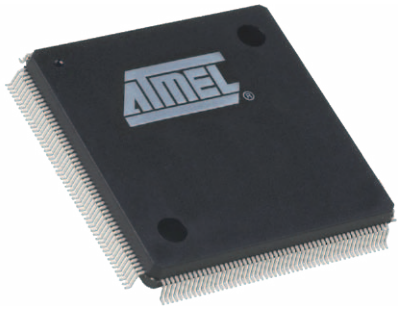
ISIX-RTOS
Minisystem operacyjny dla mikrokontrolerów 32-bitowych

W artykule przedstawiamy oprogramowanie spełniające rolę prostego systemu operacyjnego, pozwalającego na jednoczesną realizację kilku zadań bez ryzyka wzajemnego zakłócania ich działania. Prostota korzystania z ISIX-a i jego niewielkie wymagania wobec sprzętu pozwalają stosować go na wielu 32-bitowych mikrokontrolerach.



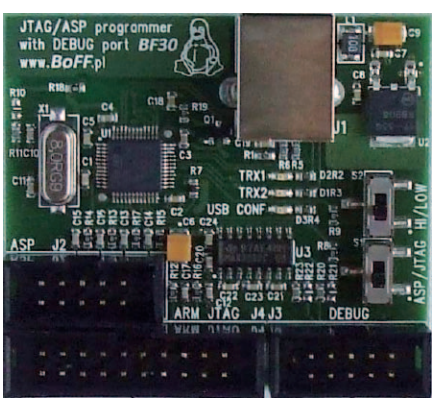
Dodatkowe informacje:
ISIX-RTOS można pobrać ze strony domowej projektu *http://bryndza.boff.pl/index.php?dz=rozne&id=isixrtos*

Istnieje wiele systemów operacyjnych przeznaczonych dla mikrokontrolerów. Większość z nich cechuje się rozbudowanym API, skomplikowaną konfiguracją uzależnioną od platformy, a czasami dużym rozmiarem kodu. Systemy te również najczęściej narzucają sposób pisania aplikacji. Cechy te skłoniły mnie do opracowania systemu ISIX-RTOS zawierającego proste API, którego można się nauczyć w krótkim czasie. ISIX-RTOS jest biblioteką obsługi wątków i komunikacji międzywątkowej dla mikrokontrolerów, nienarzucającą zupełnie sposobu pisania programu czy korzystania z ulubionych bibliotek programisty. System powstał z myślą o 32-bitowych mikrokontrolerach **ARM** (dostępny jest port dla rdzenia Cortex-M3) oraz możliwości pisania aplikacji w języku C++.

Budowa systemu ISIX-RTOS

ISIX-RTOS ma budowę modułową, dzięki czemu jego jądro jest niezależne od peryferii mikrokontrolera. ISIX-RTOS składa się z trzech bibliotek linkowanych statycznie:

- *libisix* (*isix-1.0.tar.gz*) – jądro systemu/biblioteka obsługi wątków.
- *libfoundation* (*libfoundation-1.0.tar.gz*) – biblioteka użytecznych funkcji systemowych niezależna od platformy, zawierająca lekką implementację funkcji bibliotecznych C/C++ np. *printf* oraz implementacja funkcji niezbędnych do prawidłowego kompilatora działania C++.
- *libstm32* (*libstm32-1.0.tar.gz*) – biblioteka uruchomieniowa specyficzna dla danej rodziny mikrokontrolerów zawierająca obsługę układów peryferyjnych, skrypty linkera, skrypty OpenOCD do obsługi interfejsu JTAG (m.in. opracowany przez firmę Boff.pl interfejs BF30 – **fot. 1**).



Fot. 1. Wygląd interfejsu JTAG BF30

Modułowa budowa upraszcza strukturę systemu i uniezależnia jądro systemu od funkcji specyficznych dla danego typu mikrokontrolera. Umożliwia również wykorzystanie poszczególnych części zupełnie niezależnie: na przykład biblioteka *libstm32* może być użyta w oddzielnym projekcie niekorzystającym z systemu. Takie podejście sprzyja wielokrotnemu użyciu tego samego kodu w wielu projektach i zmniejszeniu liczby błędów.

Biblioteka systemu operacyjnego *libisix* nie inicjalizuje żadnych układów peryferyjnych mikrokontrolera (poza rdzeniem). Wszystkie funkcje specyficzne znajdują się w bibliotece *libstm32* i muszą być wywołane przez aplikację użytkownika.

Opis funkcji API

Scheduler (plik nagłówkowy *scheduler.h*):

```
void isix_init(prio_t num_priorities);
```

Funkcja ta inicjalizuje system ISIX-RTOS. Jako argument przyjmuje maksymalną liczbę priorytetów zadań używanych przez scheduler. Zadanie o najwyższym priorytecie ma priorytet 0, zadanie o najniższym priorytecie ma priorytet *num_priorities-1*. Każdy dodatkowy priorytet używany przez system zmniejsza liczbę wolnego RAM-u na stercie o 16 bajtów. W systemie może być dowolna liczba zadań o takim samym priorytecie, które są szeregowane według algorytmu karuzelowego *Round-Robin*.

```
prio_t isix_get_min_priority(void);
```

Funkcja pobiera minimalny dopuszczany numer priorytetu w systemie (zadanie o najniższym priorytecie)

```
void isix_start_scheduler(void)
__attribute__((noreturn));
```

Funkcja uruchamia system i rozpoczyna szeregowanie zadań, jest to ostatnia funkcja wywoływana z funkcji głównej *main()*.

```
void isix_bug(void);
```

Wywołanie tej funkcji powoduje zatrzymanie systemu operacyjnego. Powinna być ona wywoływana w wyniku wystąpienia krytycznego błędu. W przypadku, gdy aplikacja skompilowana jest w trybie debug, powoduje to wypisanie na terminalu dodatkowych informacji diagnostycznych.

```
tick_t isix_get_jiffies(void);
```

Funkcja zwraca liczbę cykli zegarowych od momentu włączenia systemu. Długość cyklu zależy od częstotliwości przerwania zegarowego, którą określa stała *ISIX_HZ*.

```
static inline void isix_yield() { port_yield(); }
```

ISIX-RTOS – klasyczny scheduler z priorytetowaniem zadań – charakteryzuje się następującymi cechami:
- dowolna liczba priorytetów dla zadań (ograniczona jedynie pojemnością pamięci RAM),
- wywłaszczanie zadań oraz algorytm *round-robin* dla zadań o takim samym priorytecie,
- komunikacja międzyprocesowa oparta o semafory i kolejki FIFO,
- wsparcie dla języka C++ oraz wrappery klas dla języka C++,
- inicjalizacja peryferii mikrokontrolera poza systemem,
- małe wymagane zasoby sprzętowe.

Funkcja powoduje oddanie sterowania oraz przeprowadzenie ponownego zaszeregowania zadań.

Zarządzanie zadaniami/wątkami (plik nagłówkowy *task.h*):

```
task_t* isix_task_create(task_
func_ptr_t task_func, void *func_
param, unsigned long stack_depth,
prio_t priority);
```

Funkcja tworzy nowe zadanie (wątek). Jako pierwszy argument *task_func* przyjmuje wskaźnik do funkcji tworzącej zadanie (wątek), o następującej sygnaturze:

```
void task_func(void *arg) __
attribute__((noreturn));
```

Drugi argument *func_param* określa argument przekazany do funkcji zadania/wątku, który można odczytać z tej funkcji za pomocą argumentu *arg*. Argument *stack_depth* określa wielkości stosu dla danego zadania (wątku). Minimalną dopuszczalną wielkość pamięci określa stała *ISIX_MIN_STACK_DEPTH*. Argument *priority* określa priorytet przydzielony dla tworzonego zadania. Funkcja zwraca wskaźnik na strukturę kontrolną zadania *task_t** lub *NULL* w przypadku wystąpienia błędu.

```
static inline int isix_task_
change_prio( task_t* task, prio_t
new_prio );
```

Funkcja pozwala zmienić bieżący priorytet przydzielonego zadania na inny z dopuszczalnego zakresu. Jako argument przyjmuje wskaźnik do struktury kontrolnej zadania oraz nowy priorytet zadania. Funkcja ta zwraca kod błędu lub *ISIX_EOK (0)*. Pozostałe kody błędów znajdują się w pliku nagłówkowym *error.h*.

```
int isix_task_delete(task_t
*task);
```

Funkcja powoduje zatrzymanie zadania (wątku) przekazanego jako argument *task* oraz jego usunięcie. Funkcja ta zwraca kod błędu.

Do komunikacji międzyprocesowej służą semafory, których API zawarto w pliku nagłówkowym *semaphore.h*:

```
sem_t* isix_sem_create(sem_t
*sem,int val);
```

Funkcja tworzy (gdy argument *sem* ma wartość *NULL*) lub inicjalizuje semafor **sem* i przypisuje mu wartość początkową *val*. Funkcja zwraca wskaźnik do struktury kontrolnej semafora lub *NULL* w przypadku niepowodzenia.

```
int isix_sem_wait(sem_t *sem,
tick_t timeout);
```

Funkcja zmniejsza wartość semafora o 1, gdy jego wartość jest większa od 0, w przypadku gdy wartość semafora jest równa 0, powoduje uśpienie bieżącego wątku (zadania) do momentu podniesienia semafora lub do upłynięcia czasu *timeout*. W przypadku, gdy argumentowi *timeout* przypisano wartość *ISIX_TIME_INFINITE*, wątek ten czeka bezwarunkowo do chwili podniesienia semafora. Kod błędu *ISIX_EOK* oznacza, że funkcja powróciła w wyniku podniesienia semafora. Kod błędu *ISIX_ETIMEOUT* oznacza wystąpienia przeterminowania.

```
int isix_sem_get_isr(sem_t *sem);
```

Funkcja jest odpowiednikiem *sem_wait*, która może być wywoływana z kontekstu procedury obsługi przerwania, co wynika z niemożności odroczenia procedury obsługi przerwania.

```
static inline int isix_sem_
signal(sem_t *sem);
static inline int isix_sem_
signal_isr(sem_t *sem);
```

Te funkcje powodują podniesienie semafora poprzez zwiększenie jego zawartości o 1 oraz ewentualne wybudzenie oczekującego zadania (procesu). Wersja funkcji z sufiksem *_isr* może być wywoływana z kontekstu procedury obsługi przerwania. W przypadku powodzenia zwraca wartość *ISIX_EOK (0)*.

```
int isix_sem_destroy(sem_t *sem);
```

Funkcja powoduje usunięcie semafora przekazanego jako argument *sem* oraz zwolnienie zasobów zajmowanych przez ten semafor. W przypadku powodzenia zwraca wartość *ISIX_EOK*.

```
tick_t isix_ms2tick(unsigned long
ms);
```

Przy okazji tematyki C++ chciałbym poruszyć temat mitu, który panuje w wielu kręgach, według którego rzekomo C++ nie nadaje się do pisania oprogramowania na mikrokontrolery i jest on przeznaczony jedynie dla większych systemów komputerowych np. komputerów PC. Mity te nie mają wiele wspólnego z rzeczywistością i wynikają raczej z nieznajomości C++ czy samych opcji kompilatora. Głównym zarzutem kierowanym pod kątem języka C++ jest rzekome twierdzenie, że aplikacja napisana w C++ potrzebuje dużo więcej pamięci RAM i FLASH niż aplikacja napisana w języku C. Drugim mitem jest zarzut pod kątem szybkości działania aplikacji. Język C++ jest bardzo potężnym narzędziem mającym wiele możliwości niedostępnych w języku C: enkapsulacja, dziedziczenie, polimorfizm, wyjątki, wzorce, RTTI ,biblioteka STL. Nieumiejętne korzystanie rzeczywiście może powodować „puchnięcie" kodu, co nie jest wadą języka C++, a raczej nieumiejętnym korzystaniem z jego możliwości. Omówimy, w jaki sposób uniknąć problemów i uzyskać podobny rozmiar aplikacji jak w języku C.

1. Enkapsulacja – inaczej ukrywanie danych polegające na ukrywaniu metod lub danych składowych klasy tak, aby były one dostępne jedynie dla metod wewnętrznych lub funkcji zaprzyjaźnionych. Ukrywanie danych wykonywane jest tylko i wyłącznie na etapie kompilacji i koszt w czasie wykonania jest zerowy. W podobny sposób ukrywamy dane w obrębie danego modułu w języku C za pomocą słowa kluczowego *static*.
2. Dziedziczenie – jest operacją stworzenia nowej klasy na podstawie klasy już istniejącej. Dziedziczenie wykonywane jest na etapie kompilacji, trudno więc mówić o jakimkolwiek dodatkowym koszcie.
3. Polimorfizm – czyli wielopostaciowość, to mechanizm pozwalający programiście używać metod na kilkanaście różnych sposobów. Polimorfizm metody w języku C++ realizowany jest za pomocą słowa kluczowego *virtual*. Ma on pewien koszt, ponieważ w momencie wywołania metody wirtualnej linker nie zna adresu tej metody, a więc musi być on ustalony w czasie wykonania. Adresy metod wirtualnych przechowywane są w specjalnej tablicy *vtable* (*virtual table*). Wywołanie funkcji wirtualnej charakteryzuje się dodatkowym kosztem wywołania funkcji na podstawie offsetu w tablicy *vtable*, co stanowi dodatkowy koszt wywołania pośredniego. Na przykład wywołanie metody wirtualnej
ptr->mw(a,b);
kompilator przetłumaczy jako:
ptr->vtable[MW_OFFSET](a,b);
co stanowi bardzo niewielki koszt. Musimy pamiętać że podobna konstrukcja w języku wymaga używania logiki warunkowej *if..else* lub *switch..case(),* co również stanowi dodatkowy koszt co do rozmiaru kodu oraz narzut wykonania. Dodatkowo na niekorzyść języka C przemawia to, że taki kod występuje najczęściej w wielu miejscach w programie, co stanowi dodatkowe obciążenie. Reasumując: polimorfizm ma pewien dodatkowy koszt, jednak implementacja podobnej funkcjonalności w języku C jest równie, a nawet bardziej kosztowna i podatna na błędy. Poza tym gdy nie potrzebujemy polimorfizmu, nie używamy słowa kluczowego *virtual* i wówczas adres metody ustalany jest na etapie wykonania.
4. Wyjątki – są mechanizmem zmiany przepływu sterowania w języku C++ służącym do obsługi zdarzeń wyjątkowych, a w szczególności sytuacji błędnych. Takie wyjątki w C++ mają dość duży narzut, głównie na zajętość pamięci programu wynikający z implementacji. Nic nie każe nam jednak z nich korzystać, wystarczy do opcji kompilatora C++ podać flagę *-fno-exceptions* i używać kodów błędów podobnie jak w języku C.
5. Wzorce – są mechanizmem tworzenia funkcji/klas wzorcowych bez uwzględniania typów, na których ten kod operuje. Funkcje te mają koszt związany z rozmiarem, ponieważ są specjalizowane w miejscu wywołania, jednak umiejętne ich używane, np. wykorzystanie dziedziczenia z innych klas bazowych, pozwala zminimalizować narzut. Podobny mechanizm generyczny w języku C implementujemy za pomocą makr preprocesora i nikt przy tym bardzo nie prostestuje.
6. RTTI (*Run Time Type Information*) – informacja o typie w trakcie wykonania polega na dołączeniu do kodu dodatkowych informacji o typach. W C++ RTTI ma dodatkowy koszt związany z wielkością kodu wynikowego oraz pamięci RAM zawierającej dodatkową strukturą przechowującą nazwę klasy. Jednak w większości aplikacji mechanizm RTTI jest zbędny i można go wyłączyć, dodając do kompilatora flagę *-fno-rtti*.
7. Biblioteka STL (*Standard Template Library*) – biblioteka standardowa C++ zawierająca algorytmy, pojemniki, iteratory oraz inne funkcje w postaci szablonów. Biblioteka STL może powodować znaczne zwiększenie kodu programu, więc wskazana jest duża ostrożność podczas jej wykorzystywania. Jednak przy odrobinie znajomości biblioteki STL nawet i w aplikacjach przeznaczonych dla mikrokontrolerów można korzystać z jej dobrodziejstwa, np. klasa *vector*, algorytmy np. *for_each* itp.

Funkcja powoduje przeliczenie liczby milisekund przekazanych jako argument *ms* na liczbę cykli systemu operacyjnego

```
static inline int isix_
wait(tick_t timeout);
```

Funkcja usypia zadanie (wątek) na określoną liczbę cykli systemu. W przypadku powodzenia po zakończeniu oczekiwania zwraca wartość *ISIX_EOK*.

Inną możliwością komunikacji międzyprocesowej (międzywątkowej) są kolejki FIFO umożliwiające przekazywanie danych pomiędzy zadaniami:

```
fifo_t* isix_fifo_create(int n_
elem, size_t elem_size);
```

Funkcja tworzy kolejkę FIFO na *n_elem* elementów, o wielkości każdego elementu *elem_size*. W przypadku powodzenia funkcja zwraca wskaźnik na strukturę kontrolną *fifo_t*, natomiast w przypadku niepowodzenia zwraca wartość *NULL*.

```
int isix_fifo_write(fifo_t *fifo,
const void *item, tick_t
timeout);
int isix_fifo_write_isr(fifo_t
*queue, const void *item);
```

Funkcje zapisują do kolejki o argumencie *fifo* element o wartości przekazanej za pomocą argumentu *item*. Wersja pierwsza w przypadku gdy kolejka FIFO jest zapełniona, powoduje uśpienie zapisującego procesu (wątku) na czas przekazany jako argument *timeout* lub do momentu zwolnienia miejsca w kolejce. Wersja z przyrostkiem *_isr* służy do wywoływania z kontekstu przerwania i nie powoduje zablokowania. W przypadku powodzenia zwraca wartość *ISIX_EOK*.

```
int isix_fifo_read(fifo_t *fifo,void
*item, tick_t timeout);
int isix_fifo_read_isr(fifo_t
*queue, void *item);
```

Funkcje te są analogiczne do wcześniej opisanych funkcji *isix_fifo_write()* i powodują odczytanie danych z kolejki FIFO.

```
int isix_fifo_count(fifo_t *fifo);
```

Funkcja zwraca liczbę elementów znajdujących się aktualnie w kolejce lub kod błędu, gdy wartość zwracana jest mniejsza od 0.

```
int isix_fifo_destroy(fifo_t *fifo);
```

Funkcja powoduje skasowanie kolejki FIFO oraz zwolnionych przez nią zasobów. W przypadku powodzenia zwraca wartość *ISIX_EOK (0)*.

Funkcje alokacji pamięci na stercie systemu, zawarte w pliku nagłówkowym *memory.h*:

```
void* isix_alloc(size_t size);
```

Funkcja powoduje alokację *size* bajtów pamięci na stercie systemowej. W przypadku powodzenia zwraca wskaźnik do zaalokowanej pamięci, w przypadku niepowodzenia zwraca wartość NULL.

```
void isix_free(void *mem);
```

Funkcja ta powoduje zwolnienie pamięci na stercie systemowej (argument *mem*), zaalokowanej wcześniej za pomocą funkcji *isix_alloc*.

Wrappery klas C++ dla ISIX-a. Kilka słów o języku C++ dla mikrokontrolerów

W przypadku wykorzystania języka C++ wszystkie funkcje systemowe dostępne są w przestrzeni nazw *isix::* . Dodatkowo przygotowano wrappery w postaci klas dla zadań (klasa *task_base*), semaforów (klasa *semaphore*) oraz kolejek (klasa wzorcowa *fifo*). W przypadku nowego zadania/wątku w języku C++ każda klasa powinna dziedziczyć z klasy *task_base* i powinna implementować metodę wirtualną *run()*, która stanowi zadanie/wątek danej klasy. Klasa *semaphore* jest prostym wrapperem C++ na API języka C i implementuje wszystkie metody opisane wcześniej w tekście. Klasa *fifo* jest wrapperem C++ na API kolejek FIFO i została zaimplementowana w postaci klasy wzorcowej. Aby utworzyć kolejkę FIFO 10 elementów typu *int*, wystarczy zdefiniować zmienną w postaci *isix::fifo<int> kolejka(10)*;

Podsumowując: umiejętne stosowanie języka C++, znajomość jego mechanizmów oraz działania kompilatora pozwala uzyskać aplikacje o podobnym rozmiarze i zajmowanych zasobach. Dodatkowe mechanizmy kontrolne języka C++ przyczynią się do dużo większej niezawodności pisanego oprogramowania niż analogicznych programów w języku C++.

Lucjan Bryndza, EP
lucjan.bryndza@ep.com.pl