

Zuzanna Godek 318373

MARM sprawozdanie z laboratorium 5

Celem laboratorium było zapoznanie się z systemami czasu rzeczywistego stosowanymi w systemach wbudowanych na przykładzie systemu ISIX RTOS.

Zadanie 1.1 Tworzenie wybudzanie oraz usypianie wątków

W tym zadaniu należało stworzyć dwa wątki, jeden (t1) odczytujący przycisk USER i w przypadku wykrycia wciśnięcia wybudzający, bądź usypiający drugi wątek. Natomiast drugi wątek (t2) miał migać diodą z częstotliwością 2Hz po wybudzeniu.

Na początku zadeklarowano globalne zmienne do obsługi przycisku i diody oraz uchwyt na wątek t2, żeby mógł być wykorzystywany przez wątek t1 do wybudzania/usypiania t2:

```
constexpr auto led_0 = periph::gpio::num::PD13;
constexpr auto btn_0 = periph::gpio::num::PA0;

volatile ostask_t t2_handle = nullptr;
```

Zaimplementowano inicjalizację leda PD13 (typ wyjścia pushpull oraz prędkość na niską) i przycisku USER (z podciąganiem w dół) oraz USART do logów:

```
void led_button_init() {
    periph::gpio::setup(led_0, periph::gpio::mode::out {
        periph::gpio::outtype::pushpull,
        periph::gpio::speed::low
    });

    periph::gpio::setup(btn_0, periph::gpio::mode::in {
        periph::gpio::pulltype::down
    });
}

auto usart_protected_init() -> void {
    static isix::mutex m_mtx;
    dblog_init_locked(
        [](int ch, void*) { return
periph::drivers::uart_early::putc(ch); },
        nullptr,
        []() { m_mtx.lock(); },
        []() { m_mtx.unlock(); },
        periph::drivers::uart_early::open, "serial0", 115200
    );
}
```

Funkcja wątku t1 wyglądała następująco:

```
void t1(void*) {
    bool t2_is_running = false;

    bool last_btn_state = false;

    for(;;) {
        bool curr_btn_state = periph::gpio::get(btn_0);

        if (curr_btn_state && !last_btn_state) {

            isix::wait_ms(20);

            if (periph::gpio::get(btn_0)) {
                dbprintf("Przycisk wcisniety!");

                if (t2_handle != nullptr) {
                    if (t2_is_running) {
                        dbprintf("Usypiam T2");
                        isix_task_suspend(t2_handle);
                        t2_is_running = false;
                    } else {
                        dbprintf("Budze T2");
                        isix_task_resume(t2_handle);
                        t2_is_running = true;
                    }
                }
            }

            last_btn_state = curr_btn_state;

            isix::wait_ms(20);
        }
    }
}
```

Odczytywała stan przycisku poprzez odpytywanie co 20 ms, zapisując poprzedni stan i porównując go potem z obecnym, żeby wykryć przyciśnięcie. W przypadku zmiany stanu na wciśnięty zaimplementowano prosty debouncing drgań zestyków, który po 20ms sprawdzał czy przycisk był nadal wciśnięty. Jeśli tak to odpowiednio usypiano lub budzono wątek t2.

Wątek t2 posiadał poniższą funkcję:

```
void t2(void*) {
```

```

int i = 0;
for(;;) {
    periph::gpio::set(led_0, i % 2);
    i++;
    isix::wait_ms(250);
}
}

```

Wątek t2 działa w nieskończonej pętli i przełącza diodę na stan przeciwny. Częstotliwość 2 Hz zrealizowano poprzez funkcję sleep na 250 ms, ponieważ jest to pół okresu odpowiadającego 2Hz, a dioda ma być przez pół okresu włączona i pół wyłączona.

Funkcja main wyglądała następująco:

```

auto main() -> int
{
    usart_protected_init();
    led_button_init();

    dbprintf("<<<< START SYSTEMU BUDZIK >>>>");

    static constexpr auto tsk_stack = 2048;
    static constexpr auto tsk_prio = 6;

    static auto t2_thread_obj = isix::thread_create_and_run(
        tsk_stack,
        tsk_prio,
        isix_task_flag_suspended,
        t2,
        nullptr
    );

    t2_handle = t2_thread_obj.tid();

    if (t2_handle == nullptr) {
        dbprintf("Bład tworzenia watku T2!");
    }

    isix::task_create(t1, nullptr, tsk_stack, tsk_prio);

    isix::start_scheduler();

    return 0;
}

```

Wątek t2 był tworzony w stanie suspended, czyli nie działał dopóki nie włączył go wątek t1. W tej funkcji zapisano do zmiennej globalnej uchwyt wątku t2.

Zweryfikowano działanie programu poprzez klikanie przycisku i obserwowanie diody oraz odczytywanie logów programu na porcie szeregowym.

```
tasks.cpp:84 | <<<< START SYSTEMU BUDZIK >>>>
tasks.cpp:26 | Przycisk wcisniety!
tasks.cpp:34 | Budze T2
tasks.cpp:26 | Przycisk wcisniety!
tasks.cpp:30 | Usypiam T2
tasks.cpp:26 | Przycisk wcisniety!
tasks.cpp:34 | Budze T2
tasks.cpp:26 | Przycisk wcisniety!
tasks.cpp:30 | Usypiam T2
tasks.cpp:26 | Przycisk wcisniety!
tasks.cpp:34 | Budze T2
tasks.cpp:26 | Przycisk wcisniety!
tasks.cpp:30 | Usypiam T2
```

Zadanie 1.2. Komunikacja z wykorzystaniem wspólnej pamięci oraz semaforów

Celem tego zadania było zaimplementowanie programu który włączał lub wyłączał diodę ale bez mrugania jeśli była włączona. W tym przypadku należało skorzystać z semafora do komunikacji między wątkami.

Także na początku zdefiniowano zmienne globalne i zamiast uchwytu na wątek stworzono globalną zmienną semafora:

```
constexpr auto led_0 = periph::gpio::num::PD13;
constexpr auto btn_0 = periph::gpio::num::PA0;
static isix::semaphore semaforek{0, 1};
```

Inicjalizacja led'a oraz przycisku była taka sama.

Natomiast funkcja wątku t1 wyglądała następująco:

```
void t1(void*) {

    bool last_btn_state = false;

    for(;;) {
        bool curr_btn_state = periph::gpio::get(btn_0);

        if (curr_btn_state && !last_btn_state) {

            isix::wait_ms(20);

            if (periph::gpio::get(btn_0)) {
```

```

        dbprintf("Przycisk wcisniety!");
        bool curr_val = semaforek.getval();
        if (curr_val) {
            semaforek.reset(0);
        }
        else {
            semaforek.signal();
        }
    }

    last_btn_state = curr_btn_state;

    isix::wait_ms(20);
}
}

```

Podobnie jak w 1.1 działała na zasadzie odpytywania stanu przycisku z prostym debouncingiem, a jeśli wykryła wciśnięcie to ustawiała wartość semafora na przeciwną.

Wątek t2 natomiast co 20 ms sprawdzał stan semafora i ustawiał led na jego wartość:

```

void t2(void*) {
    for(;;) {
        periph::gpio::set(led_0, semaforek.getval());
        isix::wait_ms(20);
    }
}

```

W tym zadaniu main był bardzo prosty i wątki uruchamiano z takim samym priorytetem i bez uśpienia:

```

auto main() -> int
{
    usart_protected_init();
    led_button_init();

    dbprintf("<<<< START SYSTEMU SEMAFOREK >>>>");

    static constexpr auto tsk_stack = 2048;
    static constexpr auto tsk_prio = 6;

    isix::task_create(t1, nullptr, tsk_stack, tsk_prio);
    isix::task_create(t2, nullptr, tsk_stack, tsk_prio);
}

```

```

    isix::start_scheduler();

    return 0;
}

```

Działanie programu zweryfikowano poprzez klikanie przycisku i oglądanie diody LD3, a także poprzez odczyt logów:

```

semaphore.cpp:78 | <<<< START SYSTEMU SEMAFOREK >>>>
semaphore.cpp:24 | Przycisk wcisniety!
semaphore.cpp:31 | Dioda ma sie wlaczyc!
semaphore.cpp:24 | Przycisk wcisniety!
semaphore.cpp:27 | Dioda ma sie wylaczyc!
semaphore.cpp:24 | Przycisk wcisniety!
semaphore.cpp:31 | Dioda ma sie wlaczyc!
semaphore.cpp:24 | Przycisk wcisniety!
semaphore.cpp:27 | Dioda ma sie wylaczyc!
semaphore.cpp:24 | Przycisk wcisniety!
semaphore.cpp:31 | Dioda ma sie wlaczyc!
semaphore.cpp:24 | Przycisk wcisniety!
semaphore.cpp:27 | Dioda ma sie wylaczyc!

```

Zadanie 1.3. Komunikacja z wykorzystaniem kolejek FIFO

W tym zadaniu komunikacja między przyciskiem a wątkiem była zrealizowana przy użyciu kolejki FIFO. Powodowało to, że wątek był usypiany jeśli przycisk był nieaktywny, natomiast po wykryciu przyciśnięcia przerwanie wrzucało informację do kolejki, która budziła wątek i ten zmieniał stan diody.

Na początku zdefiniowano zmienne globalne, w tym kolejkę FIFO:

```

constexpr auto led_0 = periph::gpio::num::PD13;
constexpr auto btn_0 = periph::gpio::num::PA0;

static isix::fifo<bool> kolejeczka(10);

```

Następnie skonfigurowano led PD13 i przycisk, tym razem tak żeby generował on przerwania na linii 0 na wzrastającym zboczu sygnału z przycisku:

```

void led_button_init() {
    periph::gpio::setup(led_0, periph::gpio::mode::out {
        periph::gpio::outtype::pushpull,
        periph::gpio::speed::low
    });

    periph::gpio::setup(btn_0, periph::gpio::mode::in {
        periph::gpio::pulltype::down
    });
}

```

```

});
LL_APB2_GRP1_EnableClock(LL_APB2_GRP1_PERIPH_SYSCFG);

LL_SYSCFG_SetEXTISource(LL_SYSCFG_EXTI_PORTA, LL_SYSCFG_EXTI_LINE0);

LL_EXTI_EnableIT_0_31(LL_EXTI_LINE_0);

LL_EXTI_EnableRisingTrig_0_31(LL_EXTI_LINE_0);
NVIC_SetPriority(EXTI0_IRQn, 6);
NVIC_EnableIRQ(EXTI0_IRQn);
}

```

Inicjalizacja USART była taka sama jak wcześniej.

Natomiast dodano obsługę przerwania która wrzucała daną bool do kolejki przy użyciu funkcji *push_isr* (nieblokującej, bezpiecznej dla przerw). Nie zastosowano *dbprintf* do logów, bo wykorzystuje mutex, którego nie można używać w obsłudze przerwania.

```

extern "C" {
    //! Exti 0 vector
    void exti0_isr_vector() {
        LL_EXTI_ClearFlag_0_31(LL_EXTI_LINE_0);
        kolejczka.push_isr(true);
    }
}

```

Natomiast przełączanie diody zrealizowano w wątku:

```

void t1(void*) {
    bool received_val;
    bool last_state = false;
    for (;;) {
        if (kolejczka.pop(received_val, ISIX_TIME_INFINITE) ==
ISIX_EOK) {
            last_state = !last_state;
            periph::gpio::set(led_0, last_state);
            dbprintf("Odebrano informacje z kolejki! Zmiana LED na: %d",
last_state);
        }
    }
}

```

Próbował w nieskończonej pętli wyciągnąć dane z kolejki, jeśli ich tam nie było to był usypiany. Jeśli były to przełączał led'a na stan przeciwny.

Funkcja main była analogiczna do poprzednich zadań.

Zweryfikowano działanie programu poprzez obserwację diody po przyciśnięciu przycisku i odczytanie logów:

```
Type [C-a] [C-h] to see available commands
Terminal ready
♦fifoirq.cpp:79|<<<< START SYSTEMU KOLEJECZKA >>>>
fifoirq.cpp:28|Odebrano informacje z kolejki! Zmiana LED na: 1
fifoirq.cpp:28|Odebrano informacje z kolejki! Zmiana LED na: 0
fifoirq.cpp:28|Odebrano informacje z kolejki! Zmiana LED na: 1
fifoirq.cpp:28|Odebrano informacje z kolejki! Zmiana LED na: 0
fifoirq.cpp:28|Odebrano informacje z kolejki! Zmiana LED na: 1
fifoirq.cpp:28|Odebrano informacje z kolejki! Zmiana LED na: 0
```

Zadanie 1.4. Zaawansowana komunikacja z wykorzystaniem mechanizmu zdarzeń bitowych (events)

To zadanie polegało na ustawianiu bitów w zmiennej zdarzeń bitowych przez stan wysoki na odpowiedniej linii, a następnie oczekiwanie na ustalone kombinacje i w odpowiedzi włączanie odpowiednich diod.

Najpierw dla wygody zdefiniowano zmienne globalne ledów i linii PC, a także zmienną zdarzeń bitowych:

```
constexpr auto led_3 = periph::gpio::num::PD13;
constexpr auto led_4 = periph::gpio::num::PD12;
constexpr auto led_5 = periph::gpio::num::PD14;
constexpr auto led_6 = periph::gpio::num::PD15;
constexpr auto k0 = periph::gpio::num::PC0;
constexpr auto k1 = periph::gpio::num::PC1;
constexpr auto k2 = periph::gpio::num::PC2;
constexpr auto k3 = periph::gpio::num::PC3;
constexpr auto k4 = periph::gpio::num::PC4;
static isix::event wydarzenia;
```

Potem zainicjalizowano standardowo ledy (tryb wyjścia, push-pull), a także linie P0...P4 (tryb wejścia z pull do stanu wysokiego):

```
void led_button_init() {
    periph::gpio::setup(led_3, periph::gpio::mode::out {
        periph::gpio::outtype::pushpull,
        periph::gpio::speed::low
    });
    periph::gpio::setup(led_4, periph::gpio::mode::out {
        periph::gpio::outtype::pushpull,
        periph::gpio::speed::low
    });
};
```

```

periph::gpio::setup(led_5, periph::gpio::mode::out {
    periph::gpio::outtype::pushpull,
    periph::gpio::speed::low
});
periph::gpio::setup(led_6, periph::gpio::mode::out {
    periph::gpio::outtype::pushpull,
    periph::gpio::speed::low
});
periph::gpio::setup(k0, periph::gpio::mode::in {
    periph::gpio::pulltype::up
});
periph::gpio::setup(k1, periph::gpio::mode::in {
    periph::gpio::pulltype::up
});
periph::gpio::setup(k2, periph::gpio::mode::in {
    periph::gpio::pulltype::up
});
periph::gpio::setup(k3, periph::gpio::mode::in {
    periph::gpio::pulltype::up
});
periph::gpio::setup(k4, periph::gpio::mode::in {
    periph::gpio::pulltype::up
});
}

```

Następnie zaimplementowano funkcje wątków sprawdzających stan linii poprzez odpytywanie i ustawiających odpowiednie bity:

```

void tk0(void*) {
    bool last_k0 = true;
    for (;;) {
        bool curr_k0 = periph::gpio::get(k0);
        if(!curr_k0 && last_k0) {
            dbprintf("Wykryto stan wysoki na linii PC0");
            wydarzenia.set(1);
        }
        last_k0 = curr_k0;
        isix::wait_ms(20);
    }
}

void tk1(void*) {
    bool last_k1 = true;
    for (;;) {

```

```

        bool curr_k1 = periph::gpio::get(k1);
        if(!curr_k1 && last_k1) {
            dbprintf("Wykryto stan wysoki na linii PC1");
            wydarzenia.set(2);
        }
        last_k1 = curr_k1;
        isix::wait_ms(20);
    }
}

void tk2(void*) {
    bool last_k2 = true;
    for (;;) {
        bool curr_k2 = periph::gpio::get(k2);
        if(!curr_k2 && last_k2) {
            dbprintf("Wykryto stan wysoki na linii PC2");
            wydarzenia.set(4);
        }
        last_k2 = curr_k2;
        isix::wait_ms(20);
    }
}

void tk3(void*) {
    bool last_k3 = true;
    for (;;) {
        bool curr_k3 = periph::gpio::get(k3);
        if(!curr_k3 && last_k3) {
            dbprintf("Wykryto stan wysoki na linii PC3");
            wydarzenia.set(8);
        }
        last_k3 = curr_k3;
        isix::wait_ms(20);
    }
}

void tk4(void*) {
    bool last_k4 = true;
    for (;;) {
        bool curr_k4 = periph::gpio::get(k4);
        if(!curr_k4 && last_k4) {
            dbprintf("Wykryto stan wysoki na linii PC4");
            wydarzenia.set(16);
        }
    }
}

```

```

    }

    last_k4 = curr_k4;
    isix::wait_ms(20);
}
}

```

Następnie zaimplementowano funkcje wątków obsługujących diody na podstawie zdarzeń bitowych, gdzie tl3-tl5 czekały na pojedyncze bity, natomiast tl6 czekał na kombinację (logiczną sumę) bitów 3 i 4 co daje wartość dziesiętną 24 (bo $2^3 + 2^4 = 8 + 16 = 24$). W funkcji wait do zmiennej *isix::event* ustawiono najpierw bit/bity na które ma czekać, a potem flagi *clear_on_exit* i *wait_for_all* na true, żeby czyścić bity i czekał na wszystkie (logiczny AND potrzebny dla tl6).

```

void tl3(void*) {
    bool last_state = false;
    for (;;) {
        if (wydarzenia.wait(1, true, true, ISIX_TIME_INFINITE) & 1) {
            last_state = !last_state;
            if (last_state) {
                dbprintf("Włączam diodę LD3");
            }
            else {
                dbprintf("Wyłączam diodę LD3");
            }
           Periph::gpio::set(led_3, last_state);
        }
    }
}

```

```

void tl4(void*) {
    bool last_state = false;
    for (;;) {
        if (wydarzenia.wait(2, true, true, ISIX_TIME_INFINITE) & 2) {
            last_state = !last_state;
            if (last_state) {
                dbprintf("Włączam diodę LD4");
            }
            else {
                dbprintf("Wyłączam diodę LD4");
            }
           Periph::gpio::set(led_4, last_state);
        }
    }
}

```

```

    }
}

void tl5(void*) {
    bool last_state = false;
    for (;;) {
        if (wydarzenia.wait(4, true, true, ISIX_TIME_INFINITE) & 4) {
            last_state = !last_state;
            if (last_state) {
                dbprintf("Włączam diodę LD5");
            }
            else {
                dbprintf("Wyłączam diodę LD5");
            }
           Periph::gpio::set(led_5, last_state);
        }
    }
}

void tl6(void*) {
    bool last_state = false;
    for (;;) {
        if (wydarzenia.wait(24, true, true, ISIX_TIME_INFINITE) & 24) {
            last_state = !last_state;
            if (last_state) {
                dbprintf("Włączam diodę LD6");
            }
            else {
                dbprintf("Wyłączam diodę LD6");
            }
           Periph::gpio::set(led_6, last_state);
        }
    }
}

```

Funkcja main była analogiczna do poprzednich, uruchamiała wszystkie wątki.

Poprawność działania zweryfikowano poprzez zwieranie odpowiednich pinów (P0...P4) z GND i obserwację diod oraz logów (wykonano sekwencję włączenia wszystkich diod po kolei a potem ich wyłączenia):

```

events.cpp:212|<<<< START SYSTEMU WYDARZENIA >>>>
events.cpp:30|Wykryto stan wysoki na linii PC0
events.cpp:97|Włączam diodę LD3
events.cpp:43|Wykryto stan wysoki na linii PC1
events.cpp:114|Włączam diodę LD4
events.cpp:56|Wykryto stan wysoki na linii PC2
events.cpp:131|Włączam diodę LD5
events.cpp:69|Wykryto stan wysoki na linii PC3
events.cpp:82|Wykryto stan wysoki na linii PC4
events.cpp:148|Włączam diodę LD6
events.cpp:30|Wykryto stan wysoki na linii PC0
events.cpp:100|Wyłączam diodę LD3
events.cpp:43|Wykryto stan wysoki na linii PC1
events.cpp:117|Wyłączam diodę LD4
events.cpp:56|Wykryto stan wysoki na linii PC2
events.cpp:134|Wyłączam diodę LD5
events.cpp:69|Wykryto stan wysoki na linii PC3
events.cpp:82|Wykryto stan wysoki na linii PC4
events.cpp:151|Wyłączam diodę LD6

```

- **W jakich zastosowaniach użycie mechanizmu mutex jest lepsze niż zastosowanie semaforów?**

Zastosowanie mutexa jest lepsze od semaforów w przypadku izolacji zasobów współdzielonych przez wątki o różnych priorytetach. Zapobiega to wówczas zjawisku inwersji priorytetów, poprzez implementację mechanizmu dziedziczenia priorytetów przez mutexy.

- **Do czego potrzebujemy systemów operacyjnych czasu rzeczywistego w mikrokontrolerach?**

Żeby zapewnić determinizm wykonywanym zadaniom. Mikrokontrolery zazwyczaj stosuje się do sterowania fizycznymi urządzeniami, które muszą być obsługiwane w określonym czasie, żeby np. nie doszło do ich uszkodzenia w wyniku opóźnień reakcji kontrolera.