



Mikrokontrolery ARM

Laboratorium 5

Krzysztof Pierczyk
8 maja 2020



WSTĘP

Na kolejnych zajęciach laboratoryjnych zajęliśmy się podstawowymi mechanizmami synchronizacji międzywątkowej. Na przykładzie prostych aplikacji sprowadzających się do manipulacji pojedynczymi pinami I/O skorzystaliśmy z **mechanizmów tworzenia wątków, semaforów, kolejek FIFO** oraz **systemu zdarzeń bitowych**. Tak jak w przypadku poprzednich zajęć wykorzystaliśmy w tym celu system operacyjny ISIX przeznaczony dla mikrokontrolerów z rdzeniami ARM.

ZADANIE 1

Pierwsze zadanie dotyczyło tworzenia i manipulowania wątkami w systemie ISIX. Naszym zadaniem była implementacja prostej dwuwątkowej aplikacji, w której jeden z wątków obsługiwałby miganie diodą LED, a drugi monitorował stan przycisku USER i po wykryciu naciśnięcia zawieszał / wznawiał pierwszy wątek. Biorąc pod uwagę, że kod konfiguracyjny oraz kod pierwszego wątku są trywialne i były już niejednokrotnie wykorzystywane w tej samej formie na poprzednich zajęciach, skupimy się na kodzie wątku nadzorującego. Po utworzeniu wątku nadzorującego tworzony jest wątek obsługujący miganie diody. Wykorzystywana jest w tym celu funkcja `thread_create_and_run(...)`.

```
isix::thread blink_h =
    isix::thread_create_and_run(
        1536,
        isix::get_min_priority(),
        isix_task_flag_suspended,
        app::blink
    );
```

Rysunek 1. Utworzenie wątku migającego przez wątek nadzorujący

Jako trzeci argument przyjmuje on flagi konfiguracyjne – w tym przypadku flagę `isix_task_flag_suspended`, która sprawia, że wątek tworzony jest jako zawieszony. Po utworzeniu wątku program wchodzi do pętli `while(true)`, w której odczytuje stan

przycisku USER. Zaimplementowana została **programowa procedura eliminacji drgań styków**. Ze względu na fakt, że była ona przedstawiana już w ramach poprzednich zajęć, jej kod również został pominięty.

Główny fragment programu przedstawiony został poniżej. Po stwierdzeniu przez wątek nadzorujący, że przycisk został wciśnięty, odczytywany jest aktualny stan drugiego wątku. Stan ten jest następnie sprawdzany w zależności od tego, czy wątek jest zawieszony, czy działający (lub uśpiony przez funkcję `isix::wait_ms(...)`) to jego stan jest zmieniany, a na diagnostyczny port szeregowy zostaje wysłany stosowny komunikat. Jeżeli stan wątku jest inny nie przewidziany, na port zostaje wysłana wiadomość o awarii, a system zostaje zresetowany.

```
auto state = blink_h.get_state();

if(state == OSTHR_STATE_SUSPEND){
    blink_h.resume();
    dbprintf("Thread resumed");
}
else if(state == OSTHR_STATE_RUNNING || state == OSTHR_STATE_SLEEPING){
    dbprintf("Thread suspended");
    blink_h.suspend();
}
else{
    dbprintf("'blink' thread crashed (state : %i). Rebooting...", state);
    isix::wait_ms(500);
    isix_reboot();
}
```

Rysunek 2. Fragment programu odpowiadający za zawieszanie i wybudzanie wątku migającego diodą

Program reaguje zgodnie z oczekiwaniami – wciśnięcie przycisku USER powoduje przerwanie / wznowienie migania diody LD3.

ZADANIE 2

W zadaniu drugim zajęliśmy się pierwszym z mechanizmów synchronizacji międzywłtkowej – **semaforami**. Ponownie poligonem treningowym był prosty program operujący na przycisku i diodzie LED. Tym razem jeden z dwóch wątków miał za zadanie monitorować stan przycisku USER, natomiast drugi zapalać lub gasić diodę LD3 w zależności od stanu przycisku. Komunikacja między dwoma wątkami miała się odbyć za pośrednictwem zmiennej, która byłaby zabezpieczona właśnie za pomocą semafora.

Ponownie utworzono dwa wątki, tym razem oba w stanie aktywnym. Kod wątku odbierającego dane został przedstawiony poniżej

```
void reader() {
    while(true) {

        // Wait for data to arrive
        semaphore.wait(ISIX_TIME_INFINITE);

        // Update LD3 state
        periph::gpio::set(LD3, pressed);
        dbprintf("LED3 switched");

    }
}
```

Rysunek 3. Wątek czytający informację o stanie przycisku

Jak widać, jedynym zadaniem wątku jest oczekiwanie na odblokowanie (binarnego) globalnego semafora semaphore oraz przełączenie diody LED (zgodnie ze stanem przycisku USER przechowywanym w zmiennej `bool pressed`) i zakomunikowanie tego faktu na diagnostycznym porcie szeregowym. Odrobinę większym skomplikowaniem charakteryzuje się funkcja sprawdzająca stan przycisku. Ponownie kod związany z programową realizacją eliminacji drgań styków został pominięty. Najważniejsza część programu została przedstawiona na Rys.4.

```

        if(periph::gpio::get(BUTTON)){
            if(!pressed){
                pressed = true;
                semaphore.signal();
            }
        }
        else{
            pressed = false;
            semaphore.signal();
        }
    }
}
else if(pressed){
    pressed = false;
    semaphore.signal();
}
}

```

Rysunek 4. Kod informujący o zmianie stanu przycisku

Jak widać, w każdym przypadku, gdy stan przycisku ulega zmianie, wołana jest metoda `signal()` obiektu semafora. Powoduje to odblokowanie czekającego wątku `reader()` oraz zaktualizowanie stanu diody. Wątek ten następnie ponownie zawiesza się na semaforze do następnego wywołania `signal()`. Ponownie aplikacja funkcjonuje zgodnie z oczekiwaniami – dioda LD3 świeci się tak długo, jak długo wciśnięty jest przycisk USER.

Semaforey, szczególnie użyty w powyższym przykładzie semafor binarny. Mogą wydawać się podobne do **mutexów** (*ang. mutual exclusion*). W obu przypadkach wątek może zatrzymać się na obiekcie blokującym i wznowić swoją pracę, gdy otrzyma sygnał o jego zwolnieniu. Między oboma typami synchronizacji istnieją jednak pewne różnice. Po pierwsze, semafor jest mechanizmem **sygnalizującym**, gdy mutex jest mechanizmem blokującym. Oznacza to, że zadaniem semafora jest wzajemne powiadamianie się wątków o dostępności nowych zasobów. Mutex używany jest natomiast wtedy, gdy wątek wchodzi w obszar operujący na pamięci, która może być zmieniana tylko przez jednego użytkownika.

Ponadto licznik semafora **może być inkrementowany przez dowolny wątek** niezależnie od tego, przez kogo został on dekrementowany. Mutex z kolei może zostać odblokowany jedynie przez wątek, który go zablokował. Ostatnia różnica dotyczy zachowania priorytetu wątków w systemie. Semafor działa na zasadzie prostego licznika. Jeżeli licznik ten jest niezerowy, to pierwszy wątek, który zatrzyma się na nim (niezależnie od swojego priorytetu) zdekrementuje go i przejdzie do sekcji krytycznej. W przypadku mutexów możliwe jest

dopuszczanie oczekujących wątków zgodnie z ich priorytetem. Mechanizm ten jest w wielu przypadkach niezbędny aby uniknąć tzw. **inwersji priorytetów**.

Semafore pozwalają wielu wątkom użytkować wspólnie pewną ilość zasobów, natomiast mutexy umożliwiają wielu wątkom korzystanie z jednego zasobu, przy czym dostęp do nich odbywa się na wyłączność. Jeżeli wątki muszą współdzielić kilka zasobów, wykorzystanie mutexów może być uciążliwe, gdyż ich liczba wzrasta wraz z ilością tychże zasobów. Jeżeli natomiast zasób występuje pojedynczo, lepszym wyborem będzie właśnie mutex, ponieważ zapewnia obsługę wątków zgodnie z ich priorytetami.

ZADANIE 3

Praktyczna strona kolejnego z zadań była taka sama jak w przypadku zadania 2. Program miał wizualizować (domyślnie przy pomocy diody) stan przycisku USER. Różnica tkwić miała jednak w implementacji. Po pierwsze wykorzystana miała być w tym przypadku komunikacja **między wątkiem a procesem**. Po drugie, komunikacja ta miała się odbywać z wykorzystaniem **kolejki FIFO**.

Idąc za przykładem poprzednich zadań, poniższa analiza skupi się jedynie na tych elementach programu, które nie były omawiane w ramach poprzednich zajęć laboratoryjnych. Po pierwsze, odczytywanie stanu przycisku USER odbywała się tym razem w procedurze przerwania. Pin PA0, do którego podłączony jest przycisk, został skonfigurowany jako źródło przerwań linii 0 modułu przerwań zewnętrznych EXTI. Przerwania zostały skonfigurowane tak, aby aktywowane były zarówno **na zboczu opadającym jak i narastającym**.

```
extern "C" {
    void exti0_isr_vector() {
        LL_EXTI_ClearFlag_0_31(LL_EXTI_LINE_0);
        fifo.push_isr(periph::gpio::get(BUTTON));
    }
}
```

Rysunek 5. Procedura obsługi przerwania pochodzącego od przycisku

Procedura obsługi przerwania jest prosta i składa się tylko z dwóch instrukcji. Pierwsza z nich to oczywiście zgłoszenie flagi przerwania w module EXTI. Druga to wstawienie do kolejki `fifo` aktualnej wartości pinu PA0. Kolejka ta została zainicjalizowana jako 1-elementowa kolejka przechowująca dane typu `bool`. Warto zauważyć, że wykorzystana została metoda `push_isr(...)`, a nie `push(...)`. Spowodowane jest to faktem, że w razie przepełnienia kolejki nie zawiesza ona programu, co byłoby niedopuszczalne wewnątrz procedury obsługi przerwania.

```
void reader() {  
  
    static bool pressed = false;  
  
    while(true) {  
        fifo.pop(pressed, ISIX_TIME_INFINITE);  
        periph::gpio::set(LED3, pressed);  
        dbprintf("LED3 switched");  
    }  
}
```

Rysunek 5. Wątek odbierający dane z kolejki FIFO

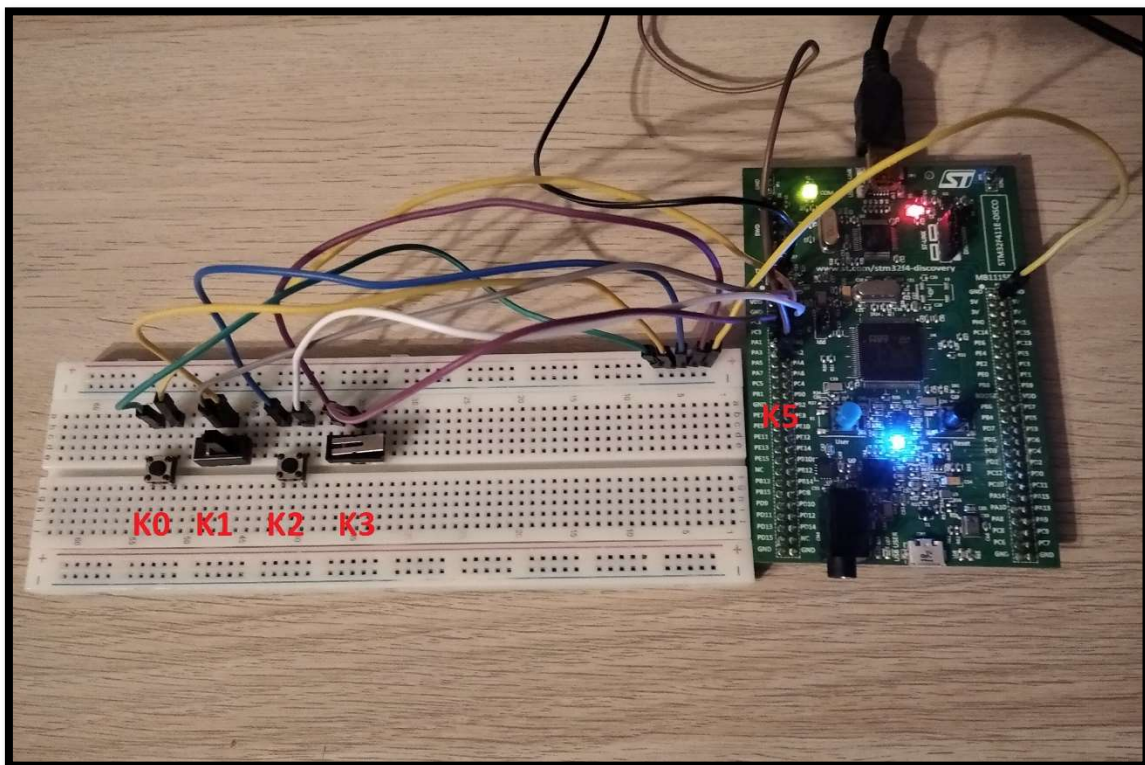
Kod wątku sterującego diodą LED również jest bardzo prosty. Pracuje on w pętli `while(true)`, w której to oczekuje na pojawienie się nowego komunikatu w kolejce. Timeout został ustawiony na nieskończony, ponieważ wątek nie ma żadnych ważnych zadań, na które mógłby przeznaczyć czas oczekiwania. Po odczytaniu nowej wiadomości z kolejki ustawia on stan diody LED3 na właśnie wartość odczytaną. Dodatkowo wysyła on prosty komunikat na diagnostyczny port szeregowy. Również tym razem aplikacja działa zgodnie z oczekiwaniami.

ZADANIE 4

Ostatnie z zadań wymagało od nas skorzystania z mechanizmu zdarzeń bitowych. Jest on odpowiednikiem mechanizmów *poll* / *epoll* znanych z systemów Unixowych. Udostępnia on 31-bitowe rejestry, których zawartość może być ustawiana przez dowolny wątek. Wątki, które oczekują na pojawienie się nowych danych lub zajście jakiegoś zdarzenia mogą

zablokować się na obiekcie `isix::event` w oczekiwaniu na pojawienie się konkretnej kombinacji bitów. Dzięki temu utworzyć można niemalże dowolną kombinację warunków wznowiających pracę wątku oczekującego.

W naszym przypadku ponownie mechanizm ten miał zostać wykorzystany do sterowania diodami LED dostępnymi na płytce ewaluacyjnej. Zgodnie z treścią zadania należało utworzyć 5 wątków dostarczających dane oraz 4 wątki je konsumujące. Dostarczanie danych miało polegać na monitorowaniu stanu 5 pinów I/O i informowaniu o ich wciśnięciu wątków konsumujących. Zmianę stanu linii I/O najłatwiej jest przeprowadzić przy pomocy przycisku. Jako, że na płytce ewaluacyjnej dostępny jest tylko jeden, 4 pozostałe zostały dołączone z zewnątrz. Wykorzystane zostały 3 przyciski monostabilne i (z braku zasobów) jeden bistabilny. Połączenia w układzie zostały przedstawione na Rys. 6.



Rysunek 6. Dołączenie przycisków do płytki ewaluacyjnej

Przyciski zewnętrzne zostały dołączone do pinów C0-C3. Te z kolei zostały programowo podciągnięte do napięcia zasilania. W takiej konfiguracji przystąpiono do pisania programu. Wątki konsumencki zostały zrealizowane przez pojedynczą funkcję `reader(. . .)`. Jej ciało zostało przedstawione na rysunku Rys.7.


```

void reader(const unsigned char LD_num){

    while(true){
        switch(LD_num){
            case 0:
                event.wait(0x1, true, true);
                break;
            case 1:
                event.wait(0x2, true, true);
                break;
            case 2:
                event.wait(0x4, true, true);
                break;
            case 3:
                event.wait(0x18, true, true);
                break;
        }

        periph::gpio::toggle(LD[LD_num]);
        dbprintf("LD%i set to %i", LD_num + 3, periph::gpio::get(LD[LD_num]));

    }
}

```

Rysunek 7. Ciało wątków oczekujących na wystąpienie zdarzenia bitowego

Funkcja przyjmuje pojedynczy argument typu `unsigned char`. Jest to numer diody, którą kontroluje wątek (w zakresie 0-3). W zależności od tego numeru oczekiwane jest pojawienie się innej kombinacji bitowej w obiekcie `event`. Pierwsze trzy wątki oczekują na pojawienie się pojedynczego bitu. Wątek czwarty **wymaga zapalenia dwóch bitów** co jest równoważne wciśnięciu przycisków K3 i K4 w dowolnej kolejności. Po zarejestrowaniu zdarzenia, odpowiadający mu bit jest automatycznie zerowany (za co odpowiada pierwsza flaga `true` na liście argumentów metody `reader(const unsigned char)`). Wątek przełącza stan kontrolowanej przez niego diody LED i komunikuje to na diagnostycznym porcie szeregowym.

Wątki monitorujące stan przycisków również zostały zaimplementowane z wykorzystaniem pojedynczej funkcji. Przyjmuje ona ponownie liczbę całkowitą w zakresie 0-4, która przypisuje wątek do danego przycisku. Gdy procedura eliminacji drgań styków uzna przycisk za wciśnięty, wołana jest metoda `set(...)` obiektu `event` wraz z odpowiednią maską bitową. Na koniec, na port szeregowy wysyłana jest informacja o zarejestrowaniu wciśnięcia danego przycisku.

```

void writer(const unsigned char K_num) {

    // Obsługa programowej eliminacji drgań styków
    // ...
    // ...
    // ...

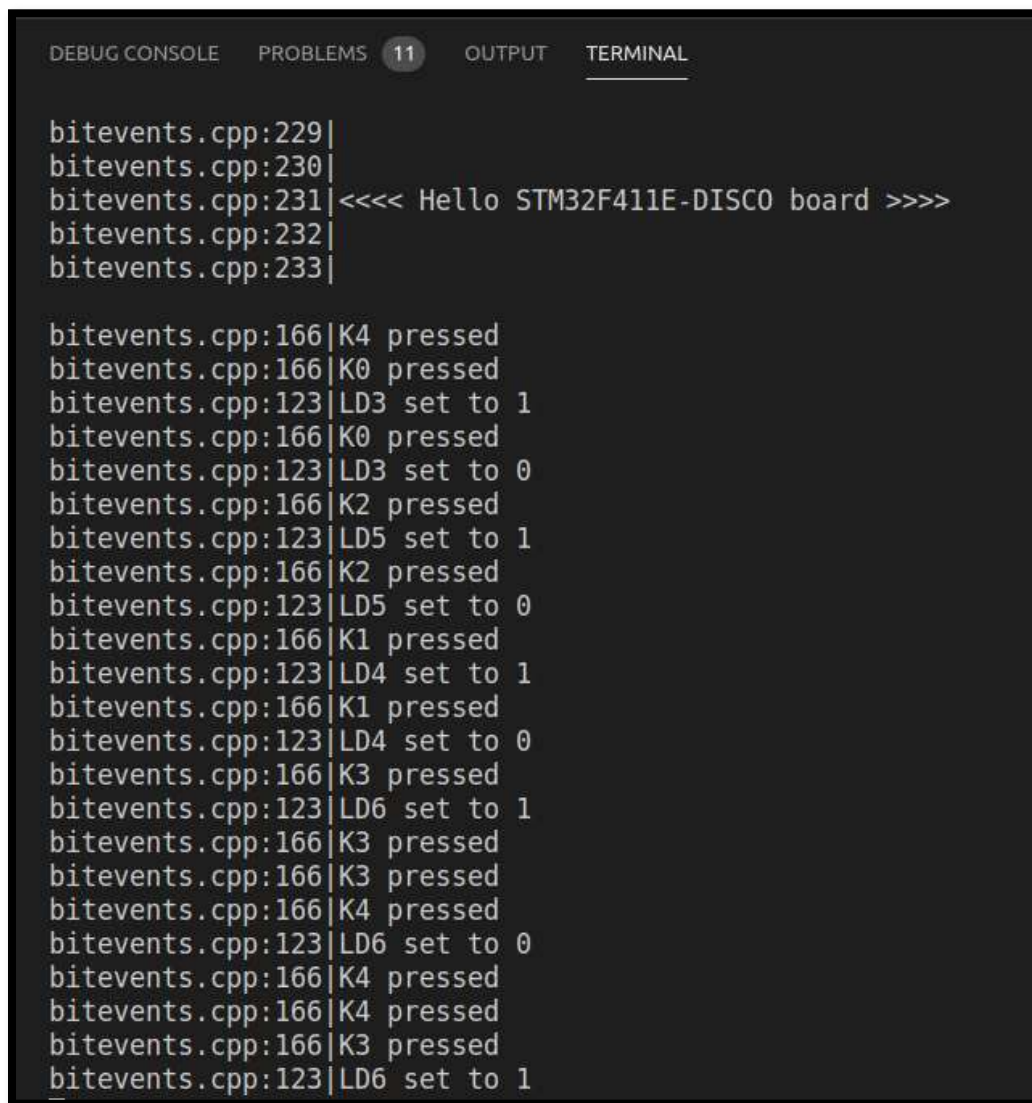
    switch(K_num){
        case 0:
            event.set(0x1);
            break;
        case 1:
            event.set(0x2);
            break;
        case 2:
            event.set(0x4);
            break;
        case 3:
            event.set(0x8);
            break;
        case 4:
            event.set(0x10);
            break;
    }
    dbprintf("%i pressed", K_num);
}

```

Rysunek 8. Ciało wątków generujących zdarzenia bitowe

Mechanizm zdarzeń bitowych pozwala w łatwy sposób obsłużyć złożone schematy sygnalizacji międzywątkowej. Umożliwia jednoczesną synchronizację wielu procesów oraz ułatwia zarządzanie energią poprzez zlikwidowanie potrzeby aktywnego oczekiwania.

Również w przypadku ostatniego zadania kod zadziałał zgodnie z oczekiwaniami bez większych problemów. Efekt działania możemy zobaczyć na logu wypisanym na porcie szeregowym przedstawionym na Rys.9.



```
DEBUG CONSOLE  PROBLEMS 11  OUTPUT  TERMINAL

bitevents.cpp:229|
bitevents.cpp:230|
bitevents.cpp:231|<<<< Hello STM32F411E-DISCO board >>>>
bitevents.cpp:232|
bitevents.cpp:233|

bitevents.cpp:166|K4 pressed
bitevents.cpp:166|K0 pressed
bitevents.cpp:123|LD3 set to 1
bitevents.cpp:166|K0 pressed
bitevents.cpp:123|LD3 set to 0
bitevents.cpp:166|K2 pressed
bitevents.cpp:123|LD5 set to 1
bitevents.cpp:166|K2 pressed
bitevents.cpp:123|LD5 set to 0
bitevents.cpp:166|K1 pressed
bitevents.cpp:123|LD4 set to 1
bitevents.cpp:166|K1 pressed
bitevents.cpp:123|LD4 set to 0
bitevents.cpp:166|K3 pressed
bitevents.cpp:123|LD6 set to 1
bitevents.cpp:166|K3 pressed
bitevents.cpp:166|K3 pressed
bitevents.cpp:166|K4 pressed
bitevents.cpp:123|LD6 set to 0
bitevents.cpp:166|K4 pressed
bitevents.cpp:166|K4 pressed
bitevents.cpp:166|K3 pressed
bitevents.cpp:123|LD6 set to 1
```

Rysunek 8. Komunikaty wypisywane przez wątki producentów i konsumentów

PODSUMOWANIE

Z biegiem lat mikrokontrolery zaczynają być stosowane w coraz to nowych obszarach. Ich rosnąca moc obliczeniowa, a także poprawa wydajności energetycznej sprawiają, że są one idealnym rozwiązaniem we wszelkiego rodzaju układach monitorowania, regulacji, w zastosowaniach IoT czy nawet w urządzeniach multimedialnych. W związku z tym, tworzone projekty stają się coraz bardziej złożone i na pewnym etapie ich rozbudowywanie z wykorzystaniem jedynie 'gołego' sprzętu staje się niemożliwe. Odpowiedzią na ten problem stały się systemy czasu rzeczywistego. Wprowadzając często niewielki narzut obliczeń na procesor udostępniają one wiele mechanizmów programistycznych znanych

z komputerów ogólnego przeznaczenia jak wielowątkowość, mechanizmy synchronizacji, uniwersalne sterowniki peryferiów czy implementacji różnego rodzaju stosów programowych. Narzędzia te są nieocenione w pracy programisty systemów wbudowanych. W bardzo dużym stopniu skracają one, tak ważny w dzisiejszych czasach, *time to market* oraz eliminują wiele potencjalnych błędów.

Kompaktowość RTOS pisanych na mikrokontrolery sprawia, że nie uniemożliwiają one pracy jednostki ze spełnieniem deterministycznych ograniczeń czasowych. Jest to szczególnie ważne w przypadku układów o wymagających dużej precyzji czasowej, które są spotykane w robotyce i automatyce. Przykładem może być rozwijana na naszym wydziale platforma robotyczna Velma. Obecnie jest ona sterowana przy użyciu systemu Linux, który nie jest w stanie dotrzymać rygorów czasowych wymaganych przez ramiona robota. Sprawia to, że system ma tendencje do zawieszania się czy restartowania w nieoczekiwanych momentach. Sytuację tę mogłoby rozwiązać zaimplementowanie interfejsu pomiędzy jednostką centralną a sprzętem, który wykorzystując system RTOS byłby w stanie komunikować się z dotrzymaniem wymaganych rygorów.

Dalszy rozwój techniki będzie kładł jeszcze większy nacisk na tworzenie kompaktowych, uniwersalnych i niezawodnych systemów czasu rzeczywistego. Ich rozwój jest jedną z kluczowych kwestii dla rozwoju coraz większego segmentu elektroniki. Warto zatem zapoznać się ze specyfiką ich działając uczęszczając na przykład na zajęcia z MARM.