Projektowanie Systemów Wbudowanych

Podstawowe informacje o systemie µC/OS-II

Autorzy: mgr inż. Dominik Bąk i mgr inż. Leszek Ciopiński

Spis treści

System operacyjny czasu rzeczywistego....1
Struktura jądra (Leszek Ciopiński)....3
Porty (Dominik Bąk i Leszek Ciopiński)....6
Zarządzanie procesami (Dominik Bąk)....7
Tworzenie procesów OSTaskCreate()....7
Usuwanie procesu OSTaskDel()....8
Usuwanie procesu OSTaskDelReq () 8
Zmiana priorytetu procesów OSTaskChangePrio () 9
Blokowanie procesów OSTaskSuspend () 9
Wznawianie procesów OSTaskResume () 10
Semafory (Leszek Ciopiński)....10
Muteksy (Leszek Ciopiński)....12
Skrzynki wiadomości (Dominik Bąk)....14
Tworzenie skrzynki pocztowej, OSMboxCreate() 15
Usuwanie skrzynek pocztowych OSMboxDel() 15
Oczekiwanie na wiadomości od skrzynek OSMboxPend() 16
Wysyłanie wiadomości do skrzynek OSMboxPost() 16
Otrzymywanie wiadomości bez oczekiwania OSMboxAccept () 16
Kolejki komunikatów (Dominik Bąk)....17
Zarządzanie pamięcią (Dominik Bąk) 20
Tworzenie partycji pamięci OSMemCreate()....21
Pobieranie bloków pamięci z partycji OSMemGet()....22
Zwracanie bloków do partycji pamięci OSMemPut()....22
Uzyskiwanie informacji na temat partycji pamięci OSMemQuery()....22

System operacyjny czasu rzeczywistego

System operacyjny czasu rzeczywistego (ang. *Real-Time Operating System - RTOS*) to komputerowy system operacyjny, który został opracowany tak, by spełnić wymagania narzucone na czas wykonywania zadanych operacji. Systemy takie stosuje się jako elementy komputerowych systemów sterowania pracujących w reżimie czasu rzeczywistego - system czasu rzeczywistego.

Ogólnie można przyjąć założenie, że zadaniem systemu operacyjnego czasu rzeczywistego oraz oprogramowania pracującego pod jego kontrolą i całego sterownika komputerowego jest wypracowywanie odpowiedzi (np. sygnałów sterujących kontrolowanym obiektem) na skutek wystąpienia pewnych zdarzeń (zmianie sygnałów z czujników sterownika). Biorąc to pod uwagę, podstawowym wymaganiem dla systemu operacyjnego czasu rzeczywistego jest określenie najgorszego (najdłuższego) czasu, po jakim urządzenie komputerowe wypracuje odpowiedź po wystąpieniu zdarzenia. Ze względu na to kryterium, systemy operacyjne czasu rzeczywistego dzielą się na dwa rodzaje:

- Twarde - takie, dla których znany jest najgorszy (najdłuższy) czas odpowiedzi, oraz wiadomo jest, że nie zostanie on przekroczony.

- Miękkie - takie, które starają się odpowiedzieć najszybciej jak to możliwe, ale nie wiadomo jest, jaki może być najgorszy czas odpowiedzi.

Nietrywialnym problemem w tego typu systemach operacyjnych jest algorytm szeregowania oraz podziału czasu. W systemie operacyjnym czasu rzeczywistego trzeba określić, któremu z procesów należy przydzielić procesor oraz na jak długi czas, aby wszystkie wykonywane procesy spełniały zdefiniowane dla nich ograniczenia czasowe.

Pojawienie się systemów operacyjnych tego typu wiąże się z m.in. zapotrzebowaniem techniki wojskowej na precyzyjne w czasie sterowanie rakietami. Obecnie systemy operacyjne tego typu są wykorzystywane powszechnie w przemyśle cywilnym, sterują również urządzeniami takimi jak na przykład: centrale telefoniczne, marsjańskie lądowniki NASA oraz samochodowy ABS.

System operacyjny czasu rzeczywistego [online]. Wikipedia : wolna encyklopedia, 2013-03-12 21:04Z [dostęp: 2013-04-13 15:26Z]. Dostępny w Internecie: //pl.wikipedia.org/w/index.php?title=System_operacyjny_czasu_rzeczywistego&oldid=35107375

MicroC/OS-II – prosty, niewielki system operacyjny czasu rzeczywistego.

System umożliwia uruchomienie do 64 procesów użytkownika, synchronizację i komunikację. Nie posiada graficznej powłoki, systemu plików ani obsługi sieci. Niewielkie rozmiary pozwalają uruchomić go nawet na procesorach 8 bit (np. Intel 8051) z niewielką pamięcią programu (dla 8051 wystarczy 14kB).

Największą zaletą systemu, oprócz małego rozmiaru kodu wykonywalnego, są certyfikaty, umożliwiające zastosowanie go w aparaturze medycznej, czy awionice wedle najsurowszych norm bezpieczeństwa (DO-178B Level A oraz EUROCAE ED-12B).

MicroC/OS-II [online]. Wikipedia : wolna encyklopedia, 2013-03-15 18:50Z [dostęp: 2013-04-13 15:30Z]. Dostępny w Internecie: //pl.wikipedia.org/w/index.php?title=MicroC/OS-II&oldid=35469768

System operacyjny uC/OS-II jest systemem typu twardego, gdyż w jego API dostępne są funkcje, których czas wykonywania możemy ograniczyć do zadanej wartości. Należy tu jednak zauważyć, że zadeklarowanie zbyt krótkiego czasu może doprowadzić do zakończenia działania funkcji przed uzyskaniem prawidłowego wyniku. Uzyskuje się wówczas informację, o przekroczeniu zadanego czasu. Istnieje jednak możliwość wywołania danej funkcji bez nakładania rygoru czasu na jej wykonanie.

Struktura jądra (Leszek Ciopiński)

Sekcją krytyczną nazywamy taki fragment kodu, w którym dokonywane są operacje na zmiennych współdzielonych przez różne procesy. Istotne jest wobec tego, aby kilka procesów nie wykonywało swojej sekcji krytycznej w tym samym czasie, gdyż mogło by to doprowadzić do przeplotu. Podobnie jak wszystkie systemy czasu rzeczywistego, system µC/OS-II wymaga wyłączenia przerwań podczas wchodzenia do sekcji krytycznej oraz do ich przywrócenia podczas jej opuszczania. Zapobiega to nieprzewidzianemu przerywaniu wykonywania kodu sekcji krytycznej przez inny proces lub obsługę przerwania. Realizowane jest to poprzez użycie makr `OS_ENTER_CRITICAL()` oraz `OS_EXIT_CRITICAL()`. Ponieważ ich implementacja zależy od procesora zadeklarowane są w pliku nagłówkowym OS_CPU.H. Makra te zawsze muszą występować w parze, gdzie pierwsze z nich rozpoczyna, a drugie kończy sekcję krytyczną.

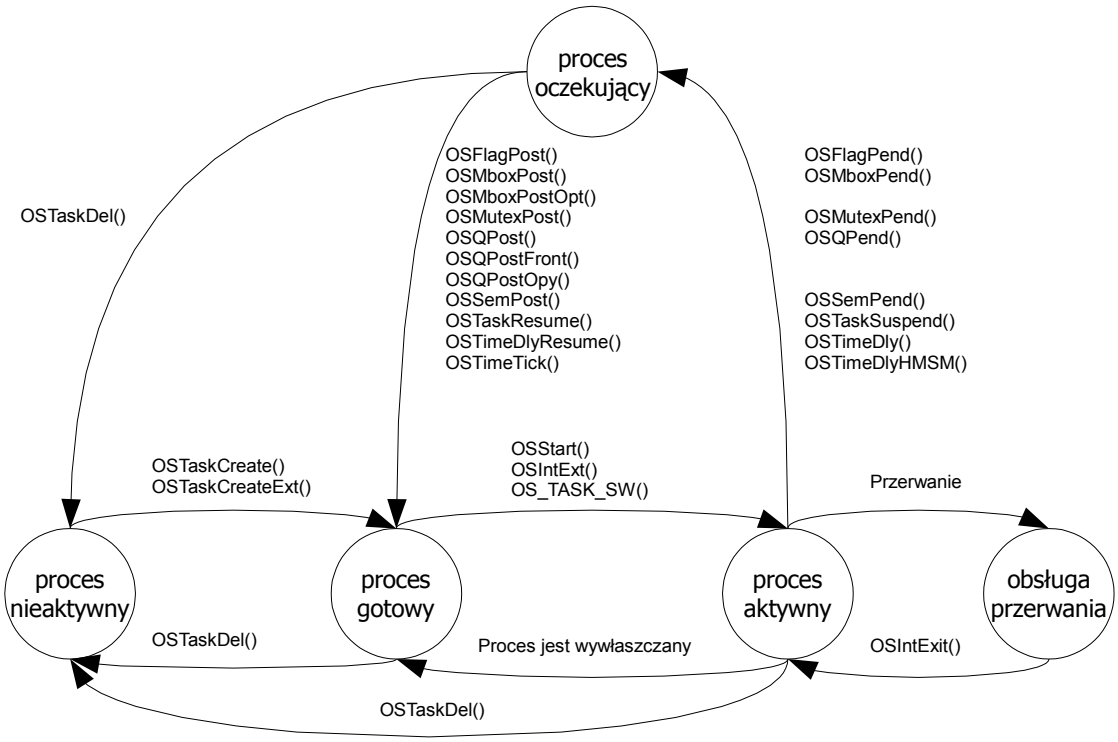
Procesem nazywamy zbiór zadań procesora razem z zapamiętanym jego stanem oraz stosem pamięci do przechowywania zmiennych lokalnych. Proces musi wykonywać się w niekończącej się pętli. System µC/OS-II wymaga aby proces używał w pętli przynajmniej jednej z następujących funkcji: `OSFlagPend()`, `OSMboxPend()`, `OSMutexPend()`, `OSQPend()`, `OSSemPend()`, `OSTimeDly()`, `OSTimeDlyHMSM()`, `OSTaskDel(OS_PRIO_SELF)`, `OSTaskSuspend(OS_PRIO_SELF)`. Proces należy napisać w formie funkcji języka C o prototypie: `void NazwaProcesu(void *dane)`. Do jego utworzenia można użyć funkcji: `OSTaskCreate()` lub `OSTaskCreateExt()`. Obie funkcje jako parametry przyjmują: wskaźnik na funkcję, która ma być wykonywana jako proces; wskaźnik na parametr, który ma być przekazany dla procesu; wskaźnik na zaalokowaną pamięć stosu oraz priorytet procesu. Funkcja `OSTaskCreateExt()` umożliwia podanie kilku dodatkowych parametrów. Priorytet jest wartością całkowitoliczbową z przedziału od 0 (najwyższy priorytet) do maksymalnie 63 (najniższy priorytet). Programista ma jednak możliwość ograniczenia najwyższej wartości priorytetu (najniższego priorytetu), dlatego jest ona zadeklarowana jako stała `OS_LOWEST_PRIO`. Ponadto w systemie µC/OS-II niedozwolone jest, aby istniały co najmniej dwa procesy o tym samym priorytecie. Przy próbie utworzenia kolejnych procesów o tym samym priorytecie, system odmówi wykonania polecenia informując, w komunikacie o błędzie, o zaistniałym konflikcie priorytetów. Do usuwania procesów z systemu służy funkcja `OSTaskDel()`, która jako parametr przyjmuje priorytet procesu do usunięcia. Każdy proces przechowuje wartość swojego priorytetu w stałej. Jeśli więc proces ma się sam zakończyć, należy użyć polecenia `OSTaskDel(OS_PRIO_SELF)`, zamiast `return` lub `break` które mogłyby doprowadzić do wyjścia z niekończącej się pętli procesu.

W systemie operacyjnym proces może znajdować się w jednym z pięciu stanów. Diagram przejść pomiędzy nimi został przedstawiony na rysunku 1. Stanami tymi są:

- proces nieaktywny – istnieje skompilowana wersja procesu w pamięci RAM lub ROM, jednak jest ona niedostępna dla systemu µC/OS-II. Proces w tym stanie nie posiada przypisanego ani stosu ani priorytetu.
- proces gotowy – stan w którym proces nie oczekuje na jakiekolwiek zdarzenie (np. zwolnienie semafora) i może ubiegać się o zajęcie procesora.
- proces aktywny – w systemie µC/OS-II w danym czasie tylko jeden proces może znajdować się w tym stanie. Kod tego procesu wykonywany jest na procesorze. Jako proces aktywny wybierany jest zawsze ten, który ma najwyższy priorytet i jest w stanie „proces gotowy” lub „proces aktywny”. Proces wykonuje się dopóty, dopóki nie zakończy swojego wykonywania, nie przejdzie w stan „procesu oczekującego” lub „obsługi przerwania”, lub nie pojawi się „proces gotowy” o priorytecie wyższym od „procesu aktywnego”.
- obsługa przerwania – proces może znaleźć się w tym stanie na skutek przerwania. Pozbawiany

jest on procesora na rzecz procedury obsługi przerwania. Ponieważ na skutek jej wykonywania mógł pojawić się „proces gotowy” o priorytecie wyższym niż ostatnio wykonywany na procesorze, po zakończeniu obsługi przerwania µC/OS-II sprawdza listę „procesów gotowych”. Jeśli najwyższy priorytet z tej grupy jest mniejszy od priorytetu procesu ostatnio wykonywanego, odzyskuje on procesor. W przeciwnym przypadku przechodzi on w stan „proces gotowy”, a nowym „procesem aktywnym” zostaje proces z grupy „procesów gotowych” o najwyższym priorytecie.

- proces oczekujący – to stan w którym proces, nim wznowi swoje wykonywanie, musi poczekać na jakieś określone zdarzenie. Może nim być zmiana stanu semafora lub upłynięcie pewnego okresu czasu.



Rysunek 1: Diagram zmiany stanów procesu. [1]

W chwili tworzenia procesu przypisywany jest do niego *blok kontrolny procesu*, `OS_TCB`. Jest to struktura danych, która jest wykorzystywana przez µC/OS-II do zapamiętywania stanu procesu w chwili jego wywłaszczenia. Pozwala ona na wznowienie wykonywania procesu dokładnie w miejscu, w którym został on wstrzymany. Wszystkie struktury `OS_TCB` znajdują się w pamięci RAM. Warto zauważyć, że obie funkcje systemowe służące do tworzenia procesu mogą zgłaszać w komunikacie o błędzie brak wolnych struktur `OS_TCB`. Rozmiar tych struktur nie jest stały i zależy od konfiguracji systemu µC/OS-II. Ilość struktur rezydujących w pamięci można zredukować poprzez zmniejszenie parametru `OS_MAX_TASKS` (maksymalna ilość procesów w systemie) w pliku konfiguracyjnym `OS_CFG.H`. System operacyjny zwiększa jednak ilość alokowanych struktur `OS_TCB` o wartość parametru `OS_N_SYS_TASKS` (`uCOS_II.h`). Dodatkowe struktury wykorzystywane są na potrzeby własne systemu, takie jak obsługa procesu bezczynności lub procesu statystycznego (jeśli jest używany).

W systemie µC/OS-II zmiana kontekstu realizowana jest jako przerwanie procesora, co umożliwia zapisanie jego rejestrów. Ponieważ sposób symulowania przerwania zależy od

używanego procesora, konieczne jest zaimplementowanie makra `OS_TASK_SW()`.

Działanie procesu planisty może zostać zawieszone poprzez użycie funkcji `OSSchedLock()`. Zabezpiecza to proces wywołujący tą funkcję przed wywłaszczeniem, nawet jeśli pojawi się proces w stanie „gotowy" o wyższym priorytecie niż proces aktywny. Stan ten utrzymuje się dopóki proces aktywny nie wywoła funkcji `OSSchedUnlock()`, odblokowującej planistę. Zmienna `OSLockNesting` zlicza ile razy została wywołana funkcja `OSSchedLock()`, jednak tylko do wartości 255. Dlatego też µC/OS-II nie pozwala na zagnieżdżanie funkcji blokującej planistę więcej niż 255 razy. `OSSchedUnlock()` działa odwrotnie – dekrementuje wartość zmiennej `OSLockNesting`. Osiągnięcie przez nią wartości 0 powoduje wznowienie procesu planisty. Konieczne jest więc używanie obu funkcji w parze. Użycie tych funkcji pozostaje jednak bez wpływu na przerwania. Przez cały czas pozostają one rozpoznawane i obsługiwane. Jest to też jedna z różnic w porównaniu z makrami obsługi sekcji krytycznej (`OS_ENTER_CRITICAL()` i `OS_EXIT_CRITICAL()`) wyłączającymi również przerwania.

Po wywołaniu funkcji `OSSchedLock()` aplikacja nie może wywoływać żadnych funkcji systemowych, które doprowadziły by do przejścia procesu aktywnego do innego stanu. Funkcjami tymi są: `OSFlagPend()`, `OSMboxPend()`, `OSMutexPend()`, `OSQPend()`, `OSSemPend()`, `OSTaskSuspend(OS_PRIO_SELF)`, `OSTimeDly()` i `OSTimeDlyHMSM()`. Blokowanie procesu planisty w praktyce może okazać się przydatne, gdy proces o niskim priorytecie wysyła wiadomości do różnych skrzynek, kolejek lub semaforów i oczekuje się, żeby nie został wywłaszczony, dopóki nie zakończy wysyłania informacji do wszystkich skrzynek, kolejek lub semaforów.

W systemie µC/OS-II zawsze musi istnieć proces gotowy do wykonywania, dlatego bezpośrednio po uruchomieniu systemu, tworzy on automatycznie proces zwany procesem bezczynności. Uzyskuje on najniższy z dopuszczalnych priorytetów, czyli wartość określoną poprzez stałą `OS_LOWEST_PRIO`. Proces ten nigdy nie może być usunięty przez uruchamianą aplikację. Jego podstawowym zadaniem jest istnieć w systemie. Wykonuje on jednak dwie istotne rzeczy. Po pierwsze inkrementuje zmienną `OSIdleCtr` wykorzystywaną przez proces statystyczny. Drugim zadaniem jest wywołanie funkcji `OSTaskIdleHook()`. Służy ona do umożliwienia programiście napisania własnego fragmentu tego procesu. Jedynym ograniczeniem jest tu, tak jak w przypadku funkcji blokującej proces planisty, zakaz używania funkcji systemowych, które wymagają oczekiwania na jakieś zdarzenie i wywołują przejście procesu w stan oczekiwania. Możliwość napisania funkcji `OSTaskIdleHook()` może okazać się przydatna do wysłania procesorowi rozkazu przejścia w tryb oszczędzania energii, w szczególności w układach zasilanych bateryjne.

System µC/OS-II wymaga, aby instrukcje obsługi przerwania (ang. Interrupt Service Routine – ISR) napisane były w asemblerze. Środowisko Altera Nios II IDE umożliwia jeszcze inny sposób pisania funkcji obsługi przerwań, który był już wykorzystywany w poprzednich instrukcjach laboratoryjnych.

µC/OS-II umożliwia programową obsługę przerwania zegarowego. Sygnał zegarowy generuje przerwanie, które może być obsłużone tylko wtedy, gdy µC/OS-II uruchomi już tryb wielozadaniowości. Dlatego nie wolno uaktywniać obsługi przerwania zegarowego do czasu, gdy wywołane zostanie polecenie `OSStart()`. Uaktywnienia takiego należy dokonać w pierwszym procesie, który zostanie uruchomiony po wywołaniu wspomnianej funkcji. Nie zastosowanie się do tego wymogu może doprowadzić do błędnego zakończenia się aplikacji.

Uruchamianie systemu µC/OS-II odbywa się w dwóch etapach – inicjalizacja i start systemu. Bezpośrednio po uruchomieniu aplikacji, gdy rozpoczyna się wykonywanie instrukcji funkcji `main()`, nie mamy jeszcze dostępu do żadnej funkcji systemu operacyjnego. Dostęp ten uzyskujemy wywołując funkcję `OSInit()`. Rozpoczyna ona proces inicjalizacji systemu. Tworzony jest wówczas proces bezczynności oraz, jeśli ustawione są stałe `OS_TASK_STAT_EN` i

OS_TASK_CREATE_EXT_EN na 1, również proces statystyczny. Ponadto tworzone są pola i struktury zgodnie z utworzoną przez programistę konfiguracją zawartą w pliku OS_CFG.H.

Po wykonaniu inicjalizacji systemu należy wykonać inicjalizację opracowywanej aplikacji. Wymagane jest, aby na tym etapie utworzyć przynajmniej jeden proces przy pomocy funkcji OSTaskCreate(). Dopiero teraz możemy uruchomić system µC/OS-II wywołując funkcję OSStart(). Należy zauważyć, że system nigdy nie zwraca sterowania z tej funkcji. Oznacza to, że kod pisany za nią nigdy nie zostanie wykonany. Jeśli więc istnieje konieczność ustawienia jakichś parametrów po wywołaniu tej funkcji, konieczne jest zrobienie tego w jednym z istniejących procesów. Można też rozważyć wcześniejsze utworzenie specjalnego procesu, którego celem będzie jedynie dokonanie wstępnej konfiguracji i który zakończy się funkcją OSTaskDel(OS_PRIO_SELF) zaraz po wykonaniu swojego zadania.

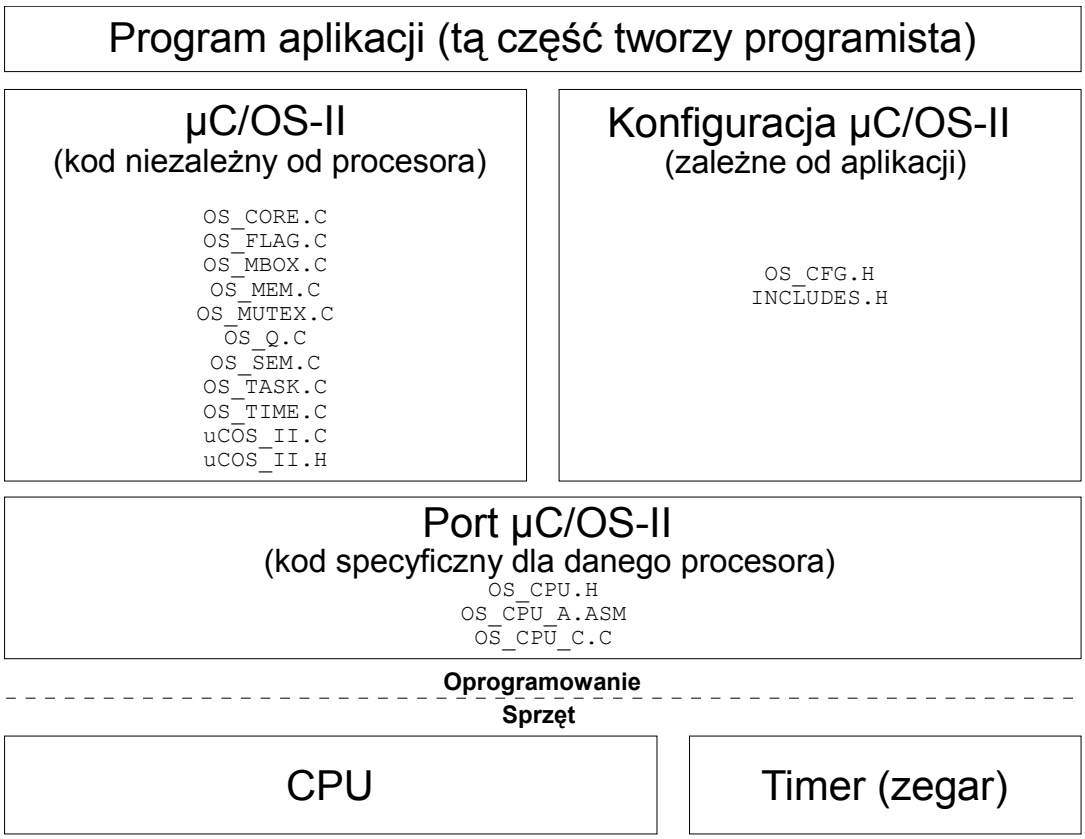
Porty (Dominik Bąk i Leszek Ciopiński)

Większość kodu systemu µC/OS-II napisana jest w języku C, co ułatwia jego przenoszenie pomiędzy różnymi architekturami. Mimo to istnieje spora grupa operacji, która musi być wykonywana w sposób charakterystyczny dla danego procesora. W opisywanym systemie operacyjnym wszystkie operacje tego typu zostały zebrane w trzech plikach: OS_CPU.H, OS_CPU_C.C i (opcjonalnie) OS_CPU_ASM.ASM. Wszystkie zaś muszą znajdować się w folderze \SOFTWARE\uCOS-II na dysku twardym komputera. Pliki te nazywamy portem. Stanowią one bowiem rodzaj spoiwa pomiędzy konkretną architekturą procesora, a resztą systemu.

Aby możliwe było napisanie portu dla danego procesora musi on spełniać następujące wymagania:

1. Dla danego procesora musi istnieć kompilator języka C, który może generować kod uruchamiany współbieżnie.
2. Procesor obsługuje przerwania i jest w stanie dostarczać przerwania występujące w regularnym, ściśle określonym odstępie czasu (najczęściej pomiędzy 10 a 100Hz).
3. Przerwania mogą być wyłączane i załączane z poziomu języka C.
4. Procesor posiada stos sprzętowy na którym może pomieścić dużą ilość danych, możliwe, że wiele kilobajtów.
5. Procesor musi posiadać instrukcję do zapisu i odczytu wskaźnika stosu oraz rejestrów albo z wykorzystaniem stosu sprzętowego albo z wykorzystaniem pamięci.

Jeżeli procesor nie spełnia wszystkich wymagań, tak jak np. Motorola z serii 6805, która nie spełnia warunków 4. i 5. system µC/OS-II nie może być na nim uruchamiany.



Rysunek 2: Architektura sprzętowo-programowa µC/OS-II.

Dla Altera Nios II istnieje gotowy port µC/OS-II, który jest częścią środowiska Altera Nios II IDE. Więcej informacji na ten temat portów znaleźć w publikacji [1] w rozdziałach 13., 14. i 15.

Zarządzanie procesami (Dominik Bąk)

Jeden z poprzednich rozdziałów wspominał o tym, że procesy wykonują się w niekończącej się pętli albo są usuwane po wykonaniu się przez wywołanie odpowiedniej funkcji systemowej. Rozdział ten opisuje funkcje systemowe znajdujące się w pliku `OS_TASK.C`, które pozwalają opracowywanej aplikacji na utworzenie, usunięcie, zmianę priorytetu, wstrzymanie i wznowienie wykonywania procesu oraz uzyskanie informacji o nich. µC/OS II potrafi obsłużyć do 64 procesów, jednakże cztery najwyższe i cztery najniższe priorytety należy zarezerwować dla procesów, które mogą zostać utworzone przez sam system operacyjny. Obostrzenia te uwzględniają również przyszłe wymagania systemu µC/OS-II. Obecnie, jedynymi priorytetami jakie są używane to `OS_LOWEST_PRIO i OS_LOWEST_PRIO-1`. W związku z powyższym, projektowana aplikacja nie może wykorzystywać więcej niż 56 procesów jednocześnie. Należy pamiętać o tym, że niższa wartość priorytetu to wyższy priorytet zadania. W aktualnej wersji µC/OS II, numer priorytetu służy również jako identyfikator zadania. Poniżej przedstawione zostały funkcje systemowe pomagające w zarządzaniu procesami.

Tworzenie procesów OSTaskCreate()

```
INT8U OSTaskCreate (void (*task) (void * PD), void * pdata, OS_STK
```

```
* ptos, INT8U prio);
```

Proces tworzymy poprzez wywołanie jednej z dwóch funkcji: `OSTaskCreate()` lub `OSTaskCreateExt()`, które jako parametry przyjmują, między innymi, adres kodu i parametry początkowe procesu. Funkcja `OSTaskCreate()` jest kompatybilna wstecz z µC/OS II, a `OSTaskCreateExt()` jest wersją rozszerzoną wobec `OSTaskCreate()`, mogącą przyjmować dodatkowe parametry. Procesy mogą być tworzone zarówno przed rozpoczęciem pracy wielozadaniowej, jak i po, uruchamiane przez inne procesy, pod warunkiem, że co najmniej jeden proces jest tworzony przed uruchomieniem trybu wielozadaniowego [tzn. zanim nastąpi wywołanie `OSStart ()`]. Dodatkowo, należy pamiętać, że procesy nie mogą być tworzone przez procedurę obsługi przerwania (ISR). Do parametrów funkcji należą: `task` wskaźnik na kod procesu, `pdata` służący do przekazywania parametrów początkowych do procesu, gdy jest tworzony. `ptos` – wskaźnik na szczyt stosu procesu. Stos jest używany do przechowywania zmiennych lokalnych, parametrów funkcji, adresów powrotu i rejestrów CPU podczas przerwań. Ilość pamięci jaką należy zarezerwować na stos procesu jest uzależniona zarówno od przewidywanych wymagań procesu, jak również przewidywanej ilości zagnieżdżonych przerwań. Określając rozmiar stosu należy wziąć pod uwagę ilość pamięci niezbędną do przechowywania zmiennych lokalnych, zmiennych należących do wywoływanych w procesie funkcji oraz pamięci przeznaczonej dla procedur obsługi przerwania. Jeśli stała `OS_STK_GROWTH` ustawiona jest na 1 to stos rośnie w dół. Wtedy też `ptos` musi wskazywać na szczyt stosu. Jeśli `OS_STK_GROWTH` jest ustawiony na 0, stos jest budowany w przeciwnym kierunku. `prio` oznacza priorytet zadania. Wartość ta musi być unikatowa w skali systemu, ponieważ jest równocześnie identyfikatorem procesu. Im niższa wartość przypisana temu parametrowi, tym wyższy priorytet. Funkcja `OSTaskCreate()` zwraca następujące kody błędów:

- `OS_NO_ERR` – jeśli funkcja wykonała się poprawnie.
- `OS_PRIO_EXIST` – jeżeli użyto istniejącego już priorytetu.
- `OS_PRIO_INVALID` – jeżeli `prio` ma wyższą wartość niż `OS_LOWEST_PRIO`.
- `OS_NO_MORE_TCB` – jeśli µC/OS II nie ma wolnych struktur `OS_TCB`.

Usuwanie procesu OSTaskDel()

```
NT8U OSTaskDel (INT8U prio);
```

`OSTaskDel()` pozwala aplikacji na usunięcie procesu o priorytecie określonym w parametrze `prio`. Przekazanie funkcji wartość stałej `OS_PRIO_SELF`, nakazuje systemowi usunięcie procesu wywołującego tą funkcję. `OSTaskDel()` zwraca następujące kody błędów:

1) `OS_NO_ERR` - jeśli proces nie zostanie usunięty;
2) `OS_TASK_DEL_IDLE` – próbowano usunąć proces bezczynności
3) `OS_TASK_DEL_ERR` – proces do usunięcia nie istnieje
4) `OS_PRIO_INVALID` - określony proces ma wyższą wartość priorytetu niż `OS_LOWEST_PRIO`
5) `OS_TASK_DEL_ISR` - próba usunięcia zadania z procedury obsługi przerwania (ISR)

Należy zachować ostrożność przy usuwaniu procesów, które współdzielą zasoby. Dlatego zalecane jest użycie funkcji `OSTaskDelReq()`, która jest znacznie bezpieczniejsza.

Usuwanie procesu OSTaskDelReq ()

```
INT8U OSTaskDelReq (INT8U prio);
```

`OSTaskDelReq()` to kolejna funkcja służąca do usuwania procesów. Zasadniczo

wykorzystuje się ją przy usuwaniu zadań, które mogą współdzielić zasoby (np. zadania mogą mieć wspólny semafor) z innymi procesami. W tym przypadku, dopóki zasób nie zostanie zwolniony, procesu nie można usunąć. Funkcja `OSTaskDelReq()` oznacza proces jako proces do usunięcia. Polega to na przesłaniu usuwanemu procesowi informacji z prośbą o zakończenie działania. Kiedy usuwany proces zwolni współdzielony zasób, powinien wywołać funkcję `OSTaskDelReq(OS_PRIO_SELF)`. Jeśli zwrócona wartość jest taka sama, jak wartość stałej `OS_TASK_DEL_REQ`, oznacza to, że proces został poproszony o zakończenie swojego działania. Powinien on wówczas wywołać funkcję `OSTaskDel(OS_PRIO_SELF)`. Ponieważ funkcja `OSTaskDelReq()` nie powoduje automatycznego zakończenia procesu, może być wywoływana wielokrotnie w celu sprawdzenia, czy usunięcie procesu doszło do skutku. `OSTaskDelReq()` zwraca następujące kody błędów:

- `OS_NO_ERR` – jeśli proces został usunięty.
- `OS_TASK_NOT_EXIST` – jeśli proces nie istnieje. Zgłaszający proces może monitorować kod błędu, aby sprawdzić, czy zadanie zostało faktycznie usunięte.
- `OS_TASK_DEL_IDLE` – próba usunięcia procesu bezczynności (wykonywanie tej czynności jest zabronione).
- `OS_PRIO_INVALID` – przekazany funkcji parametr ma wartość większą niż `OS_LOWEST_PRIO` lub `OS_PRIO_SELF` ma nie określoną wartość.
- `OS_TASK_DEL_REQ` – jeśli pojawi się prośba o usunięcie procesu.

Zmiana priorytetu procesów OSTaskChangePrio ()

```
INT8U OSTaskChangePrio (INT8U oldprio, INT8U newprio);
```

Funkcja `OSTaskChangePrio()` wykorzystywana jest do zmiany priorytetu procesu. Argumentami tej funkcji są `oldprio` i `newprio`. Pierwszy z parametrów to numer priorytetu do zmiany, drugi zaś jest nowym priorytetem procesu. `OSTaskChangePrio()` zwraca następujące kody błędów:

- `OS_NO_ERR` jeżeli priorytet procesu został zmieniony.
- `OS_PRIO_INVALID` jeżeli stary i nowy priorytet są sobie równe lub przekroczony został `OS_LOWEST_PRIO`.
- `OS_PRIO_EXIST` jeśli nowy priorytet jest już używany.
- `OS_PRIO_ERR` jeśli proces określony przez `oldprio` nie istnieje.

Blokowanie procesów OSTaskSuspend ()

```
INT8U OSTaskSuspend(INT8U prio);
```

`OSTaskSuspend()` zawiesza wykonanie procesu o podanym priorytecie poprzez parametr `prio`. Można przekazać tu też wartość stałej `OS_PRIO_SELF`, jeżeli proces nie zna swojego własnego numeru priorytetu. W takim przypadku inny proces musi wznowić zawieszony proces poprzez użycie `OSTaskResume()`. Ponadto, jeśli proces zawiesza sam siebie, jest on pozbawiany procesora do czasu jego wznowienia, a kontrolę nad procesorem przejmuje następny proces „gotowy” o najwyższym priorytecie. `OSTaskSuspend()` zwraca następujące kody błędów:

- `OS_NO_ERR` – jeżeli operacja zakończyła się pomyślnie.
- `OS_TASK_SUSPEND_IDLE` – jeżeli próbowano zawiesić proces bezczynności.
- `OS_PRIO_INVALID` – jeżeli priorytet jest wyższy niż maksymalny dozwolony lub stała

OS_PRIO_SELF nie posiada określonej wartości.
• OS_TASK_SUSPEND_PRIO – jeżeli proces, który usiłowano zawiesić, nie istnieje.

Wznawianie procesów OSTaskResume ()

```
INT8U OSTaskResume(INT8U prio);
```

OSTaskResume() wznawia proces, który został zawieszony przez funkcję OSTaskSuspend(). Praktycznie OSTaskResume() jest jedyną funkcją, która może wznowić zawieszony proces. Argumentem tej funkcji jest prio określający priorytet procesu do wznowienia. OSTaskResume() zwraca następujące kody błędów:
• OS_NO_ERR – jeżeli funkcja wykonała się pomyślnie.
• OS_TASK_RESUME_PRIO – jeżeli proces, który ma być wznowiony, nie istnieje.
• OS_TASK_NOT_SUSPENDED – jeżeli proces nie został zawieszony.
• OS_PRIO_INVALID – jeżeli wartość prio jest wyższa lub równa OS_LOWEST_PRIO.

Semafory (Leszek Ciopiński)

W systemie µC/OS-II semafory składają się z dwóch elementów. Pierwszym z nich jest licznik: 16-bitowa zmienna całkowitoliczbowa bez znaku. Jej wartość określa ile procesów może jeszcze wejść do obszaru chronionego przez semafor. Drugim elementem jest lista procesów oczekujących na osiągnięcie przez licznik semafora wartości większej od zera. System operacyjny µC/OS-II udostępnia sześć funkcji do wykonywania operacji na semaforach: OSSemAccept(), OSSemCreate(), OSSemDel(), OSSemPend(), OSSemPost() i OSSemQuery().

W celu używania semaforów w systemie µC/OS-II, niezbędne jest dokonanie odpowiednich ustawień w pliku konfiguracyjnym OS_CFG.H. Mechanizm semaforów jest włączany poprzez ustawienie parametru OS_SEM_EN na 1. Jeśli ustawiony jest on na 0, to niezależnie od ustawień pozostałych parametrów, żadna z operacji semaforowych nie będzie udostępniana przez system operacyjny. Podstawowymi funkcjami, które są dostępne zawsze, gdy obsługa semaforów jest włączona są: OSSemCreate(), OSSemPend() i OSSemPost(). Użycie pozostałych funkcji może być załączane i wyłączane oddzielnie przy pomocy parametrów: OS_SEM_ACCEPT_EN dla OSSemAccept(), OS_SEM_DEL_EN dla OSSemDel(), oraz OS_SEM_QUERY_EN dla OSSemQuery().

Programista ma dostęp do semafora poprzez wskaźnik na typ OS_EVENT. Aby utworzyć semafor należy użyć funkcji OSSemCreate(). Przyjmuje ona tylko jeden parametr, którym jest liczba 16-bitowa bez znaku. Wartość ta przepisywana jest do licznika tworzonego semafora. Na ogół, jeśli semafor ma być wykorzystywany jako wskaźnik wystąpienia jakiegoś zjawiska, inicjalizowany jest wartością 0. W przeciwnym przypadku podaje się wartość określającą ilość procesów mogących na raz korzystać ze współdzielonych zasobów. Wartością zwracaną przez funkcję OSSemCreate() jest wskaźnik na semafor. Programista musi jednak sprawdzić, czy zwróconą wartością nie jest NULL. Oznaczałoby to, że w systemie nie ma już wolnych struktur ECB i semafor nie może zostać utworzony. Funkcja tworzenia semafora może być wywoływana poprzez proces lub kod startowy (po funkcji OSInit(), a przed OSStart()). Zanim semafor zostanie użyty zawsze musi najpierw zostać utworzony. Funkcje obsługi przerwania nie mogą tworzyć semaforów.

Funkcją do usuwania uprzednio utworzonego semafora jest OSSemDel(). Należy

używać jej jednak ze szczególną ostrożnością. Łatwo zauważyć, że może dojść do sytuacji, gdy któryś z procesów będzie próbował uzyskać dostęp do usuniętego semafora. Dlatego zaleca się, aby zanim semafor zostanie usunięty, usunięte zostały wszystkie procesy, które mogą się do niego odwoływać. Funkcję tą można wywoływać jedynie z poziomu innego procesu. Niedozwolone jest więc wywoływanie jej w funkcji obsługi przerwania (ISR).

Funkcja `OSSemDel()` przyjmuje trzy parametry. Pierwszym z nich jest wskaźnik na semafor, wyrażony typem `OS_EVENT`, który ma zostać usunięty. Drugi parametr to flaga określająca warunki usunięcia semafora. Przekazanie w nim flagi `OS_DEL_NO_PEND` oznacza, że semafor zostanie usunięty tylko wtedy, gdy przypisana do niego lista procesów oczekujących będzie pusta. Flaga `OS_DEL_ALWAYS` oznacza, że semafor zostanie usunięty niezależnie od tego, czy są procesy, które czekają na jego podniesienie, czy nie. Jeśli lista procesów oczekujących usuwanego semafora nie jest pusta, procesy z tej listy są przenoszone w stan procesów gotowych. Ostatnim parametrem pobieranym przez funkcję jest wskaźnik na zmienną przechowującą kod błędu. Jeśli operacja usuwania semafora zakończyła się powodzeniem, zmienna ta zawiera flagę `OS_NO_ERR`. W przeciwnym przypadku zwracana jest jedna z wartości: `OS_ERR_DEL_ISR` gdy funkcja usuwania wywoływana jest z ISR, `OS_ERR_INVALID_OPT` jeśli przekazano nieprawidłową flagę jako drugi parametr funkcji, `OS_ERR_TASK_WAITING` oznaczającą, że co najmniej jeden proces oczekuje na podniesienie semafora, `OS_ERR_EVENT_TYPE` gdy pierwszy parametr nie jest wskaźnikiem na semafor, oraz `OS_ERR_PEVENT_NULL` oznaczającym, że nie ma żadnej dostępnej struktury `OS_EVENT`. Funkcja zwraca `NULL` w przypadku powodzenia lub wskaźnik na semafor, jeśli nie udało się go usunąć.

Oczekiwanie na semafor jest dostarczane przez funkcję zajęcia semafora `OSSemPend()`. Funkcja ta może być wywoływana wyłącznie przez procesy. Pierwszy z trzech parametrów tej funkcji to wskaźnik na używany semafor. Drugim parametrem jest maksymalny czas oczekiwania na podniesienie semafora podany jako ilość impulsów zegara systemowego. Podanie wartości 0 oznacza, że proces będzie czekał do skutku. Ostatnim parametrem jest wskaźnik na zmienną przechowującą kod błędu operacji. W przypadku poprawnego jej wykonania zmienna ta przyjmie wartość `OS_NO_ERR`. Ponadto może przyjąć wartości: `OS_TIMEOUT` – gdy dostęp do zasobu nie został przyznany, a został przekroczony maksymalny czas oczekiwania, `OS_ERR_EVENT_TYPE` gdy pierwszy parametr nie jest wskaźnikiem na semafor, `OS_ERR_PEND_ISR` gdy funkcja wywoływana jest w procedurze obsługi przerwania, oraz `OS_ERR_PEVEBT_NULL` gdy wskaźnik na semafor jest pusty.

Funkcja `OSSemPend()` może jednak doprowadzić do wywłaszczenia procesu z procesora. Funkcja `OSSemAccept()` również służy do zajmowania semafora. W odróżnieniu jednak od poprzedniej funkcji pobiera tylko jeden parametr – wskaźnik na semafor. Funkcja ta zwraca 0 jeśli nie uzyskano dostępu do chronionego zasobu. Jeśli licznik semafora jest większy od 0 jego wartość jest dekrementowana, a funkcja `OSSemAccept()` zwraca wartość licznika sprzed dekrementacji. Zwrócenie więc wartości większej od 0 oznacza przyznanie dostępu. W odróżnieniu też od funkcji `OSSemPend()` funkcja `OSSemAccept()` nie powoduje wywłaszczenia procesu i może być wywołana również przez funkcję obsługi przerwania (ISR).

Sygnalizowanie semafora jest możliwe tylko przy użyciu funkcji `OSSemPost()`. Jako jedyny parametr pobiera ona wskaźnik na semafor. Jej wywołanie powoduje inkrementację licznika semafora. Sprawdzana jest wtedy też lista procesów oczekujących. Jeśli nie jest pusta, proces o najwyższym priorytecie przenoszony jest w stan „gotowy”. Należy więc zauważyć, że wywołanie tej funkcji może, ale nie musi, w sposób pośredni doprowadzić do wywłaszczenia procesu z procesora przez planistę. Dlatego funkcja ta może być wywoływana zarówno przez proces jak i funkcję obsługi przerwania (kiedy to planista jest wstrzymany). Wartością zwracaną przez `OSSemPost()` jest kod błędu. Wartość `OS_NO_ERR` oznacza, że operacja została wykonana pomyślnie, `OS_SEM_OVF` informuje o przekręceniu licznika semafora, wartość

OS_ERR_EVENT_TYPE oznacza, że parametr przekazany funkcji nie jest semaforem, a wartość OS_ERR_PEVENT_NULL określa przekazanie funkcji wskaźnika pustego – NULL.

µC/OS-II umożliwia też sprawdzenie stanu semafora, łącznie z jego wartością, bez jego modyfikacji. Służy do tego funkcja OSSemQuery(). Pobiera ona dwa parametry. Pierwszym jest wskaźnik na semafor. Drugi zaś to wskaźnik na zaalokowaną przez programistę strukturę OS_SEM_DATA, która będzie zawierała informacje o stanie semafora włączając w to wartość jego licznika. Wartością zwracaną jest kod błędu. Prawidłowe wykonanie funkcji sygnalizowane jest wartością OS_NO_ERR. W przypadku nie podania wskaźnika do semafora lecz na inny element zwrócona zostanie wartość OS_ERR_EVENT_TYPE. OS_ERR_PEVENT_NULL oznacza przekazanie funkcji wskaźnika NULL jako wskaźnik do semafora.

Muteksy (Leszek Ciopiński)

Podobnie jak wszystkie semafory binarne muteksy umożliwiają dostęp do współdzielonych zasobów tylko jednemu procesowi w danym czasie. Ponieważ przyjmują tylko dwa stany, nie można ich (w przeciwieństwie do zwykłych semaforów) wykorzystywać do zliczania ilości wystąpień pewnego, określonego zdarzenia w systemie. W odróżnieniu od zwykłego semafora binarnego, w systemie µC/OS-II muteksy spełniają jeszcze jedną funkcję – umożliwiają zredukowanie ilości problemów inwersji priorytetów. Do problemu tego typu może dojść w następującej sytuacji: dany jest proces o niskim priorytecie, który zajmuje zwykły semafor binarny. Następnie, zostaje on wywłaszczony na rzecz procesu o wysokim priorytecie, który również zamierza zająć wspomniany już semafor. Ponieważ nie uzyskuje dostępu, jest wywłaszczany i wznawiane jest wykonywanie procesu o niskim priorytecie. Problem pojawia się, jeśli w tym momencie pojawi się proces gotowy o priorytecie wyższym od procesu aktywnego, a niższym od procesu oczekującego na semafor. Planista przekaże mu bowiem procesor. W efekcie proces o najwyższym priorytecie będzie musiał czekać, aż najpierw wykona się proces o priorytecie niższym. Dopiero po zakończeniu wykonywania się procesu o średnim priorytecie kontrolę nad procesorem odzyska proces o priorytecie najniższym, do czasu aż nie zwolni zajętego semafora, kiedy to kontrolę nad procesorem ponownie odzyska proces o najwyższym priorytecie. Jest to typowy problem w systemach czasu rzeczywistego.

Muteksy w systemie µC/OS-II rozwiązują problem inwersji priorytetów w następujący sposób. Podczas ich tworzenia niezbędne jest podanie dowolnego ale wolnego, wysokiego priorytetu. Jeśli muteks jest zajęty przez proces o niskim priorytecie, a proces o priorytecie wyższym próbuje go zająć, proces o priorytecie niższym odzyskuje procesor. Aby jednak nie dopuścić do inwersji priorytetów, priorytet procesu odzyskującego procesor jest podnoszony do wartości priorytetu muteksu. Pierwotna wartość priorytetu procesu jest przywracana w chwili, gdy zwolni on muteks. Zabieg ten powoduje, że nawet jeśli pojawi się proces o priorytecie wyższym od aktualnego, będzie on musiał poczekać, aż zajmowany muteks zostanie zwolniony.

Muteksy systemu µC/OS-II zawierają trzy elementy: flagę określającą, czy muteks jest dostępny, czy nie (wartość 0 lub 1), priorytet, jaki ma zostać przypisany procesowi zajmującemu muteks w czasie, gdy proces o wyższym od niego priorytecie oczekuje na jego zajęcie, oraz listę procesów oczekujących na zwolnienie muteksu, by móc go zająć.

Do operacji na muteksach system µC/OS-II udostępnia sześć funkcji: OSMutexCreate(), OSMutexDel(), OSMutexPend(), OSMutexPost(), OSMutexAccept() i OSMutexQuery(). Aby możliwe było ich wykorzystywanie należy ustawić parametr OS_MUTEX_EN na 1 w pliku konfiguracyjnym OS_CFG.H. Jeśli parametr ten ustawiany jest na wartość 0, żadna z powyższych sześciu funkcji nie będzie dostępna, niezależnie od pozostałych ustawień konfiguracyjnych. Istnieje jeszcze możliwość indywidualnego wyłączania

i włączania trzech funkcji przy pomocy parametrów: OS_MUTEX_ACCEPT_EN dla OSMutexAccept(), OS_MUTEX_DEL_EN dla OSMutexDel() i OS_MUTEX_QUERY_EN dla OSMutexQuery().

Podobnie jak w przypadku zwykłych semaforów, dostęp do muteksów w systemie µC/OS-II odbywa się poprzez wskaźnik na zmienną typu OS_EVENT. Funkcją służącą do tworzenia muteksu jest OSMutexCreate(). Pobiera ona dwa parametry: pierwszym jest priorytet, który ma być wykorzystywany w celu uniknięcia problemu inwersji priorytetów. Musi on być wyższy od priorytetu każdego z procesów, który będzie ubiegać się o zajęcie muteksu. Ponadto priorytet muteksu nie może być używany przez jakikolwiek inny proces ani w chwili jego tworzenia, ani w czasie jego istnienia. Drugim parametrem jest wskaźnik na zmienną przetrzymującą kod błędu operacji. Wartość OS_NO_ERR oznacza pomyślne utworzenie muteksu. Wystąpienie błędu jest sygnalizowane przez następujące wartości: OS_ERR_CREATE_ISR – gdy funkcja tworzenia muteksu wywoływana jest w funkcji obsługi przerwania, OS_PRIO_EXIST jeśli istnieje już w systemie proces o priorytecie podanym jako pierwszy parametr funkcji, OS_ERR_PEVENT_NULL gdy nie ma już dostępnych struktur OS_EVENT, oraz OS_PRIO_INVALID jeśli podany został priorytet o wartości większej (czyli priorytet niższy) niż wartość parametru OS_LOWEST_PRIO. Wartością zwracaną przez funkcję tworzenia muteksu jest wskaźnik na nowo utworzony element.

Do usunięcia uprzednio utworzonego muteksu służy funkcja OSMutexDel(). Podobnie, jak w przypadku usuwania semaforów, należy funkcji usuwania muteksów używać ostrożnie, tak aby nie doszło do sytuacji, gdy któryś z procesów będzie chciał się odwołać do nieistniejącego już muteksu. W związku z powyższym zaleca się aby przed usunięciem muteksu, usunąć wszystkie procesy, które mogą się do niego odwoływać.

Pierwszym parametrem funkcji OSMutexDel() jest wskaźnik na muteks. Kolejny parametr określa warunek usunięcia muteksu. Ustawienie go na OS_DEL_NO_PEND oznacza, że muteks będzie usunięty tylko wówczas, gdy żaden proces nie próbuje go zająć. Flaga OS_DEL_ALWAYS nakazuje bezwzględne usunięcie muteksu. W takim przypadku wszystkie procesy oczekujące na jego zajęcie zostaną przeniesione w stan procesów gotowych. Ostatnim parametrem jest wskaźnik na zmienną przechowującą kod błędu. Jeśli funkcja została wykonana poprawnie, jej wartość to OS_NO_ERR. W przeciwnym przypadku możemy otrzymać następujące komunikaty: OS_ERR_DEL_ISR jeśli funkcja wywoływana jest z procedury obsługi przerwania, OS_ERR_INVALID_OPT gdy jako drugi parametr podana została niewłaściwa flaga, OS_ERR_TASK_WAITING jeśli istnieje przynajmniej jeden proces oczekujący na zajęcie muteksu, OS_ERR_EVENT_TYPE w przypadku, gdy wskaźnik przekazany jako pierwszy parametr nie jest muteksem, a OS_ERR_PEVENT_NULL jeśli wskaźnik ten jest o wartości NULL.

Do zajmowania muteksu system µC/OS-II udostępnia dwie funkcje. Pierwszą z nich jest OSMutexPend(). Może ona spowodować wywłaszczenie procesu, jeśli muteks nie jest wolny w momencie wywołania tej funkcji. Jako pierwszy parametr przyjmuje ona wskaźnik na muteks. Drugim parametrem jest czas podany jako ilość impulsów zegara systemowego, po przekroczeniu którego proces z powrotem przejdzie do stanu gotowego, nawet jeśli nie uda mu się zająć muteksu. Podanie wartości 0 oznacza jednak, że proces będzie czekał „do skutku”. Ostatnim, trzecim, parametrem jest wskaźnik na zmienną przechowującą kod błędu. Wartość OS_NO_ERR oznacza, że funkcja wykonana została prawidłowo, a muteks został zajęty. Wartość OS_TIMEOUT informuje o przekroczeniu czasu oczekiwania na zajęcie muteksu. W takiej sytuacji proces nie uzyskał dostępu do chronionego zasobu. OS_ERR_EVENT_TYPE informuje o nie przekazaniu w pierwszym parametrze funkcji wskaźnika na muteks, a OS_ERR_PEVENT_NULL, że parametr ten jest wartości NULL. Ostatnim komunikatem może być OS_ERR_PEND_ISR oznaczającym wywołanie funkcji z procedury obsługi przerwania (ISR).

Drugą funkcją służącą do zajmowania muteksu w systemie µC/OS-II jest

OSMutexAccept(). Nie powoduje ona jednak blokowania procesu. Przyjmuje ona dwa argumenty. Pierwszym jest wskaźnik na muteks, a drugim wskaźnik na zmienną przechowującą kod błędu. Jeśli jej wartość, to OS_NO_ERR oznacza to, że wywołanie funkcji wykonane zostało prawidłowo. Błąd sygnalizowany jest następującymi wartościami: OS_ERR_EVENT_TYPE jeśli przekazany w pierwszym parametrze wskaźnik nie wskazuje na muteks, a OS_ERR_PEVENT_NULL jeśli wartość pierwszego parametru wynosi NULL. Wartość OS_ERR_PEND_ISR informuje, że funkcja wywołana została z funkcji obsługi przerwania. Omawiana funkcja zwraca jedną z dwóch wartości: 0, gdy muteks zajęty jest przez inny proces, oraz 1 jeśli zajęcie muteksu zakończyło się powodzeniem.

Do zwalniania muteksu służy funkcja OSMutexPost(). Przyjmuje ona tylko jeden parametr – wskaźnik na muteks. Wartością zwracaną jest kod błędu. W przypadku prawidłowego wykonania się operacji, uzyskamy wartość OS_NO_ERR. Przy nieprawidłowym jej wykonaniu, możemy otrzymać jedną z następujących wartości: OS_ERR_EVENT_TYPE jeśli przekazany przez nas parametr nie jest wskaźnikiem na muteks, a jeśli jest on równy NULL otrzymamy kod OS_ERR_PEVENT_NULL, OS_ERR_POST_ISR informuje o wywołaniu funkcji z procedury obsługi przerwania, a kod OS_ERR_NOT_MUTEX_OWNER, jeśli proces usiłuje zwolnić muteks, którego nie zajął.

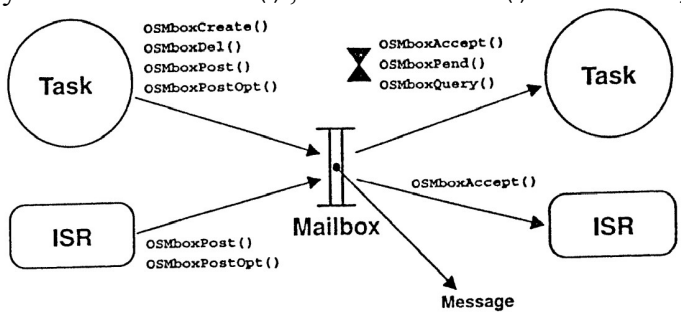
Istnieje również możliwość sprawdzenia bieżącego stanu muteksu bez jego modyfikowania. Funkcja OSMutexQuery() jako pierwszy parametr przyjmuje wskaźnik na muteks, a jako drugi – wskaźnik na zaalokowaną przez program strukturę typu OS_MUTEX_DATA, do której zapisane będą bieżące informacje o stanie muteksu. Funkcja zwraca kod błędu informujący o powodzeniu operacji (wartość OS_NO_ERR) lub przyczynach jego niepowodzenia. Znaczenie zwracanych wartości: OS_ERR_EVENT_TYPE i OS_ERR_PEVENT_NULL jest takie samo jak w funkcji opisywanej powyżej. Wartość OS_ERR_QUERY_ISR informuje o wywołaniu funkcji z procedury obsługi przerwania.

Skrzynki wiadomości (Dominik Bąk)

Skrzynka pocztowa w µC/OS II jest obiektem pozwalającym procesom i procedurom obsługi przerwań na wysłanie zmiennej o wielkości wskaźnika do innego zadania. Wskaźnik ten najczęściej wskazuje na strukturę danych, specjalnie przygotowaną dla danej aplikacji, zawierającą przesyłaną wiadomość. µC/OS II dostarcza sześć funkcji do obsługi skrzynek pocztowych: OSMboxCreate(), OSMboxPend(), OSMboxPost(), OSMboxPostOpt(), OSMboxAccept() i OSMboxQuery(). Możliwość korzystania ze skrzynek pocztowych w µC/OS II wiąże się z odpowiednią konfiguracją parametrów w OS_CFG.H. Należy pamiętać o tym, że żadna z funkcji systemowych nie jest dostępna, kiedy OS_MBOX_EN przyjmuje wartość 0. W celu włączenia obsługi skrzynek, pierwszą czynnością, jaką trzeba podjąć jest przestawienie tego parametru na 1. Należy zauważyć, że OSMboxCreate() i OSMboxPend() nie mogą być pojedynczo wyłączane, w przeciwieństwie do pozostałych czterech funkcji. Funkcje te są bowiem niezbędne, jeśli obsługa skrzynek, w systemie µC/OS-II, jest włączona. Ponadto koniecznie należy włączyć co najmniej jedną z funkcji służących do wysyłania wiadomości: OSMboxPost() lub OSMboxPostOpt().

Rysunek 3.3. przedstawia rodzaje relacji, jakie zachodzą między procesami, procedurami obsługi przerwań (ISR) i skrzynkami pocztowymi. Skrzynka pocztowa może przechowywać tylko jeden wskaźnik (jest wtedy pełna) lub przechowywać wskaźnik NULL (skrzynka pocztowa jest pusta). Maksymalna ilość dostępnych skrzynek ustawiana jest poprzez parametr OS_MAX_EVENTS w pliku nagłówkowym OS_CFG.H. Jak pokazano na rysunku 3.3., zarówno proces, jak i ISR mogą wysyłać wiadomości poprzez OSMboxPost() lub OSMboxPostOpt(), jednakże tylko

procesom wolno wywoływać `OSMboxDel()`, `OSMboxPend()` i `OSMboxQuery()`.



Rysunek 3.: Relacje między zadaniami, ISR i skrzynką pocztową wiadomości. [1 - strona 230]

Tworzenie skrzynki pocztowej, OSMboxCreate()

```
OS_EVENT * OSMboxCreate (void * msg);
```

Skrzynka pocztowa musi zostać utworzona, zanim zostanie użyta. Tworzenie skrzynki pocztowej wiąże się z użyciem funkcji `OSMboxCreate()` i określeniem jej wartości początkowej, poprzez wskaźnik `msg`. Najczęściej jest to wartość `NULL` (skrzynka pusta), ale możliwe jest też umieszczenie w niej wiadomości (skrzynka pełna). Użycie określonego rozwiązania uzależnione jest od zastosowania skrzynki. Jeżeli skrzynka użyta jest do sygnalizacji zdarzeń, to inicjujemy ją pustym wskaźnikiem (`NULL`), ponieważ inna wartość uruchomiłaby obsługę zdarzenia, które nie wystąpiło. W przypadku wykorzystywania skrzynek do przyznawania dostępu do zasobów dzielonych, wywoływana jest funkcja z niezerową wartością wskaźnika (non-`NULL`). W tym przypadku skrzynki użyte są jako semafory binarne.

Usuwanie skrzynek pocztowych OSMboxDel()

```
OS_EVENT *OSMboxDel (OS_EVENT *pevent, INT8U opt, INT8U *err);
```

Funkcja ta usuwa wskazaną skrzynkę pocztową i odblokowuje wszystkie procesy na nią czekające. Należy pamiętać, aby rozważnie z niej korzystać, ponieważ procesy mogą próbować odwoływać się do usuniętej już skrzynki. Procesy mogą się przed tym zabezpieczyć poprzez sprawdzenie parametrów zwracanych przez funkcję `OSMboxPend()`. Nie można jej, w tym celu, zastąpić funkcją `OSMboxAccept()`, ponieważ zwracane wartości nie umożliwiają sprawdzenia , czy skrzynka została usunięta Wówczas, jedynym sposobem ustalenia, czy skrzynka istnieje, czy nie, jest sprawdzenie parametru `pevent`. Jeśli okaże się, że wartość wskaźnika to `NULL`, oznacza to usunięcie skrzynki z systemu. Wywołanie funkcji usuwania skrzynki powoduje wyłączenie obsługi przerwań na czas odblokowywania procesów. Może to wydłużyć czas oczekiwania na obsłużenie przerwania. Czas wyłączenia przerwań jest wprost proporcjonalny do liczby procesów oczekujących na wiadomość ze skrzynki. Ponieważ wszystkie procesy oczekujące na skrzynkę pocztową będą gotowe do wykonania, należy być ostrożnym w aplikacjach, gdzie skrzynki używane są do ograniczania dostępu do zasobów, ponieważ przestaną one być chronione.

Parametrami funkcji `OSMboxDel()` są: `pevent` czyli wskaźnik na usuwaną skrzynkę, `opt` który udostępnia dwie metody usunięcia skrzynek: `OS_DEL_NO_PEND` (usunięcie skrzynki tylko wówczas, gdy żaden proces nie oczekuje na pobranie z niej danych) lub `OS_DEL_ALWAYS` (bezwzględne usunięcie skrzynki – procesy oczekujące przejdą w stan „procesów gotowych”) oraz `err` wskaźnik na zmienną zawierającą kod błędu.

Oczekiwanie na wiadomości od skrzynek OSMboxPend()

```
void *OSMboxPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);
```

Funkcja `OSMboxPend()` jest używana do odbierania wiadomości ze skrzynki. Dane są wysyłane do procesów przez ISR lub inne procesy. Jeśli wiadomość jest w skrzynce pocztowej, kiedy funkcja `OSMboxPend()` jest wywoływana, to komunikat jest pobierany, a skrzynka opróżniana. Następnie wiadomość zwracana jest do procesu poprzez wywoływaną przez niego funkcję. Jeśli w skrzynce nie ma komunikatu, funkcja `OSMboxPend()` wstrzymuje wywołujący ją proces, dopóki wiadomość nie zostanie odebrana, chyba że użytkownik określił maksymalny czas oczekiwania na jej odbiór. Jeśli wiadomość wysyłana jest do skrzynki, z której wiele procesów czeka na odbiór wiadomości, to system uruchamia proces z najwyższym priorytetem. Oczekujący proces, który został zawieszony przez `OSTaskSuspend()` może odebrać komunikat. Proces ten pozostaje jednak wstrzymany do momentu wznowienia go przez `OSTaskResume()`.

Pierwszym argumentem funkcji jest `pevent` – wskaźnik na skrzynkę, z której wiadomość ma zostać odebrana. Kolejny parametr to `timeout`, pozwalający uruchomić proces, jeśli wiadomość nie zostanie odebrana ze skrzynki po określonej liczbie cykli zegara. Ustawienie wartości 0 w `timeout` oznacza, że proces będzie oczekiwał na wiadomość, aż do momentu jej odebrania. Maksymalny limit czasu oczekiwania to 65.535 cykli zegara. Ostatnim parametrem funkcji jest `err` wskaźnik na zmienną przechowującą kod błędu.

Wysyłanie wiadomości do skrzynek OSMboxPost()

```
INT8U OSMboxPost(OS_EVENT *pevent, void *msg);
```

`OSMboxPost()` wysyła wiadomość do innego procesu za pośrednictwem skrzynki pocztowej. Jeśli wiadomość jest już w skrzynce to zwracany jest kod błędu informujący, że skrzynka jest pełna. Następnie, `OSMboxPost()` zwraca wywołującemu ją procesowi komunikat, że wiadomość nie została umieszczona w skrzynce. Jeżeli jakieś procesy oczekują na wiadomość w skrzynce, to proces o wyższym priorytecie odbierze z niej komunikat jako pierwszy . Jeśli jednak proces oczekujący na wiadomość ma wyższy priorytet niż proces wysyłający komunikat, to ten o wyższym priorytecie zostanie wznowiony, a ten o niższym zostanie wstrzymany, czyli nastąpi przełączenie kontekstu.

Pierwszym z argumentów funkcji jest `pevent`, czyli wskaźnik na skrzynkę, w której wiadomość ma być przechowywana. Następny, to `msg` czyli komunikat wysyłany do procesu poprzez skrzynkę. Należy pamiętać o tym, żeby nie ustawiać tego wskaźnika na `NULL`, ponieważ oznacza to, że skrzynka jest pusta. `OSMboxPostOpt()` jest funkcją rozszerzoną o dodatkowy parametr `opt` , który służy do określenia, czy komunikat ma być wysłany do jednego (`OS_POST_OPT_NONE`), czy wielu (`OS_POST_OPT_BROADCAST`) procesów oczekujących na wiadomość ze skrzynki.

Otrzymywanie wiadomości bez oczekiwania OSMboxAccept ()

```
void *OSMboxAccept(OS_EVENT *pevent);
```

`OSMboxAccept()` pozwala na sprawdzenie, czy wiadomość z określonej skrzynki jest dostępna. W odróżnieniu od `OSMboxPend()`, `OSMboxAccept()` nie powoduje zawieszenia procesu, jeśli wiadomości nie są dostępne. Jeśli wiadomość jest dostępna, to zwracana jest ona

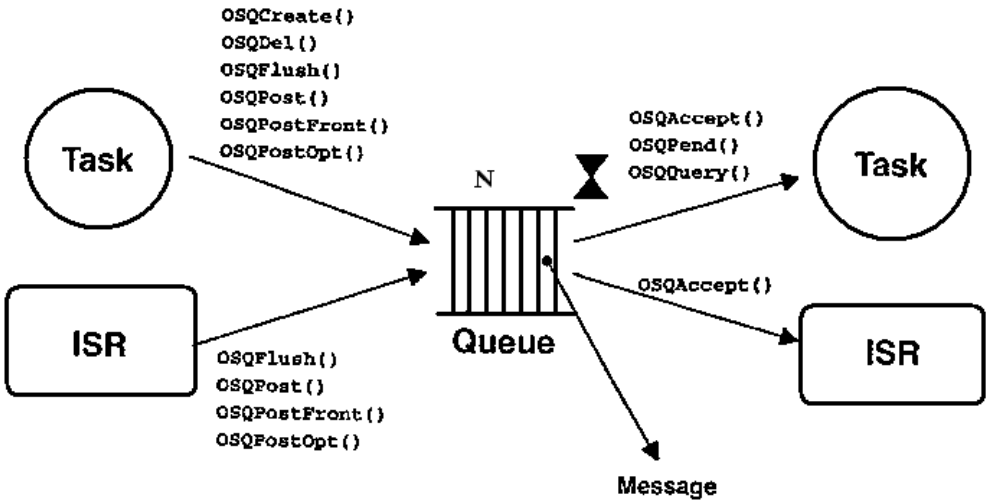
przez opisywaną funkcję, a zawartość skrzynki jest czyszczona. Niniejsza funkcja jest zwykle używane przez procedurę obsługi przerwań (ISR), ponieważ ISR nie może być blokowana. Argumentem funkcji jest `pevent` wskazujący skrzynkę, z której wiadomość ma zostać odebrana. Wartości zwracane przez funkcję to `NULL`, gdy skrzynka jest pusta, lub wiadomość, która znajdowała się w skrzynce.

Kolejki komunikatów (Dominik Bąk)

Kolejki w µC/OS II są wykorzystywane do przesyłania procesom lub ISR, wskaźników od innych procesów. Każdy ze wskaźników wskazuje na strukturę danych, zawierającą wiadomość. µC/OS II udostępnia dziewięć takich funkcji do obsługi kolejek: `OSQCreate()`, `OSQDel()`, `OSQPend()`, `OSQPost()`, `OSQPostFront()`, `OSQPostOpt()`, `OSQAccept()`, `OSQFlush()` i `OSQQuery()`.

`OSQCreate()` tworzy kolejkę komunikatów, która pozwala procesom lub procedurom obsługi przerwań na wysłanie zmiennych o wielkości wskaźnika do jednego lub więcej procesów. Argumenty funkcji to: `start`, czyli adres bazowy pamięci zarezerwowanej i zadeklarowanej jako tablica wskaźników, oraz `size` informujący o rozmiarze tablicy. `OSQDel()` usuwa kolejkę komunikatów, a wszystkie procesy, oczekujące na wiadomość od niej, przenosi w stan „procesów gotowych”. Wśród argumentów funkcji znajdują się: `pevent`, czyli wskaźnik do kolejki, `opt` umożliwiający dokonanie wyboru opcji usunięcia kolejki: usuń zawsze, niezależnie od tego, czy są procesy oczekujące na wiadomość z kolejki, czy nie (`OS_DEL_ALWAYS`) lub tylko wtedy gdy nie ma procesów oczekujących (`OS_DEL_NO_PEND`). Ostatnim parametrem jest `err` – wskaźnik na zmienną zawierającą kod błędu. Funkcja `OSQPend()` służy do odbioru wiadomości z kolejki, przy czym może powodować wstrzymanie wykonywania procesu. Argumenty funkcji to `pevent` – wskaźnik na kolejkę, z której wiadomości są odbierane, `timeout` pozwala przejść procesowi do stanu „proces gotowy”, jeśli wiadomość nie zostanie odebrana z kolejki w określonej liczbie cykli zegara (wyjątkiem jest wartość 0 oznaczająca oczekiwanie „do skutku”) oraz `err` – wskaźnik na zmienną przechowującą kod błędu. `OSQPost()` wysyła wiadomości do określonej kolejki. Pierwszy parametr funkcji to: `pevent` – wskaźnik do kolejki, do której wiadomość jest przesłana. Wskaźnik ten jest zwracany do aplikacji po utworzeniu kolejki, `msg` określa wiadomość do wysłania, parametr ten nie może przyjmować wartości `NULL`. `OSQPostFront()` wysyła wiadomości do określonej kolejki, ale w odróżnieniu od `OSQPost()`, komunikat zostaje umieszczony na początku zamiast na końcu kolejki. Użycie `OSQPostFront()` pozwala na "priorytetowe" wysyłanie wiadomości. `OSQPostOpt()` tak jak poprzednie funkcje wysyła wiadomość do kolejki. Funkcja ta powstała w celu zmniejszenia wielkości kodu, gdyż może zastąpić zarówno `OSQPost()` jak i `OSQPostFront()`. Ponadto daje możliwość przesłania wiadomości do wszystkich zadań oczekujących w kolejce. Wszystkie te opcje są konfigurowane w parametrze `opt`. `OSQAccept()` sprawdza, czy wiadomość jest dostępna w określonej kolejce. `OSQAccept()` w przeciwieństwie do `OSQPend()` nie zawiesza wywoływanego procesu, jeżeli wiadomość jest niedostępna. `OSQFlush()` usuwa zawartość kolejki i likwiduje wszystkie wiadomości do niej wysłane. `OSQQuery()` uzyskuje informacje na temat wiadomości w kolejce. Należy pamiętać, że tworzona aplikacja musi alokować strukturę danych `OS_Q_DATA`, która jest potrzebna do otrzymywania informacji z bloku kontrolnego kolejki wiadomości. `OSQQuery()` pozwala też na ustalenie, czy jakiekolwiek proces oczekuje w kolejce i jak wiele ich jest (przez zliczanie ilości wartości 1 w polu `.OSEventTbl[]`). Dzięki tej funkcji można określić również ile wiadomości znajduje się w kolejce i jaki jest jej rozmiar. W celu użycia wyżej opisanych funkcji należy dokonać odpowiedniej konfiguracji w `OS_CFG.H`. Warto pamiętać, że żadna z usług kolejki

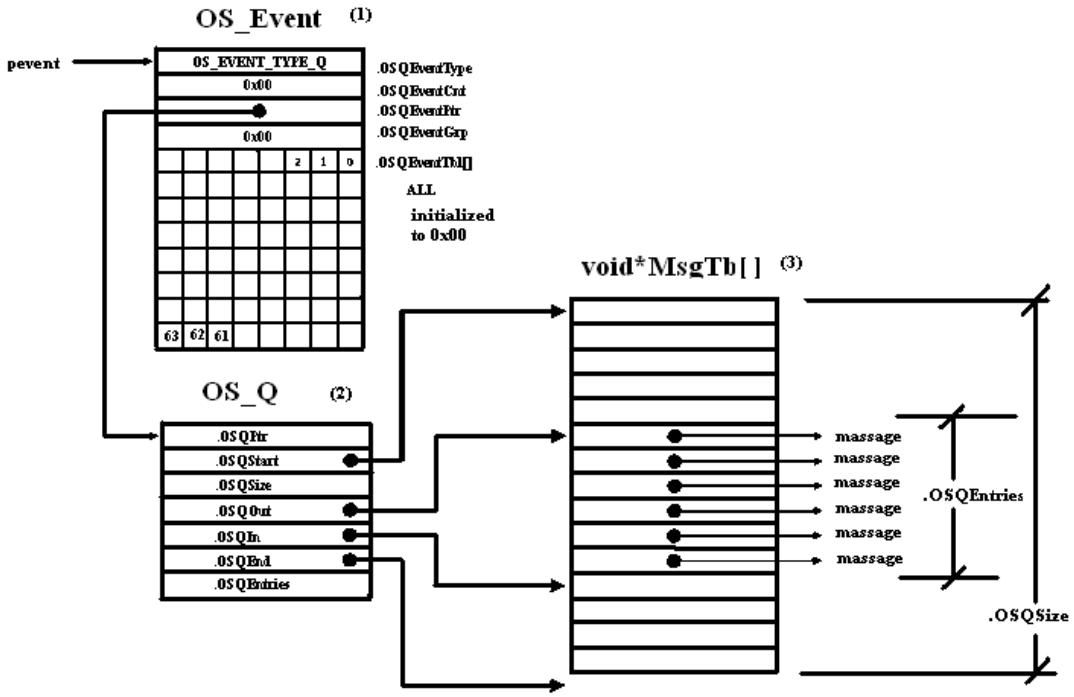
nie będzie aktywna, jeżeli `OS_Q_EN` lub `OS_MAX_QS` będzie ustawione na 0. Aby móc korzystać z kolejek niezbędna jest zmiana konfiguracji powyższych parametrów. Wartość pierwszego z nich należy ustawić na wartość 1. Drugi, oznacza maksymalną dopuszczalną ilość kolejek w aplikacji i dlatego jego wartość musi być większa od 0. Należy również zauważyć, że `OSQCreate()` i `OSQPend()` nie mogą być pojedynczo wyłączane, jak inne usługi. Na rysunku 4. przedstawione są relacje jakie zachodzą między kolejką, a procesami i ISR. Warto zwrócić uwagę, że kolejka budową przypomina skrzynkę mogącą pomieścić wiele wiadomości. Klepsydra na tym grafie reprezentuje czas wstrzymania procesu, który może być określony podczas wywoływania `OSQPend()`. `N` reprezentuje liczbę pozycji, które kolejka obsługuje. Kolejka jest pełna, gdy aplikacja użyje funkcji `OSQPost()` [`OSQPostFront()` lub `OSQPostOpt()`] *`N`* razy, zanim zostanie wywołane `OSQPend()` lub `OSQAccept()`.



Rysunek 4. Relacje między zadaniami, ISR i kolejkami wiadomości. [1 - strona 248]

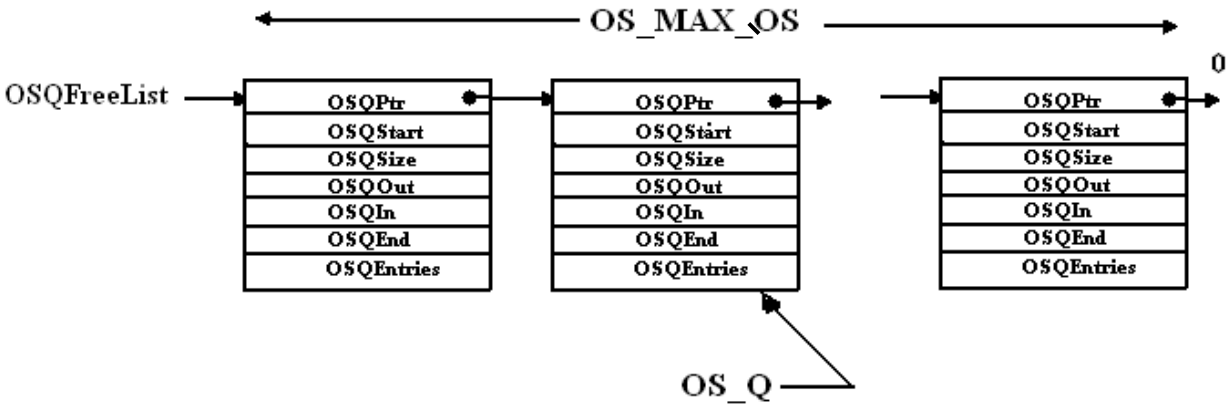
Jak przedstawiono na rysunku 4., proces i procedura obsługi przerwań (ISR) mogą wywoływać `OSQPost()`, `OSQPostFront()`, `OSQPostOpt()`, `OSQFlush()` lub `OSQAccept()`. Jednak tylko proces może wywołać `OSQDel()`, `OSQPend()` i `OSQQuery()`.

Na rysunku 5. pokazano różne struktury danych, niezbędne do implementacji kolejek wiadomości.



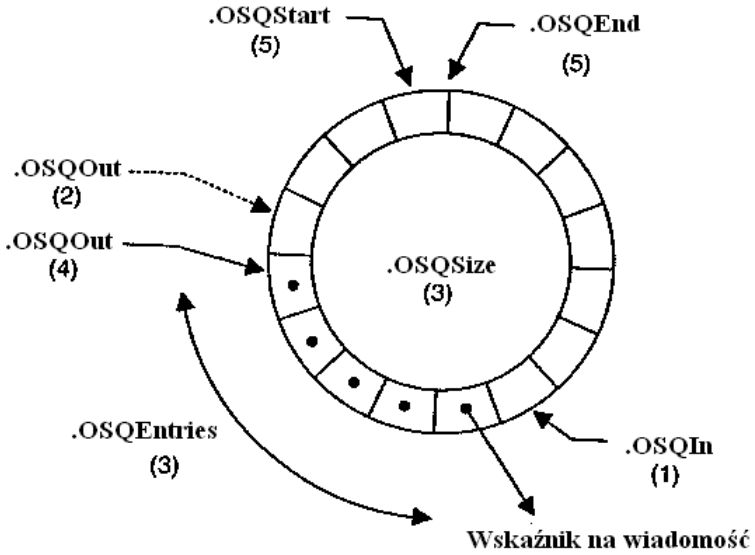
Rysunek 5. Struktura danych użyta przy kolejkach wiadomości. [1 – strona 249]

Konfiguracja stałej `OS_MAX_QS` w `OS_CFG.H` określa jak wiele kolejek może zostać użytych w aplikacji. Wartość ta powinna być większa niż 0. Kiedy µC/OS II jest zainicjowany, lista wolnych bloków kontroli kolejek jest tworzona tak, jak pokazano to na rysunku 6.



Rysunek 6. Lista wolnych bloków kontroli kolejek. [1 – strona 250]

Należy pamiętać, że kolejka przy tworzeniu jest pusta i tworzona jako bufor cykliczny, co przedstawia rysunek 7.

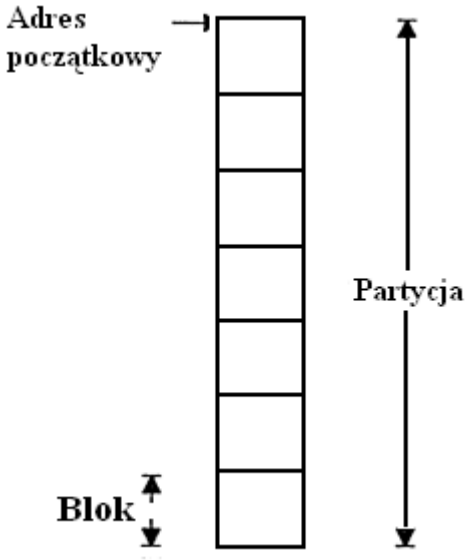


Rysunek 7. Kolejka wiadomości jako bufor cykliczny wskaźników. [1 – strona 251]

Zarządzanie pamięcią (Dominik Bąk)

Aplikacja powinna przydzielać i zwalniać pamięć dynamicznie. W tym celu używa się funkcji `malloc()` i `free()`. Wykorzystywanie ich w systemach czasu rzeczywistego wiąże się jednak z pewnym niebezpieczeństwem, gdyż z powodu fragmentacji pamięci może dojść do tego, że nie będzie można wykorzystać graniczących obszarów pamięci. Czas wykonywania `malloc()` i `free()` jest również dość długi z powodu algorytmów użytych do obsługi bloków pamięci.

µC/OS II udostępnia alternatywne funkcje działające jak `malloc()` i `free()`, umożliwiające aplikacji uzyskanie bloku pamięci o stałej wielkości z partycji utworzonej jako ciągły obszar, co przedstawiono na rysunku 8.

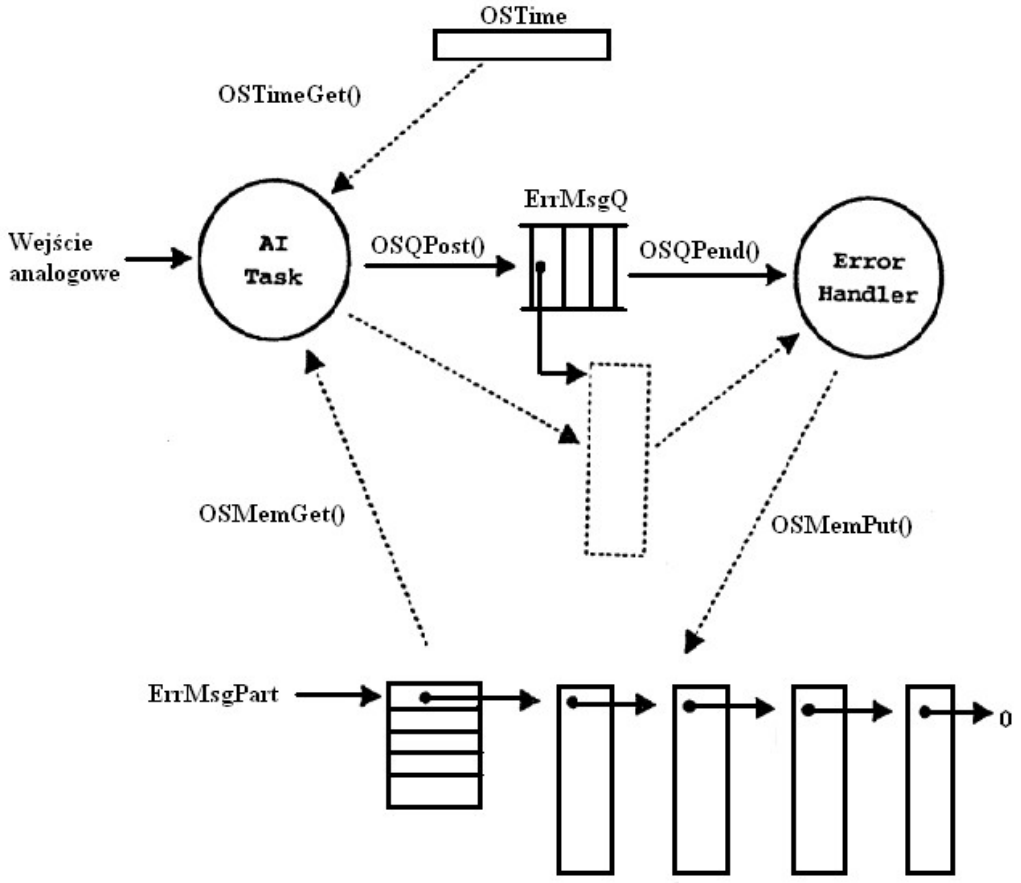


Rysunek 8. Partycja pamięci. [1 – strona 275]

Wszystkie bloki danej partycji są tej samej wielkości i są odpowiednio zaadresowane. Przydział i określenie bloków odbywa się deterministycznie i w stałym czasie.

Warto wspomnieć, że w systemie może występować więcej niż jeden podział pamięci, więc aplikacja może uzyskać bloki pamięci różnych wielkości. Jednakże, każdy blok pamięci musi zostać zwrócony do partycji, z której został pobrany. Taki typ zarządzania pamięcią jest mniej podatny na fragmentację.

Aby włączyć usługi zarządzania pamięcią w µC/OS II, należy ustawić odpowiednią konfigurację w `OS_CFG.H`. Warto wspomnieć o tym, że żadna z funkcji obsługi pamięci nie jest dostępna, jeżeli parametr `OS_MEM_EN` ustawiony jest na 0. Aby umożliwić ich używanie, wystarczy przestawić wspomniany parametr na wartość 1. Warto zwrócić uwagę na to, iż funkcji `OSMemCreate()`, `OSMemGet()` i `OSMemPut()` nie można indywidualnie wyłączać, jak inne funkcje zarządzające, ponieważ są niezbędne, gdy system zarządzania pamięcią w µC/OS-II jest włączony. Rysunek 9. przedstawia przykład dynamicznego przydziału pamięci w µC/OS-II.



Rysunek 9. Dynamiczny przydział pamięci. [1 – strona 283]

Poniżej opisane zostały funkcje, związane z zarządzaniem pamięcią.

Tworzenie partycji pamięci `OSMemCreate()`

```
OS_MEM * OSMemCreate (void * addr, INT32U nblks, INT32U blksize,
INT8U * err);
```

`OSMemCreate()` tworzy i inicjuje partycje pamięci. Każda z nich zawiera określoną liczbę ustalonej wielkości bloków. Aplikacja może uzyskać dostęp do każdego z nich, a kiedy dany blok przestaje być potrzebny, następuje jego odłożenie z powrotem na partycję. Do argumentów

funkcji należą addr czyli adres początku obszaru pamięci, który jest używany do utworzenia stałej wielkości bloków pamięci. Partycje mogą być tworzone przy użyciu statycznych tablic lub funkcji malloc() podczas uruchamiania. Drugi parametr to nblks zawierający liczbę dostępnych bloków pamięci określonej partycji. Żadna partycja nie może zawierać mniej, niż dwa bloki pamięci. Kolejny argument to blksize określający rozmiar (w bajtach) każdego bloku pamięci wewnątrz partycji. Musi być wystarczająco duży, aby pomieścić co najmniej jeden wskaźnik. Ostatni parametr to err – wskaźnik na zmienną informującą o kodzie błędu. OSMemCreate() zwraca wskaźnik do bloku kontroli pamięci tworzonej partycji, jeżeli jest ona dostępna. W przeciwnym przypadku zwraca wartość NULL.

Pobieranie bloków pamięci z partycji OSMemGet()

```
void *OSMemGet(OS_MEM *pmem, INT8U *err);
```

OSMemGet() pobiera blok z partycji pamięci. Zakłada się, że aplikacja zna rozmiar każdego bloku pamięci. Ponadto aplikacja musi zwrócić blok pamięci [używając OSMemPut()], kiedy go już nie potrzebuje. Można wywołać OSMemGet() więcej niż raz, dopóki wszystkie bloki pamięci nie są przydzielone. Argumentami funkcji są pmem i err. Pierwszy z nich jest wskaźnikiem na blok kontrolny partycji pamięci, który jest zwracany do aplikacji przez wywołanie OSMemCreate(). Drugi to wskaźnik na zmienną przechowującą kod błędu. OSMemGet() zwraca wskaźnik do przydzielonego bloku pamięci, jeśli jest dostępny lub w przeciwnym razie, zwracana jest wartość NULL.

Zwracanie bloków do partycji pamięci OSMemPut()

```
INT8U OSMemPut(OS_MEM *pmem, void *pblk);
```

OSMemPut() zwraca blok pamięci do partycji. Zakłada się, że będzie zwracany odpowiedni blok pamięci do odpowiedniej partycji. Argumenty OSMemPut() to pmem i pblk. Pierwszy z nich, podobnie jak w poprzedniej funkcji, jest wskaźnikiem na blok kontroli partycji, który jest zwracany do aplikacji przez wywołanie OSMemCreate(). pblk jest wskaźnikiem do bloku pamięci, który ma być odłożony na partycję. OSMemPut() zwraca następujące kody błędów:

- OS_NO_ERR gdy blok pamięci był dostępny i wrócił do swojej partycji.
- OS_MEM_FULL jeśli partycja nie może przyjąć więcej bloków pamięci, bo jest już pełna. Pojawienie się tego błędu informuje, że w tworzonej aplikacji wystąpił błąd, bo zwróciła ona więcej bloków niż pobrała.
- OS_MEM_INVALID_PMEM – przekazany parametr pmem jest równy NULL.
- OS_MEM_INVALID_PBLK – przekazany parametr pblk jest równy NULL.

Uzyskiwanie informacji na temat partycji pamięci OSMemQuery()

```
INT8U OSMemQuery(OS_MEM *pmem, OS_MEM_DATA *pdata);
```

OSMemQuery() pobiera informacje na temat partycji pamięci. Zasadniczo funkcja ta zwraca te same informacje, co struktura danych OS_MEM, ale w postaci nowej struktury danych OS_MEM_DATA, zawierającej dodatkowe pole, w którym informuje o liczbie bloków pamięci w użyciu.

Argumentami funkcji są `pmem` i `pdata`. Pierwszy jest wskaźnikiem do bloku kontroli partycji pamięci, który jest zwracany do aplikacji przez wywołanie `OSMemCreate()`. Drugi jest wskaźnikiem do struktury danych typu `OS_MEM_DATA`, która zawiera następujące pola:

```
void * OSAddr; / * wskazuje na adres początku partycji pamięci * /
void * OSFreeList; / * wskazuje na początek listy wolnych bloków pamięci * /
INT32U OSBlkSize; / * Rozmiar (w bajtach) każdego bloku pamięci * /
INT32U OSNBlks; / * Łączna liczba bloków w partycji * /
INT32U OSNFree; / * Liczba wolnych bloków pamięci * /
INT32U OSNUsed; / * Liczba używanych bloków pamięci * /
```

Funkcja `OSMemQuery()` może zwrócić jedną z poniższych wartości:
- `OS_NO_ERR` – jeżeli operacja zakończyła się powodzeniem.
- `OS_MEM_INVALID_PMEM` – jeżeli przekazany funkcji argument `pmem` jest równy `NULL`.
- `OS_MEM_INVALID_PDATA` – jeżeli przekazany funkcji argument `pdata` jest równy `NULL`.

Bibliografia

1. Labrosse Jean J., *MicroC/OS-II. The Real-Time Kernel*, wyd. 2, San Francisco, CMP Books, 2002, ISBN 1-57820-103-9