykład?**), a przerwanie może przyjść w dowolnym momencie to trzeba coś zrobić → **wyłączanie przerwań**!
2. `taskENTER_CRITICAL` / `taskEXIT_CRITICAL`
   a. do synchronizacji ISR ↔ TASK
   b. obsługa zagnieżdżonych wywołań (np. w TCB: uxCriticalNesting)
3. `vTaskSuspendAll` / `xTaskResumeAll`
   a. do synchronizacji TASK ↔ TASK
   b. również obsługa zagnieżdżonych wywołań
   c. w praktyce wyłącza wywłaszczanie, ale nie przerwania!

Wykorzystywane do implementacji mechanizmów wysokopoziomowych!

Przykład: długa sekcja krytyczna

```
void vTask1(void *pvParameters) {
  for (;;) {
    // At some point the task wants to perform a long operation during which it
    // does not want to get switched out.  It cannot use taskENTER_CRITICAL() nor
    // taskEXIT_CRITICAL() as the length of the operation may cause interrupts
    // to be missed - including the ticks.

    // Prevent the RTOS kernel switch out the task.
    vTaskSuspendAll();

    // Perform the operation here.  There is no need to use critical sections as
    // we have all the microcontroller processing time.  During this time interrupts
    // will still operate and the RTOS kernel tick count will be maintained.
    ...

    // The operation is complete.  Restart the RTOS kernel.
    xTaskResumeAll();
  }
}
```

Start systemu i FreeRTOS

Za wszystkie z poniższych czynności odpowiada programista:

1. Przygotowanie pamięci (zerowanie sekcji BSS)
2. Inicjowanie sprzętu (sterowniki urządzeń)
3. Tworzenie wątków użytkownika
4. Wystartowanie planisty

Pod maską:

1. Uruchomienie czasomierza sprzętowego
2. Utworzenie wątku obsługi czasomierza
   (funkcje wywoływane po upłynięciu terminu)

Idle Task – zadanie jałowe

W systemie musi istnieć zawsze co najmniej jeden wątek w stanie `RUNNING` → procesora nie da się zablokować. Tworzony poprzez wystartowanie planisty.

Możemy podpiąć tam jakąś akcję z `vApplicationIdleHook`, ale nie może używać ona wywołań blokujących.

Idle Task ma najniższy priorytet. **Czy to wystarczy?**

Może wprowadzić procesor w stan uśpienia poprzez wywołanie instrukcji `WFI` (*Wait For Interrupt*) lub podobnej?

Oszczędzanie energii

1. Ważne w serwerach i urządzeniach zasilanych z baterii!
2. Nie ma co robić → wprowadź procesor w stan niskiego zużycia energii. Musimy umieć wybudzić go przerwaniem!
3. Niewielki zysk, jeśli wybudzanie następuje co przerwanie zegarowe (1000Hz?) – może zwiększać zużycie energii!
4. **Tryb jałowy z głębokim snem** (ang. *tickless idle mode*) wyłącza periodyczne przerwanie zegarowe.
5. Po wybudzeniu przerwaniem musimy “naprawić” wartość zegara systemowego → obliczyć ile taktów przespaliśmy.
6. Chcemy prawidłowo obsługiwać terminy – za n taktów wykonaj akcję. Zmiana trybu pracy zegara systemowego!

Przykład: usypianie procesora

```c
/* The parameter is the time (ticks) until the kernel next needs to execute. */
#define portSUPPRESS_TICKS_AND_SLEEP(xIdleTime) vApplicationSleep(xIdleTime)

/* Define the function that is called by portSUPPRESS_TICKS_AND_SLEEP(). */
void vApplicationSleep(TickType_t xExpectedIdleTime) {
  uint32_t ulLowPowerTimeBeforeSleep, ulLowPowerTimeAfterSleep;

  /* Read the current time from a time source that will remain operational
   * while the microcontroller is in a low power state. */
  ulLowPowerTimeBeforeSleep = ulGetExternalTime();

  /* Stop the timer that is generating the tick interrupt. */
  prvStopTickInterruptTimer();

  /* Configure an interrupt to bring the microcontroller out of its
   * low power state at the time the kernel next needs to execute.
   * The interrupt must be generated from a source that is remains
   * operational when the microcontroller is in a low power state. */
  vSetWakeTimeInterrupt(xExpectedIdleTime);

  /* Enter the low power state. */
  prvSleep();
```

```
    /* Determine how long the microcontroller was actually in a low power state
     * for, which will be less than xExpectedIdleTime if the microcontroller was
     * brought out of low power mode by an interrupt other than that configured
     * by the vSetWakeTimeInterrupt() call.  Note that the scheduler is suspended
     * before portSUPPRESS_TICKS_AND_SLEEP() is called, and resumed when
     * portSUPPRESS_TICKS_AND_SLEEP() returns.  Therefore no other tasks will
     * execute until this function completes. */
    ulLowPowerTimeAfterSleep = ulGetExternalTime();

    /* Correct the kernels tick count to account for the time the microcontroller
     * spent in its low power state. */
    vTaskStepTick(ulLowPowerTimeAfterSleep - ulLowPowerTimeBeforeSleep);

    /* Restart the timer that is generating the tick interrupt. */
    prvStartTickInterruptTimer();
}
```

Inter-Task Communication

1. Powiadomienia zadań (Task Notifications)
2. Kolejki (Queues)

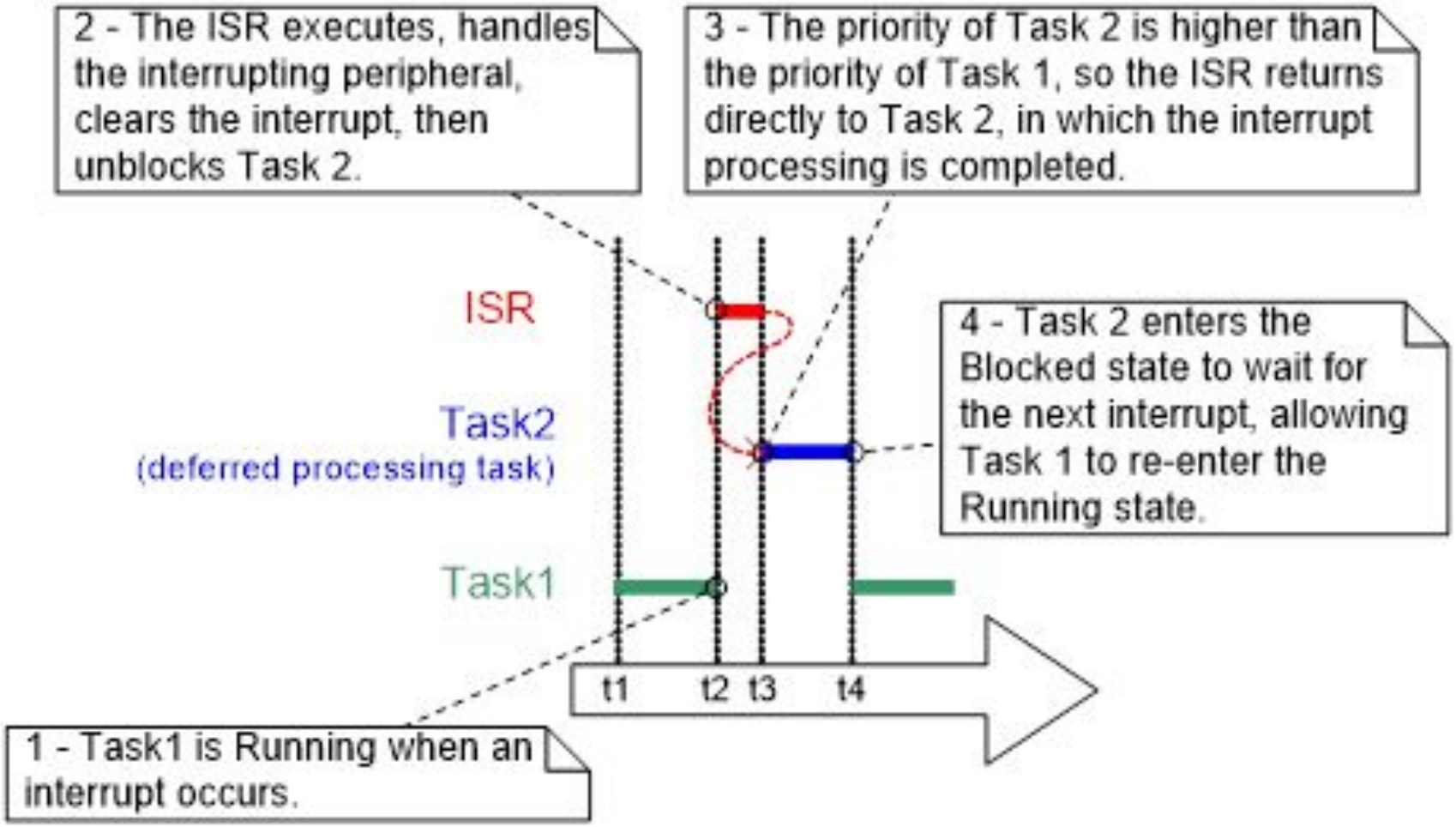
Hybrydy lub pochodne powyższych:

1. Semafory binarne (Binary Semaphores)
2. Semafory zliczające (Counting Semaphores)
3. Muteksy (Mutexes, Recursive Mutexes)
4. Bufory bajtowe / pakietowe (Stream & Message Buffers)
5. Zbiory kolejek (Queue Sets)
6. Grupy zdarzeń (Event Groups)

Odroczona obsługa przerwań: motywacja

1. Ograniczenie opóźnienia obsługi przerwania poprzez minimalizację **fluktuacji** (ang. *jitter*) – odchylenia od średniego czasu obsługi zadań periodycznych – przy założeniu, że obsługa zajmuje dużo czasu.
2. W procedurze obsługi chcemy używać pełnego API z blokowaniem włącznie, a nie tylko `*FromISR`.
3. Czas obsługi zdarzenia jest nieustalony, ale ograniczony z góry.

Przykłady?


2 - The ISR executes, handles the interrupting peripheral, clears the interrupt, then unblocks Task 2.
3 - The priority of Task 2 is higher than the priority of Task 1, so the ISR returns directly to Task 2, in which the interrupt processing is completed.
ISR
Task2
(deferred processing task)
4 - Task 2 enters the Blocked state to wait for the next interrupt, allowing Task 1 to re-enter the Running state.
Task1
t1
t2
t3
t4
1 - Task1 is Running when an interrupt occurs.

Odroczona obsługa przerwań: rozwiązanie

1. Obsługa ISR przekazuje funkcję / dane do wykonania / przetworzenia w kontekście wątku.
2. Możemy opóźnić włączenie przerwania urządzenia do momentu, w którym wątek zakończył obsługę przerwania.
3. Obsługa może być:
   a. **scentralizowana**: w wątku czasomierza systemowego (RTOS daemon) przy pomocy procedury `xTimerPendFunctionCallFromISR`,
   b. **rozproszona**: aplikacja tworzy wątki o wysokich priorytetach i dla każdego przerwania wybiera, w którym wątku należy je obsłużyć.

Jaka jest zaleta drugiego wariantu?

Powiadomienia zadań

1. Komunikacja {ISR, TASK} → TASK.
2. Każde zadanie ma w TCB `ulNotifiedValue` (`int32_t`)
3. Dostępne akcje:
   a. `eSetBits`: ustaw bity (`OR`)
   b. `eIncrement`: zwiększ wartość o 1
   c. `eSetValueWithOverwrite`: nadpisz zadaną wartością
   d. `eSetValueWithoutOverwrite`: j.w. ale tylko, jeśli nie został wybudzony
4. `TaskNotifyGive`[`FromISR`] / `ulTaskNotifyTake`:
   (może zastąpić semafor binarny / zliczający)
5. `xTaskNotify`[`AndQuery`/`Wait`/`Clear`]
   (najogólniejszy interfejs obsługi)

Przykład: czekanie na koniec transmisji

(powiadomienia zadań jako semafor binarny)

```c
/* This is an example of a transmit function in a generic peripheral driver.
 * An RTOS task calls the transmit function, then waits in the Blocked state
 * (so not using an CPU time) until it is notified that the transmission is
 * complete.  The transmission is performed by a DMA, and the DMA
 * end interrupt is used to notify the task. */

/* The task that will be notified when the transmission is complete. */
static TaskHandle_t xTaskToNotify = NULL;

/* The peripheral driver's transmit function. */
void StartTransmission(uint8_t *pcData, size_t xDataLength) {
  /* No transmission is in progress, so xTaskToNotify should be NULL.
   * A mutex can be used to guard access to the peripheral if necessary. */
  configASSERT(xTaskToNotify == NULL);

  /* Store the handle of the calling task. */
  xTaskToNotify = xTaskGetCurrentTaskHandle();

  /* An interrupt will be generated when the transmission is over. */
  vStartTransmit(pcData, xDatalength);
}
```

```c
/* The transmit end interrupt. */
void vTransmitEndISR(void) {
  BaseType_t xHigherPriorityTaskWoken = pdFALSE;

  /* A transmission was in progress, so xTaskToNotify should not be NULL. */
  configASSERT(xTaskToNotify != NULL);

  /* Notify the task that the transmission is complete. */
  vTaskNotifyGiveFromISR(xTaskToNotify, &xHigherPriorityTaskWoken);

  /* There are no transmissions in progress, so no tasks to notify. */
  xTaskToNotify = NULL;

  /* If xHigherPriorityTaskWoken is pdTRUE then a context switch should be
   * performed to ensure the interrupt returns directly to the highest priority
   * task. The macro used for this purpose is dependent on the port. */
  portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

```c
/* The task that initiates the transmission, then enters the Blocked state
 * (so not consuming any CPU time) to wait for it to complete. */
void vAFunctionCalledFromATask(uint8_t ucDataToTransmit, size_t xDataLength) {
  const TickType_t xMaxBlockTime = pdMS_TO_TICKS(200);

  /* Start the transmission by calling the function shown above. */
  StartTransmission(ucDataToTransmit, xDataLength);

  /* Wait to be notified that the transmission is complete.
   * Note the first parameter is pdTRUE, which has the effect of clearing the
   * task's notification value back to 0, making the notification value act
   * like a binary (rather than a counting) semaphore.  */
  uint32_t ulNotificationValue = ulTaskNotifyTake(pdTRUE, xMaxBlockTime);

  if (ulNotificationValue == 1) {
    /* The transmission ended as expected. */
  } else {
    /* The call to ulTaskNotifyTake() timed out. */
  }
}
```

Przykład: odroczona obsługa przerwań

(powiadomienia zadań jako semafor zliczający)

```c
/* An interrupt handler that does not process interrupts directly, but instead
 * defers processing to a high priority RTOS task.  The ISR both unblock the RTOS
 * task and increment the RTOS task's notification value. */
void vANInterruptHandler(void) {
  /* Clear the interrupt. */
  prvClearInterruptSource();

  /* If calling vTaskNotifyGiveFromISR() unblocks the handling task, and the
   * priority of the handling task is higher than the priority of the currently
   * running task, then xHigherPriorityTaskWoken will be set to pdTRUE. */
  BaseType_t xHigherPriorityTaskWoken = pdFALSE;

  /* Unblock the handling task so the task can perform any processing
   * necessitated by the interrupt.  xHandlingTask is the task's handle, which
   * was obtained when the task was created.  vTaskNotifyGiveFromISR() also
   * increments the receiving task's notification value. */
  vTaskNotifyGiveFromISR(xHandlingTask, &xHigherPriorityTaskWoken);

  /* Force a context switch if xHigherPriorityTaskWoken is pdTRUE. */
  portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

```c
/* A task that awaits to be notified that the peripheral needs servicing. */
void vHandlingTask1(void *pvParameters) {
  for (;;) {
    /* The task's notification value is incremented each time the ISR calls
     * vTaskNotifyGiveFromISR(), and decremented each time the RTOS task calls
     * ulTaskNotifyTake() - so in effect holds a count of the number of
     * outstanding interrupts. The notification value is only decremented and
     * not zeroed, and one deferred interrupt event is processed at a time. */
    uint32_t ulNotifiedValue = ulTaskNotifyTake(pdFALSE, xBlockTime);

    if (ulNotifiedValue > 0) {
      /* Perform any processing necessitated by the interrupt. */
      BaseType_t xEvent = xQueryPeripheral();
      if (xEvent != NO_MORE_EVENTS)
        vProcessPeripheralEvent(xEvent);
    } else {
      /* Did not receive a notification within the expected time. */
      vCheckForErrorConditions();
    }
  }
}
```

```c
/* A task that awaits to be notified that the peripheral needs servicing. */
void vHandlingTask2(void *pvParameters) {
  for (;;) {
    /* This time the task's notification value is zeroed out, meaning each
     * outstanding deferred interrupt event must be processed before
     * ulTaskNotifyTake() is called again. */
    uint32_t ulNotifiedValue = ulTaskNotifyTake(pdTRUE, xBlockTime);

    if (ulNotifiedValue == 0) {
      /* Did not receive a notification within the expected time. */
      vCheckForErrorConditions();
    } else {
      /* ulNotifiedValue holds a count of the number of outstanding
       * interrupts. Process each in turn. */
      while (ulNotifiedValue > 0) {
        BaseType_t xEvent = xQueryPeripheral();
        if (xEvent == NO_MORE_EVENTS)
          break;
        vProcessPeripheralEvent(xEvent);
        ulNotifiedValue--;
      }
    }
  }
}
```

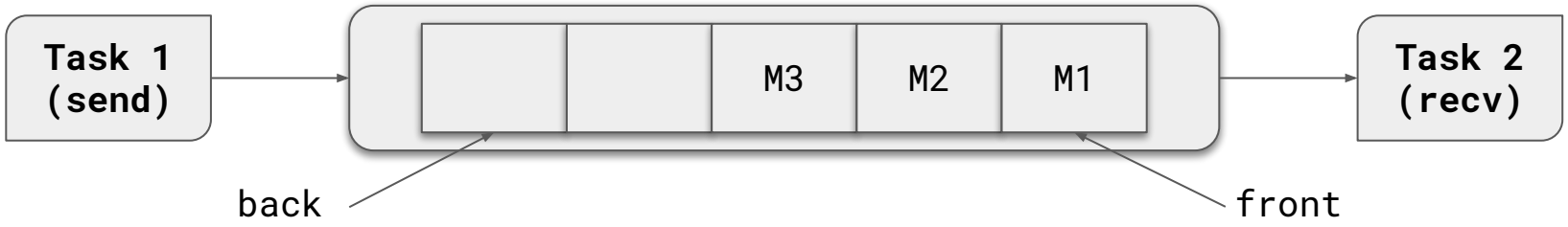
Przykład: odroczona obsługa przerwań

(powiadomienia zadań jako grupy zdarzeń)

```c
/* This example demonstrates a single RTOS task being used to process events
 * that originate from two separate interrupt service routines - a transmit
 * interrupt and a receive interrupt.  Many peripherals will use the same
 * handler for both, in which case the peripheral's interrupt status register
 * can simply be bitwise ORed with the receiving task's notification value. */

/* First bits are defined to represent each interrupt source. */
#define TX_BIT 0x01
#define RX_BIT 0x02

/* The task that will receive notifications from the interrupts. */
static TaskHandle_t xHandlingTask;

/* The implementation of the transmit interrupt service routine. */
void vTxISR(void) {
  prvClearInterrupt();

  BaseType_t xHigherPriorityTaskWoken = pdFALSE;
  /* Notify the task that the transmission is complete by setting
   * the TX_BIT in the task's notification value. */
  xTaskNotifyFromISR(xHandlingTask, TX_BIT, eSetBits,
                     &xHigherPriorityTaskWoken);
  portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

```c
/* Receive interrupt service routine is identical except for TX_BIT. */
void vRxISR(void) { ... }

/* The task that is notified by the interrupt service routines. */
static void prvHandlingTask(void *pvParameter) {
  for (;;) {
    uint32_t ulNotifiedValue;
    BaseType_t xResult =                  /* Wait to be notified of an interrupt. */
        xTaskNotifyWait(pdFALSE,          /* Don't clear bits on entry. */
                        ULONG_MAX,        /* Clear all bits on exit. */
                        &ulNotifiedValue, /* Stores the notified value. */
                        xMaxBlockTime);

    if (xResult == pdPASS) {
      /* A notification was received.  See which bits were set. */
      if (ulNotifiedValue & TX_BIT)
        prvProcessTx();
      if (ulNotifiedValue & RX_BIT)
        prvProcessRx();
    } else {
      /* Did not receive a notification within the expected time. */
      prvCheckForErrors();
    }
  }
}
```

Kolejki



1. Komunikacja {ISR, TASK} ↔ TASK.
2. Zawiera `uxLength` elementów `uxItemSize` bajtowych.
3. Poza ISR wysyłanie `xQueueSend`[`ToBack`,`ToFront`] blokuje jeśli pełna, odbieranie `xQueueReceive` jeśli pusta.
4. W ISR zawsze nieblokujące z sufiksem `FromISR`.
5. Wysyłanie / odbiór przez kopiowanie, ale elementem mogą być wskaźniki (kopia przez referencję).
6. Blokowanie zawsze z terminem.
7. Inne operacje: *Queue[`Overwrite`,`Peek`,`AddToRegistry`].

Przykład: wysyłanie komunikatów z ISR

```
void vBufferISR1(void) {
  /* We have not woken a task at the start of the ISR. */
  BaseType_t xHigherPriorityTaskWoken = pdFALSE;

  /* Loop until the buffer is empty. */
  do {
    /* Obtain a byte from the buffer. */
    char cIn = portINPUT_BYTE(RX_REGISTER_ADDRESS);

    /* Post the byte. */
    xQueueSendToBackFromISR(xRxQueue, &cIn, &xHigherPriorityTaskWoken);
  } while (portINPUT_BYTE(BUFFER_COUNT));

  /* Now the buffer is empty we can switch context if necessary. */
  portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

```c
void vBufferISR2(void) {
  /* We have not woken a task at the start of the ISR. */
  BaseType_t xHigherPriorityTaskWoken = pdFALSE;

  /* Obtain a byte from the buffer. */
  char cIn = portINPUT_BYTE(RX_REGISTER_ADDRESS);

  if (cIn == EMERGENCY_MESSAGE) {
    /* Post the byte to the front of the queue. */
    xQueueSendToFrontFromISR(xRxQueue, &cIn, &xHigherPriorityTaskWoken);
  } else {
    /* Post the byte to the back of the queue. */
    xQueueSendToBackFromISR(xRxQueue, &cIn, &xHigherPriorityTaskWoken);
  }

  /* Did sending to the queue unblock a higher priority task? */
  portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

Przykład: odbieranie komunikatów z ISR

```
QueueHandle_t xQueue;

/* Function to create a queue and post some values. */
void vAFunction(void *pvParameters) {
  /* Create a queue capable of containing 10 characters. */
  xQueue = xQueueCreate(10, sizeof(char));
  configASSERT(xQueue != 0);
  ...

  /* Post some characters that will be used within an ISR. If the queue
   * is full then this task will block for xTicksToWait ticks. */
  char cValueToPost;
  cValueToPost = 'a';
  xQueueSend(xQueue, (void *)&cValueToPost, xTicksToWait);
  cValueToPost = 'b';
  xQueueSend(xQueue, (void *)&cValueToPost, xTicksToWait);
  /* ... keep posting ... may block when the queue becomes full. */
  cValueToPost = 'c';
  xQueueSend(xQueue, (void *)&cValueToPost, xTicksToWait);
}
```

```c
/* ISR that outputs all the characters received on the queue. */
void vISR_Routine(void) {
  BaseType_t xTaskWokenByReceive = pdFALSE;
  char cRxedChar;

  while (xQueueReceiveFromISR(xQueue, (void *)&cRxedChar,
                              &xTaskWokenByReceive)
         && !portINPUT_BYTE(TX_QUEUE_NOTFULL)) {
    /* A character was received.  Output the character now. */
    vOutputCharacter(cRxedChar);

    /* If removing the character from the queue woke the task that was
     * posting onto the queue xTaskWokenByReceive will have been set to
     * pdTRUE.  No matter how many times this loop iterates only one
     * task will be woken. */
  }

  taskYIELD_FROM_ISR(xTaskWokenByReceive);
}
```

Semafory i muteksy

1. Kolejki z rozmiarem elementu równym zero!
2. Struktura kolejki normalnie przechowuje `xQueue`,
   ale `xSemaphore` jeśli semafor / muteks.
3. Rodzaje semaforów:
   a. binarny `xSemaphoreCreateBinary` → rozmiar kolejki 1,
   b. zliczający `xSemaphoreCreateCounting` → ∞.
   c. muteksy zwykłe xSemaphoreCreateMutex,
   d. muteksy rekursywne `xSemaphoreCreateRecursiveMutex`.
4. Zakładanie/zwalnianie blokady xSemaphore[`Take`,`Give`].
5. Rekursywne muteksy `TakeRecursive`,`GiveRecursive`.
6. Muteksy propagują priorytety → trzymają `xMutexHolder`,
   w związku z tym odblokować musi właściciel.

Przykład: uruchom zadanie co 10 taktów

(użycie semafora binarnego)

```
#define TICKS_TO_WAIT 10
#define INFINITE 9999

SemaphoreHandle_t xSemaphore = NULL;

void vATask(void *pvParameters) {
  /* We are using the semaphore for synchronisation so we create a binary
   * semaphore rather than a mutex.  We must make sure that the interrupt
   * does not attempt to use the semaphore before it is created! */
  xSemaphore = xSemaphoreCreateBinary();

  for (;;) {
    /* We want this task to run every 10 ticks of a timer. */
    if (xSemaphoreTake(xSemaphore, INFINITE) == pdTRUE) {
      ...

      /* Return to the top of the loop where we will block on the semaphore
       * until it is time to execute again.  Note when using the semaphore for
       * synchronisation with an ISR in this manner there is no need to 'give'
       * the semaphore back. */
    }
  }
}
```

```
void vTimerISR(void *pvParameters) {
  static unsigned char ucLocalTickCount = 0;
  static signed BaseType_t xHigherPriorityTaskWoken;

  /* A timer tick has occurred. */

  /* ... Do other time functions. */

  /* Is it time for vATask() to run? */
  xHigherPriorityTaskWoken = pdFALSE;
  ucLocalTickCount++;
  if (ucLocalTickCount >= TICKS_TO_WAIT) {
    /* Unblock the task by releasing the semaphore. */
    xSemaphoreGiveFromISR(xSemaphore, &xHigherPriorityTaskWoken);
    /* Reset the count so we release the semaphore again in 10 ticks time. */
    ucLocalTickCount = 0;
  }

  portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

Stream buffers

1. Zakłada, że tylko jeden producent i konsument.
2. Wykorzystuje powiadomienia zadań, więc są niedostępne.
3. Bufor bajtów o skończonym rozmiarze.
4. `xStreamBufferSend`[`FromISR`] czeka na n wolnych bajtów lub do upłynięcia terminu i zapisuje do bufora.
5. `xStreamBufferReceive`[`FromISR`] czeka na n bajtów danych lub do upłynięcia terminu i odczytuje z bufora.
6. W trakcie tworzenia należy podać **poziom wyzwalania** → liczba bajtów, która musi być w buforze zanim konsument zostanie wybudzony.

Gdzie Unix korzysta z podobnego mechanizmu?

```c
void vStreamBufSend(StreamBufferHandle_t xStreamBuffer) {
  uint8_t ucArrayToSend[] = {0, 1, 2, 3};
  char *pcStringToSend = "String to send";
  const TickType_t x100ms = pdMS_TO_TICKS(100);

  /* Send an array to the stream buffer, blocking for a maximum of 100ms to
   * wait for enough space to be available in the stream buffer. */
  size_t xBytesSent = xStreamBufferSend(xStreamBuffer, (void *)ucArrayToSend,
                                        sizeof(ucArrayToSend), x100ms);
  if (xBytesSent != sizeof(ucArrayToSend)) {
    /* The call timed out before there was enough space in the buffer for the
     * data to be written, but it did successfully write xBytesSent bytes. */
  }

  /* Returns immediately if there is not enough space in the buffer. */
  xBytesSent = xStreamBufferSend(xStreamBuffer, (void *)pcStringToSend,
                                 strlen(pcStringToSend), 0);
  if (xBytesSent != strlen(pcStringToSend)) {
    /* The entire string could not be added to the stream buffer because
     * there was not enough free space in the buffer, but xBytesSent bytes
     * were sent.  Could try again to send the remaining bytes. */
  }
}
```

```c
void vStreamBufRecv(StreamBuffer_t xStreamBuffer) {
  uint8_t ucRxData[20];
  const TickType_t xBlockTime = pdMS_TO_TICKS(20);

  /* Receive up to another sizeof(ucRxData) bytes from the stream buffer.
   * Wait in the BLOCKED state for a maximum of 100ms for the full
   * sizeof(ucRxData) number of bytes to be available. */
  size_t xReceivedBytes = xStreamBufferReceive(xStreamBuffer, (void *)ucRxData,
                                               sizeof(ucRxData), xBlockTime);

  if (xReceivedBytes > 0) {
    /* A ucRxData contains another xRecievedBytes bytes of data,
     * which can be processed here.... */
  }
}
```

Message buffers

1. Nadbudowane nad *stream buffers*.
2. Bufor bajtów podzielony na pakiety, każdy zapis dodaje 4-bajtowe słowo wyznaczające rozmiar pakietu.
3. `xMessageBufferSend`[`FromISR`] zapisuje cały pakiet, jeśli nie było miejsca i upłynął termin → 0.
4. `xMessageBufferReceive`[`FromISR`] odbiera cały pakiet, jeśli nie może wczytać w całości lub upłynął termin → 0.

Gdzie Unix korzysta z podobnego mechanizmu?

Pytania?