ISIX-RTOS v3 – system operacyjny dla mikrokontrolerów Cortex-M (1)

Opis funkcjonalny i charakterystyka systemu

Gdy mikrokontrolery były raczej nieskomplikowanymi układami, tworzenie oprogramowania najczęściej sprowadzało się do bezpośredniego odwoływania się do rejestrów układów peryferyjnych mikrokontrolera z wykorzystaniem programu napisanego w języku C lub w assemblerze. Współcześnie wymagania użytkownika końcowego wymuszają na twórcach oprogramowania wbudowanego zupełnie inne podejście do tworzenia aplikacji.

Przygotowanie oprogramowania wbudowanego w tradycyjny sposób, wymaga dużego nakładu czasu oraz środków, co przy coraz krótszym czasie życia produktu oraz konieczności szybkiego wprowadzania produktu na rynek jest obecnie nie do zaakceptowania. Aby w rozsądnym czasie podołać współczesnym wymaganiom narzucanym przez rynek, tworząc oprogramowanie dla mikrokontrolerów należy skorzystać z bibliotek zewnętrznych, a dodatkowo pomocne jest również skorzystanie z systemu operacyjnego dedykowanego dla mikrokontrolerów, który znacznie upraszcza strukturę programu.

Prawie dekadę temu, gdy systemy operacyjne nie były jeszcze tak popularne, na łamach Elektroniki Praktycznej przedstawiłem cykl artykułów na temat systemu operacyjnego ISIX mojego autorstwa. Od tego czasu jednak wiele się zmieniło. Ciągły rozwój sprzętu, bibliotek, oprogramowania narzędziowego oraz samego systemu ISIX spowodował, że system doczekał się nowej odsłony w wersji III, która z pierwotną wersją (oprócz założeń) nie ma wiele wspólnego.

Głosy napływające od czytelników zebrane w poprzednim cyklu artykułów spowodowały, że wszystkie przykłady zrealizowane w ramach tego kursu zostaną przygotowane dla popularnych i tanich zestawów fabrycznych produkcji ST, bez konieczności lutowania, kupowania programatorów itp. Potrzebny będzie jedynie zestaw STM32Discovery, kabel USB, umożliwiający dołączenie zestawu do komputera, oraz ewentualnie przejściówka Serial na USB, umożliwiająca dołączenie portu szeregowego mikrokontrolera do komputera PC. Przejściówka będzie potrzebna jednie dla zestawu z STM32F411-DISCO posiadającego zintegrowany programator STLINK w wersji 2.0. Pozostałe zestawy wyposażone w programator STLINK2-1 nie wymagają dołączania zewnętrznego konwertera.

W przykładach zostaną wykorzystane następujące zestawy:

1. **STM411-DISCO + Przejściówka serial na USB pracująca w standardzie napięciowym 3,3 V.**
2. STM32F469-DISCO.
3. STM32F769-DISCO.

Podstawowym zestawem wykorzystywanym do realizacji przykładów będzie zestaw pierwszy, natomiast w przypadku bardziej zaawansowanych przykładach prezentujących biblioteki graficzne czy połączenia sieciowe wykorzystywać będziemy zestaw 2) oraz 3).

Minimalne wymagania systemowe jakie są potrzebne aby uruchomić system operacyjny ISIX w wersji III przedstawiają się następująco:

- procesor: ARM Cortex-M0, -M3, -M4, -M7,
- RAM: 4 kB,
- Flash: 16 kB,
- częstotliwość taktowania rdzenia: 1 MHz.

Naturalnie, aby uzyskać pełną funkcjonalność, na przykład – obsługę sieci czy wyświetlaczy graficznych, będzie potrzebny mikrokontroler o większych zasobach.

Budowa systemu ISIX

System operacyjny ma budowę modułową, dzięki czemu istnieje możliwość użycia tylko niektórych fragmentów systemu w zależności od potrzeb. Poszczególne moduły zostały podzielone na komponenty, które mogą być używane niezależnie (**tabela 1**).

Podstawą systemu ISIX jest system budowania obrazu pamięci Flash mikrokontrolera przygotowany w oparciu o narzędzie do budowania projektów o nazwie WAF (*https://www.waf.io/*). Narzędzie to w porównaniu do GNU-MAKE używanego w poprzednich wersjach systemu, ma szereg zalet:

- Jest przenośne i działa zarówno w systemach Linuksowych jak i pod Windows czy OS-X bez konieczności do uciekania się do różnych „sztuczek" w skryptach.
- Umożliwia budowanie równoległe oraz zawiera standardowe reguły umożliwiający budowanie aplikacji dla mikrokontrolerów.
- Wspiera automatyczny cache dla obiektów bez konieczności budowania całego kodu od początku, oszczędzając czas.
- Dzięki językowi Python umożliwia łatwe tworzenie rozszerzeń, i skryptów, np. skrypty linkera mogą być łatwo generowanie automatycznie w oparciu o meta-dane zawarte w plikach konfiguracyjnych XML, które zawierają informację na temat zasobów danego mikrokontrolera.

Poprzednie części kursu i dodatkowe materiały dostępne są na na stronie www.media.avt.pl

Nazwa	Funkcja
libisix	Jądro systemu operacyjnego ISIX-RTOS
isixwaf	Skrypty systemu budowania WAF specyficzne dla ISIX
libenergymeter	Biblioteka obsługi liczników energii elektrycznej
libfoundation	Biblioteka algorytmów ogólnych
libfsfat	Biblioteka obsługi systemu plików FAT
libgfx	Mini biblioteka GUI obsługująca wyświetlacze LCD
libgsm	Biblioteka obsługi modemów GSM
libperiph	Uniwersalna biblioteka obsługi układów peryferyjnych oraz inicjalizacji mikrokontrolera
libtcpip	Obsługa TCP/IP (stos LWIP)
libtls	Obsługa SSL/TLS (EmbbedTLS)
libusb	Obsługa protokołu USB HOST oraz DEVICE

Tabela 1. Wykaz modułów systemu

- Umożliwia inteligentne znajdowanie plików źródłowych z wykorzystaniem wzorców (glob), które znacząco ułatwiają tworzenie skryptów oraz zarządzanie nimi.

W GNU-MAKE poszczególne skrypty budujące były zawarte w pliku *Makefile*, natomiast w WAF pliki opisujące proces budowania mają nazwę *wscript* i składniowo zgodne są z językiem *python*. Proces budowania aplikacji odbywa w trzech krokach:

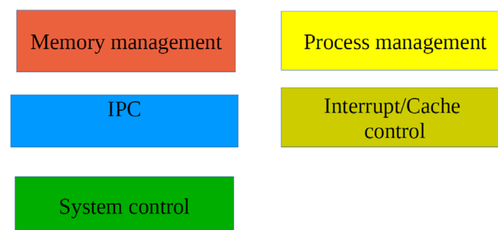
1. Konfiguracji projektu za pomocą polecania *waf configure*, gdzie możemy przekazać dodatkowe parametry konfiguracyjne do projektu za pomocą odpowiednich argumentów np. częstotliwość rezonatora kwarcowego, czy wybór dodatkowych funkcjonalności.
2. Budowania projektu za pomocą polecenia *waf* lub *waf build*.
3. Załadowania projektu do płytki docelowej za pomocą polecenia *waf program*.

Jądro systemu ISIX

Jądro systemu ISIX napisano w języku C z rozszerzeniami GNU w dialekcie C11, a jego kod źródłowy znajduje się w katalogu **libisix**. Pomimo iż podstawowe API wykorzystuję język C, wszystkie biblioteki posiadają dodatkowe klasy umożliwiające korzystanie z API systemu w języku C++. API systemu ISIX zostało podzielone na następujące na bloki funkcjonalne pokazane na **rysunku 1**. Możemy tutaj wyszczególnić wywołania systemowe odpowiedzialne za: zarządzanie pamięcią, zarządzanie procesami (zadaniami), komunikację i synchronizację międzyprocesową IPC, obsługę przerwań i sterowanie pamięciami cache procesora, oraz funkcje odpowiedzialne za kontrolę całości systemu.

Zarządzanie pamięcią

Mikrokontrolery z rdzeniem Cortex M nie mają jednostki zarządzania pamięcią. Niektóre układy mają jedynie uproszczony układ ochrony pamięci MPU, zatem system operacyjny oraz jądro systemu współdzielą wspólną przestrzeń adresową. Z jednej strony jest to zaletą, ponieważ upraszcza komunikację międzyprocesową, z drugiej strony jest wadą, ponieważ błędnie działający proces może uszkodzić jądro systemu i inne procesy. System operacyjny ISIX wykorzystuje funkcjonalność jednostki MPU i jeśli taka jest dostępna, chroni jądro systemu, procesy przed nadpisaniem obszaru stosu danych oraz dostępem do niedozwolonych fragmentów kodu. API zarządzania pamięcią z uwagi na współdzielenie przestrzeni adresowej służy do przydziału pamięci znajdującej się na stercie i może być użyte przez sterowniki oraz procesy użytkownika. Wywołania systemowe związane z obsługą pamięci przedstawia **tabela 2**.



Rysunek 1. API systemu ISIX-RTOS

Funkcje alloc/free/realloc służą do przydziału oraz zwolnienia pamięci na stercie. Działanie tych funkcji jest niedeterministyczne, więc nie powinny one być używane w procedurach obsługi przerwań. Zarządzanie obszarem sterty w systemie ISIX odbywa się za pomocą nowatorskiego algorytmu **TSLF** (*Two Levels Segregated Fit Memory*). Jest to algorytm przeznaczony dla systemów operacyjnych czasu rzeczywistego, który w porównaniu do typowego algorytmu listy wolnych bloków, zapewnia znacznie bardziej deterministyczny czas odpowiedzi oraz znacząco zmniejsza problem fragmentacji, co jest szczególnie istotnie z powodu ograniczonych zasobów i braku jednostki MMU. Wymienione funkcje alokacji pamięci są używane przez standardową bibliotekę języka C, tak więc do alokacji pamięci w celu zapewnienia kompatybilności raczej powinno używać się standardowych funkcji malloc/free oraz new/delete. Odrębną pulę stanowią wywołanie rodziny *mempool_* których zadaniem jest alokowanie oraz zwalnianie stałych bloków pamięci. Alokacja stałych bloków pamięci jest szczególnie przydatna w wypadku sterowników urządzeń, które przekazują dane pakietami o stałej wielkości, na przykład, ramki ethernet lub ramki USB. Funkcje *mempool_alloc*, oraz *mempool_free* mogą być używane w procedurach obsługi przerwań, ponieważ wykazują one złożoność obliczeniową klasy O(1).

Zarządzanie procesami

Zarządzanie procesami w systemie ISIX realizowane jest przez nieskomplikowany zestaw funkcji, który przedstawia się w sposób pokazany w **tabeli 3**.

Funkcje odpowiedzialne za zarządzenie procesami podzielono na kilka mniejszych podgrup.

Pierwszą grupę stanowią wywołania odpowiedzialne za tworzenie i usuwanie zadań. Podczas tworzenia procesu musimy jako argumenty podać wskaźnik do funkcji realizującej dany wątek, argument przekazany do wątku podczas jego uruchomienia, wielkość stosu przydzielonego dla wątku oraz priorytet wątku. Ponieważ procesor nie posiada jednostki zarządzania pamięcią rozmiar

Tabela 2. Wywołania systemowe związane z obsługą pamięci

Nazwa funkcji	Opis
isix::alloc	Alokuje obszar pamięci o rozmiarze n bajtów
isix::free	Zwalnia zaalokowany obszar pamięci
isix::realloc	Zmienia rozmiar zaalokowanego wcześniej obszaru pamięci
isix::heap_stats	Zwraca statystykę na temat wolnej pamięci oraz fragmentacji
isix::mempool_create	Tworzy nową pulę N bloków pamięci o stałym rozmiarze N
isix::mempool_destroy	Usuwa pulę bloków pamięci zwalniając zasoby
isix::mempool_alloc	Alokuje blok pamięci z puli
isix::mempool_free	zwraca blok pamięci do puli

Poprzednie części kursu i dodatkowe materiały dostępne są na stronie www.media.avt.pl

Tabela 3. Zarządzanie procesami

Nazwa funkcji	Opis
ostask_t	Uchwyt identyfikujący proces/zadanie
isix::task_create	Tworzy nowy proces/zdanie
isix::task_kill	Usuwa proces zadanie zwalniając zasoby
isix::task_change_prio	Zmienia domyślny priorytet zadania/procesu
isix::task_get_priority	Pobiera domyślny priorytet zadania procesu
isix::task_get_task_inherited_priority	Pobiera rzeczywisty (dzidziczony) priorytet zadania procesu.
isix::task_self	Pobiera uchwyt do bieżącego zadania/procesu
isix::task_suspend	Usypia określony proces/zadanie do momentu wywołania funkcji resume
isix::task_resume	Wybudza określony proces/zadanie uśpione za pomocą wywołania suspend
isix::task_wait_for	Usypia zadanie do momentu zakończenia innego zadania
isix::wait_ms	Usypia wątek na zadany okres czasu
isix::yield	Wywłaszcza bieżący proces uruchamiając planistę
isix::free_stack_space	Zwraca ilość wolnej pamięci stosu dla danego zadania/procesu
isix::get_task_state	Zwraca stan w jakim znajduje się proces
isix::task_ref	Dodaje nową referencję do zadania/procesu
isix::task_unref	Usuwa referencję z zadania/procesu

stosu musi być znany już na etapie utworzenia zadania, ponieważ wszystkie procesy pracują, we wspólnej przestrzeni adresowej i nie ma możliwości mapowania dodatkowej pamięci do przestrzeni wątku, na żądanie jak ma to miejsce w systemach z MMU.

Kolejna grupa wywołań umożliwia zarządzanie priorytetami wątków. Jeśli jesteśmy przy zarządzaniu priorytetami należy wspomnieć, że domyślnie system ISIX dysponuje 16 priorytetami wątków od 0 do 15, gdzie 0 to jest wartość liczbowa dla najwyższego priorytetu, natomiast 15 to wartość liczbowa symbolizująca najniższy priorytet. Liczba dostępnych priorytetów wątków może zostać zmieniona w systemie na etapie kompilacji poprzez zmianę definicji **CONFIG_ISIX_NUMBER_OF_PRIORITIES.** System ISIX posiada mechanizm dziedziczenia priorytetów, który zapobiega problemowi inwersji priorytetów. Dodatkowa funkcja *get_task_inherited_priority()* pozwala dowiedzieć się z jakim rzeczywistym priorytetem dany proces został zaszeregowany, natomiast *get_task_priority()* umożliwia pobranie domyślnie przydzielonego priorytetu dla procesu/zadania.

Oddzielną grupę stanowią funkcje zarządzania procesami, które umożliwiają usypianie procesów wznawianie wykonania zadań, oraz usypianie procesów do czasu gdy inny proces ulegnie zakończeniu.

Grupa wywołań diagnostycznych umożliwia sprawdzenie jaka ilość miejsca została na stosie, czy sprawdzenie stanu w jakim proces aktualnie znajduje się. W systemie ISIX każde zadanie może znaleźć się w jednym z następujących stanów:

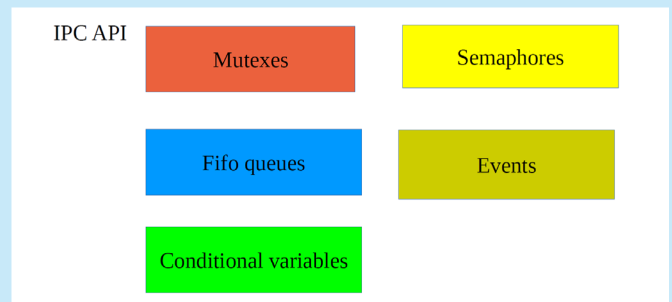
- *READY* – proces gotowy do wykonania i oczekujący na zaszeregowanie.
- *RUNNING* – proces aktualnie wykonywany.
- *CREATED* – proces został utworzony, ale jeszcze nie jest wykonywany.
- *SLEEPING* – proces jest uśpiony i oczekuje na wybudzenie.
- *WTSEM* – proces czeka na semafor .
- *ZOMBIE* – proces zombie, oczekujący na zniszczenie i zwolnienie zasobów.
- *WTEVT* – proces oczekujący na zdarzenie.
- *SUSPEND* – proces uśpiony za pomocą t*ask_suspend*.
- WTMTX – proces oczekujący na mutex
- *WTCOND* – proces oczekujący na zmienną warunkową.
- *WTEXIT* – proces oczekuje na zakończenie innego procesu.

Stan, w którym aktualnie znajduje się wątek jest wykorzystywany przez algorytm szeregujący, celem zakwalifikowania zadań do wykonania, oraz może być również użyty w celach diagnostycznych na etapie uruchamiania oprogramowania uruchamiania oprogramowania.

Przy okazji warto tutaj wspomnieć o interfejsie C++ służącym do tworzenia nowych procesów zadań, które, umożliwia utworzenie kodu zadania zarówno z prostej funkcji, funkcji lambda, jak i dowolnej metody będącej składową klasy. Na przykład, utworzenie nowego wątku (zadania), z wykorzystaniem tego mechanizmu bezpośrednio w konstruktorze klasy może wyglądać jak na **listingu 1**. W konstruktorze klasy jest tworzony wątek, który będzie stanowić metoda klasy *task_tests* o nazwie *thread.*

Komunikacja międzyprocesowa

Istotnym interfejsem systemowym jest blok wywołań systemowych IPC (I*nter-Process Communication*), który jest odpowiedzialny za synchronizację oraz wymianę danych między procesowami, a także synchronizację i wymianę danych pomiędzy procesami, a procedurami obsługi przerwań. W systemie ISIX do dyspozycji mamy kilka popularnych mechanizmów synchronizujących (**rysunek 2**).



Rysunek 2. Komunikacja międzyprocesowa w systemie ISIX

Listing 1. Tworzenie nowego wątku
```
base_task_tests()
    : m_thr( isix::thread_create(std::bind(&base_task_tests::thread,std::ref(*this))))
{
}
```

Poprzednie części kursu i dodatkowe materiały dostępne są na na stronie www.media.avt.pl

Tabela 4. API systemowe związane z semaforami w systemie ISIX	
Nazwa	**Opis**
isix::ossem_t	Typ danych definiujący uchwyt dla semafora
isix::sem_create_limited	Tworzy nowy semafor o wartości początkowej N i wartości maksymalnej M
isix::sem_create	Tworzy nowy semafor o wartości początkowej N
isix::sem_wait	Zmniejsza licznik semafora i ewentualnie usypia zadanie w oczekiwaniu na semafor
isix::sem_signal	Podnosi semafor wybudzając oczekujące zadanie lub zwiększając licznik semafora
isix::sem_signal_isr	Podnosi semafor z poziomu przerwania wybudzając oczekujące zadanie lub zwiększając licznik semafora
isix::sem_reset	Ustawia semafor na zadaną wartość wybudzając wszystkie zadania
isix::sem_getval	Zwraca aktualną wartość semafora
isix::sem_destroy	Kasuje semafor i zwalnia zasoby

Semafory

Semafory są podstawowym mechanizmem umożliwiającym synchronizację dostępu do zasobów tam gdzie zasób dzielony jest na ograniczoną liczbę użytkowników. Na przykład za pomocą semaforów aplikacja może kontrolować maksymalną liczbę otwartych plików. Typowy semafor zaimplementowany jest jako liczba typu całkowitego, która może przyjmować wartości od 0 do ustalonej wartości maksymalnej. W szczególności może być semaforem typu binarnego przyjmującym wartości jedynie z zakresu 0 oraz 1. Semafor skojarzony z danym zasobem początkowo ustawiany jest na wartość dostępnych zasobów danego typu. Proces który chce odwołać się do danego zasobu musi najpierw sprawdzić wartość związanego z tym zasobem semafora. Dodatnia wartość oznacza, że zasób jest dostępny. Przed rozpoczęciem korzystania z danego zasobu proces zmniejsza wartość semafora, a zerowa wartość oznacza, że nie ma wolnych zasobów i proces musi czekać na zwolnienie zasobu przez inny proces aktualnie zajmujący zasób. Kiedy zasób zostanie zwolniony wartość semafora jest zwiększana, a system powiadamia oczekujący proces.

W systemie ISIX możemy tworzyć zarówno semafory zliczające o określonej maksymalnej wartości jak i semafory binarne ustawiając górny limit dla semafora na wartość 1. API systemowe związane z semaforami w systemie ISIX wygląda jak w **tabeli 4**.

Do dyspozycji mamy pełny zestaw funkcji umożliwiający pracę z semaforami. Semafory mogą być wykorzystywane z poziomu procedur obsługi przerwań ISR do notyfikacji procesów/zadań oczekujących na jakieś zdarzenia od układów peryferyjnych. Pozwala to na przekierowanie bardziej czasochłonnych zadań z procedur obsługi przerwań do procesów/zadań znacząco zwiększając responsywność systemu operacyjnego. Do notyfikacji z procedur obsługi przerwań należy używać specjalnych funkcji z sufiksem *_isr.*

Semafory Mutex

Termin mutex pochodzi od angielskiego terminu ***mutually exclusive*** i jest specjalnym rodzajem semafora binarnego, wyposażonym w mechanizm zapobiegający inwersji priorytetów. Dodatkowo, jedynie proces, który założył blokadę może go odblokować, a próba odblokowania przez inny proces kończy się błędem. Idealnym zastosowaniem dla mutexów jest ochrona dostępu do danego zasobu, na przykład, gdy jakaś operacja realizowana przez jeden proces/zadanie nie może być w jednym czasie przerwana przez inny proces. Inwersja priorytetów jest niekorzystnym zjawiskiem, które powoduje że w danej chwili wykonuje się inne zadanie niż to które powinno się wykonywać zgodnie z regułami algorytmu szeregowania. Jako środek zaradczy stosuje się dziedziczenie priorytetów, które polega na tymczasowym podniesieniu priorytetów oczekujących zadań do najwyższego priorytetu ze wszystkich priorytetów oczekujących na te zasoby. W systemie ISIX mamy oddzielne API związane z obsługą muteksów (**tabela 5**).

Tabela 5. API związane z obsługą semaforów mutex	
Nazwa	**Opis**
osmtx_t	Typ danych definiujący uchwyt dla semafora mutex
isix::mutex_create	Tworzy nowy mutex
isix::mutex_lock	Blokuje mutex
isix::mutex_unlock	Odblokowuje mutex
isix::mutex_unlock_all	Zdejmuje blokadę z mutex-a wybudzając wszystkie oczekujące procesy
isix::mutex_destroy	Usuwa mutex zwalniajac zasoby

Do dyspozycji mamy pełny zestaw funkcji potrzebny do pracy z mutexami. Dodatkowo, jest wywołanie *mutex_unlock_all*, które zwalnia mutex, ale oprócz tego wybudza nie tylko jeden, a wszystkie procesy, które oczekują na zasób.

Jak możemy zauważyć w opisie brakuje funkcji z sufiksem _isr, ponieważ funkcje związane z semaforami nie powinny być wywoływane z procedur obsługi przerwań. W kontekście przerwań możemy korzystać jedynie z semaforów. Dzieje się tak ponieważ algorytm przeliczania i dziedziczenia priorytetów jest złożony obliczeniowo i nie jest odpowiedni do tego, aby wywoływać go z kontekstu przerwania.

Rysunek 3. Kolejki FIFO w systemie ISIX

Kolejki FIFO

Kolejki FIFO są wygodnym mechanizmem służącym do komunikacji międzyprocesowej, i mogą być używane do przekazywania danych pomiędzy procesami/zadaniami, jak i pomiędzy procedurami przerwań, a procesami. Kolejka komunikatów w systemie ISIX służy do przekazywania komunikatów o stałej wielkości, a podczas jej tworzenia należy określić maksymalny dopuszczalny rozmiar kolejki. Zasadę działania kolejki przedstawiono na **rysunku 3**. Ilustruje on kolejkę o maksymalnym rozmiarze 7 elementów, która zawiera 5 elementów. *Wątek #1* zapisuje poszczególne elementy do kolejki za pomocą wywołania systemowego *fifo_write()*, natomiast *wątek #2* odczytuje dane z kolejki za pomocą wywołania *fifo_read()*. Każde wywołanie funkcji odczytującej powoduje zwrócenie danej do odczytania lub uśpienie procesu do momentu aż *wątek #1* prześle nowe dane do odczytania. *Wątek #1* zapisujący dane może przejść w stan oczekiwania, jeśli w kolejce nie ma już wolnego miejsca, do czasu aż *wątek #2* odczyta dane i zwolni miejsce w kolejce. Obsługa kolejek FIFO realizowana jest przez następujące API systemowe (**tabela 6**).

Poprzednie części kursu i dodatkowe materiały dostępne są na stronie www.media.avt.pl

Tabela 6. Obsługa kolejki FIFO

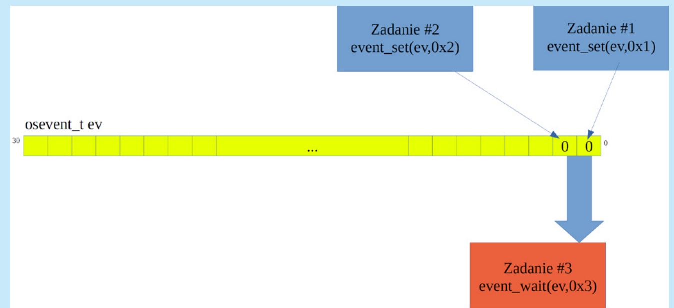
Nazwa	Opis
osfifo_t	Typ uchwytu danych reprezentujący kolejkę FIFO
isix::fifo_create	
isix::fifo_create_ex	Tworzy nową kolejkę zawierającą maksymalnie N elementów o rozmiarze M
isix::fifo_write	Zapisuje daną do kolejki FIFO.
isix::fifo_write_isr	Zapisuje daną do kolejki FIFO w kontekście przerwania
isix::fifo_read	Odczytuje daną z kolejki FIFO
isix::fifo_read_isr	Odczytuje daną z kolejki FIFO w kontekście przerwania
isix::fifo_count	Zwraca ilość elementów umieszczonych w kolejce
isix::fifo_destroy	Usuwa kolejkę FIFO i zwalnia zajmowane zasoby

Funkcje z sufiksem *_isr*, są funkcjami nieblokującymi i mogą być wywoływane w kontekście przerwania. Jeśli nie ma wolnego miejsca lub danych do odczytania, to zamiast uśpienia w oczekiwaniu na daną lub wolne miejsce jest zwracany kod błędu informujący o wystąpieniu takiej sytuacji. Kolejki FIFO są obiektami jednokierunkowymi, więc jeśli chcemy zrealizować komunikację dwukierunkową musimy użyć oddzielnych kolejek. Warto również wspomnieć, o tym że wszystkie zadania pracują we wspólnej przestrzeni adresowej, a zatem możemy za pomocą kolejek przekazywać wskaźniki do innych obszarów pamięci.

Zdarzenia bitowe (Events)

Zdarzenia bitowe są mechanizmem częściej spotykanym w systemach operacyjnych wbudowanych, niż w typowych systemach ogólnego przeznaczenia. Zdarzenia umożliwiają oczekiwanie na określone zdarzenie lub grupę zdarzeń polegające na ustawieniu odpowiedniego bitu lub grupy bitów w jednym słowie. W systemie ISIX zdarzenie przechowuje maskę bitową w zmiennej o długości 31 bitów, tak więc w tym czasie jeden proces/zadanie może czekać na 31 zdarzeń jednocześnie. Zasadę działania tego mechanizmu przedstawiono na **rysunku 4**.

Zadanie #3 wywołuje funkcję *event_wait()*, z argumentem *0x3*, co powoduje, że to zdanie zostanie uśpione do momentu ustawienia bitów o numerze porządkowym 0 oraz 1. Oba bity są wyzerowane, więc *zadanie #3* jest usypiane. *Zadanie #1* w wyniku zrealizowania jakiejś czynności ustawia bit o numerze porządkowym 0 za pomocą wywołania *event_set()* informując, iż jakaś czynność została wykonana. Podobnie po wykonaniu żądanej czynności, *zadanie #2* ustawia bit o numerze porządkowym 1 informując o zakończeniu wykonywania określonej czynności. W tym momencie, zdarzenie bitowe przyjmuje wartość 0x03, co spełnia kryterium wznowienia wykonywania zadania #3, zatem po spełnieniu tych warunków system operacyjny wznowi wykonanie zadania *zadanie #3* kończąc wywołanie



Rysunek 4. Działanie mechanizmu zdarzeń bitowych w systemie ISIX

systemowe *event_wait()*. Należy tutaj wspomnieć, że funkcja event_wait(), może oczekiwać zarówno na ustawienie wszystkich wymaganych bitów, jak i tylko jednego bitu z zadanej grupy w zależności od parametru *wait_for_all* typu bool.

System ISIX posiada również dodatkowe API które umożliwia skojarzenie danego bitu w danym evencie z dowolną kolejką FIFO, co może być wykorzystane np. do oczekiwania na zdarzenie odczytu danych z kilku kolejek przez jeden proces. Korzystając powyższego mechanizmu możemy łatwo osiągnąć podobną funkcjonalność jak w przypadku korzystania z wywołań systemowych poll/epoll w systemach zgodnych ze standardem POSIX.

API związane ze zdarzeniami bitowymi w systemie ISIX przedstawia się jak w **tabeli 7**. Podobnie jak w poprzedniej grupie wywołań funkcje z sufiksem isr mogą być używane w procedurach obsługi przerwań, natomiast pozostałe nie mogą być używane w kontekście przerwania. Warto tutaj wspomnieć o wywołaniu *event_sync*, które w sposób atomowy oczekuje na grupę bitów *bits_to_wait* a następnie ustawia grupę bitów na wartość *bits_to_set*, co może być przydatne w bardziej zaawansowanych problemach synchronizacyjnych.

Zarządzanie przerwaniami oraz pamięcią cache

Istotnym aspektem z punktu sterowników urządzeń jest zarządzanie przerwaniami, oraz pamięciami cache procesora. W pierwotnej wersji systemu ISIX nie było, żadnych wywołań dedykowanych do tego celu, tak więc zarządzenie kontrolerem przerwań było realizowane przez biblioteki zewnętrzne dedykowane dla danego mikrokontrolera. Wraz wprowadzeniem pamięci cache dla procesorów CORTEX-M7 do systemu ISIX wprowadzono wewnętrzne API obsługujące kontroler przerwań, oraz zarządzanie pamięciami cache przydatne, podczas pisania ujednoliconych sterowników urządzeń.

Kontroler przerwań

Wywołania systemowe dedykowane obsłudze kontrolera przerwań możemy podzielić na dwie grupy. Jedną grupę stanowią funkcje generyczne, natomiast drugą grupę stanowią funkcje specyficzne dla danego kontrolera. W wypadku rdzenia Cortex-M będą to funkcje specyficzne dla kontrolera przerwań NVIC (*Nested Vectorized Interrupt Controller*).

REKLAMA

KITy AVT

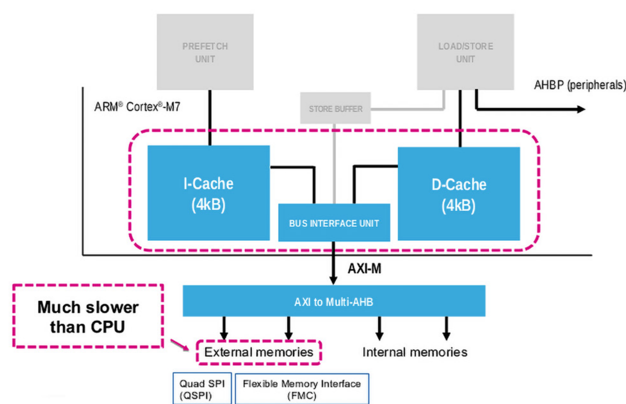
O KITach AVT przeczytasz również na Facebooku

http://bit.ly/2BjVMN7

Poprzednie części kursu i dodatkowe materiały dostępne są na na stronie www.media.avt.pl

Tabela 7. API związane ze zdarzeniami bitowymi

Nazwa	Opis
osevent_t	Typ reprezentujący obiekt zdarzeń bitowych
isix::event_create	Tworzy nowy obiekt zdarzeń bitowych
isix::event_destroy	Usuwa obiekt zdarzeń bitowych zwalniając zajęte przez niego zasoby
isix::event_sync	Oczekuje na grupę bitów zdefiniowaną przez wait_for a następnie atomowo ustawie grupę bitów bits_to_sets
isix::event_wait	Oczekuje na grupę bitów lub na dowolny bit
isix::event_clear	Zeruje grupę bitów
isix::event_clear_isr	Zeruje grupę bitów z kontekstu przerwania
isix::event_set	Ustawia grupę bitów, wybudzając ewentualnie zadania po spełnieniu warunku oczekiwania
isix::event_set_isr	Ustawia grupę bitów z poziomu przerwania, wybudzając ewentualnie zadania po spełnieniu warunku oczekiwania
isix::event_get	Pobiera wartość grupy bitów.
isix::event_get_isr	Pobiera wartość grupy bitów w kontekście przerwania.
isix::ifo_event_connect	Łączy wybraną kolejkę FIFO z wybranym bitem zdarzeń
isix::fifo_event_disconnect	Rozłącza wybraną kolejkę FIFO z wybranym bitem zdarzeń



Rysunek 5. Pamięć cache L1 w rdzeniu Cortex-M7

API przeznaczone do ogólnej obsługi przerwań umożliwia odblokowanie oraz zablokowanie wybranego kanału przerwań, czy globalne włączenie albo wyłącznie wszystkich przerwań (**tabela 8**). Pierwszą grupę stanowią wywołania odpowiedzialne za globalne włączenie albo wyłączenie systemu przerwań, mamy też możliwość atomowego zapisania poprzedniego stanu przerwań globalnych, w zmiennej lokalnej wraz z jego wyłączeniem. Istnieje również możliwość odtworzenia poprzedniego zapisanego stanu przerwań globalnych. Drugą grupę stanowią funkcje, umożliwiające włączenie lub wyłączenie wybranej linii przerwania, sprawdzenie stanu zgłoszenia przerwania, czy ustawienie lub wyzerowanie wybranej linii zgłoszenia przerwania.

Jako argument do wszystkich funkcji należy podać numer porządkowy wybranej linii, który jest specyficzny dla platformy. API obsługi przerwań specyficzne dla platformy umożliwia ustawienie różnych funkcji systemu przerwań które są specyficzne dla danej architektury. W przypadku mikrokontrolerów z rdzeniem Cortex-M3/4/7 zestaw funkcji specyficznych dla kontrolera NVIC zaprezentowano w **tabeli 9**.

Dodatkową funkcjonalność stanowią funkcje umożliwiające wyłączenie (maskowanie) przerwań ale tylko do wartości priorytetu przekazanego jako argument. Podobnie jak w funkcjach podstawowych możemy maskować przerwania z zachowaniem poprzedniego stanu maski lub bez. Mamy również możliwość ustawienia priorytetów poszczególnych przerwań za pomocą funkcji *set_irq_priority*(), co umożliwia wykonywanie czasowo krytycznych przerwań poza kolejnością. Mamy również możliwość ustawiania flag zgłoszeń poszczególnych przerwań czy możliwość ustalenia ile bitów z wartości liczbowej priorytetu będzie wykorzystanie jako priorytet, a ile jako podpriorytet przerwań. Więcej szczegółów możemy znaleźć w dokumentacji technicznej rdzenia Cortex-M.

Zarządzanie pamięciami CACHE procesora

Najbardziej zaawansowane technicznie mikrokontrolery, pracujące z częstotliwością taktowania rdzenia rzędu kilkuset MHz, mogą być wyposażone w dodatkową pamięć cache, która znacząco przyśpiesza dostęp do danych znajdujących się w wolniejszej pamięci Flash czy RAM. Ten mechanizm jest znany głównie z większych mikroprocesorów, a w świecie procesorów ARM Cortex-M został w Cortex-M7, który zamiast do magistrali AHB jest dołączony do magistrali AXI znanej z procesorów aplikacyjnych Cortex-A. Schemat blokowy pamięci Cache procesora pokazano na **rysunku 5**.

Tabela 8. API przeznaczone do obsługi przerwań

Nazwa	Opis
void isix::irq_enable();	Włącza przerwania globalnie
void isix::irq_disable();	Wyłącza przerwania globalnie
unsigned isix::irq_save(void);	Wyłącza przerwania i zapisuje aktualny stan przerwań
void isix::irq_restore(unsigned mask);	Odtwarza aktualny stand przerwań globalnych
void isix::request_irq(int irqno);	Włącza wybraną linię przerwania
void isix::free_irq(int irqno);	Wyłącza wybraną linię przerwania
bool isix_get_irq_enabled(int irqno);	Sprawdza aktualny stan wybranej linii przerwania
bool isix::get_irq_pending(int irqno);	Sprawdza czy wybrane zgłoszenie przerwania jest aktywne
void isix::set_irq_pending(int irqno);	Ustawia wybraną flagę zgłoszenia przerwania
void isix::clear_irq_pending(int irqno);	Kasuje wybraną flagę zgłoszenia przerwania

Poprzednie części kursu i dodatkowe materiały dostępne są na stronie www.media.avt.pl

Tabela 9. Funkcje specyficzne dla kontrolera NVIC

Nazwa	Opis
void isix::mask_irq_priority(isix_irq_prio_t priority);	Maskuje przerwania o priorytecie niższym niż zadany
isix::irq_raw_prio_t isix::mask_irq_save_priority(isix_irq_prio_t new_prio);	Maskuje przerwania o priorytecie niższym niż zadany i zapisuje poprzedni stan maski
isix::mask_irq_restore_priority(isix_irq_raw_prio_t prio)	Odtwarza poprzedni stan maskowania przerwań
void isix::umask_irq_priority(void)	Wyłącza maskę przerwań odblokowując przerwania
void isix::set_irq_priority(int irqno, isix_irq_prio_t priority);	Ustawia priorytet przerwań <prio,subprio>
void isix::set_raw_irq_priority(int irqno, isix_irq_raw_prio_t prio);	Ustawia priorytet przerwań używając typu wewnętrznego
isix_irq_raw_prio_t isix::irq_priority_to_raw_priority(isix_irq_prio_t prio);	Konwertuje priorytet przerwań <prio,subprio> do typu wewnętrznego
bool isix::get_active_irq(int irqno);	Sprawdza czy wybrane przerwanie jest aktywne
void isix::generate_software_interrupt(int irqno);	Generuje przerwanie programowe
void isix::event_irq_pending(bool en);	Ustawia kontroler tak by dane przerwanie generowało wewnętrzny sygnał EV dla procesora
void isix::set_irq_priority_group(enum isix_cortexm_prigroup prigroup);	Ustawia ilość bitów jaka będzie wykorzystana jako priorytet oraz ilość bitów która będzie wykorzystania jako podpriorytet przerwania.
void isix::set_irq_vectors_base(const void *vectptr);	Ustawia adres bazowy wektora przerwań.

Tabela 10. API służące do obsługi pamięci cache

Nazwa	Opis
void isix::icache_enable(bool yes);	Włącza lub wyłącza pamięć cache kodu
void isix::dcache_enable(bool yes);	Włącza lub wyłącza pamięć cache danych
void isix::inval_dcache(void);	Unieważnia całą pamięć cache danych
void isix::inval_icache(void);	Unieważnia całą pamięć cache kodu
void isix::clean_dcache(void);	Czyści całą pamięć cache danych
void isix::clean_inval_dcache(void);	Unieważnia i czyści całą pamięć cache danych
void isix::inval_dcache_by_addr(void *addr, size_t dsize);	Unieważnia wybrany fragment pamięci cache danych
void isix::clean_dcache_by_addr(void *addr, size_t dsize);	Czyści wybrany fragment pamięci cache danych
void isix::clean_inval_dcache_by_addr(void *addr, size_t dsize);	Unieważnia i czyści wybrany fragment pamięci danych

Rdzeń Cortex-M7 ma po 4 kB pamięci cache pierwszego poziomu (L1), osobno dla kodu oraz osobno dla danych. Pamięć cache jest zazwyczaj przeźroczysta dla oprogramowania, jednak przy tworzenia sterowników urządzeń korzystających z DMA jest konieczna odpowiednia synchronizacja pamięci cache z pamięcią główną. Podobny sytuacja zachodzi dla pracy wieloprocesorowej, jednak w świecie mikrokontrolerów najczęściej takich konfiguracji się nie spotyka.

System ISIX wprowadził API przeznaczone dla obsługi pamięci cache wraz z pojawieniem się rdzenia Cortex-M7. Jeśli używamy mikrokontrolera z rdzeniem Cortex-M0/M3/M4 niemającego pamięci cache, wówczas funkcję związane z obsługą cache są puste i nie wykonują żadnych czynności. Niezależnie jednak od tego czy korzystamy z rdzenia posiadającego pamięć cache, czy nie, w sterownikach urządzeń korzystających z kontrolera DMA zawsze należy ich używać, aby uzyskać kod który działa na dowolnym rdzeniu. API służące do obsługi pamięci cache przedstawiono w **tabeli 10**.

API zawiera podstawowe wywołania umożliwiające włączenie lub wyłączenie pamięci **I-Cache** (pamięć cache kodu) oraz **D-Cache** (pamięć cache danych), oraz funkcje unieważniające lub czyszczące pamięć **I-Cache** oraz **D-Cache.** W przypadku pamięci D-Cache mamy możliwość unieważnienia lub wyczyszczenia linii pamięci cache, które wskazują na określony przedział adresów fizycznych wyznaczonych przez adres początkowy oraz rozmiar obszaru, bez konieczności usuwania całości. Warto tutaj wspomnieć czym różni się operacja unieważnienia od operacji czyszczenia pamięci cache. Operacja unieważnienia unieważnia dane jakie znajdują się w pamięci cache i traktuje tak jakby tych danych tam nie było, natomiast operacja czyszczenia przepisuje zawartość pamięci cache do pamięci głównej. Operację unieważnienia stosujemy np. po zakończeniu transmisji DMA, gdy obszar pamięci zmienił się poza kontrolą procesora, natomiast operację czyszczenia należy zastosować np. przed transmisją DMA, aby dane odczytane przez kontroler DMA z pamięci były prawidłowe.

Lucjan Bryndza, EP
lucjan.bryndza@boff.pl

REKLAMA

Wstąp do Klubu AVT Elektronika – będziesz miał prawo do korzystania z szeregu przywilejów

http://bit.ly/2GaDwtQ

Poprzednie części kursu i dodatkowe materiały dostępne są na na stronie **www.media.avt.pl**