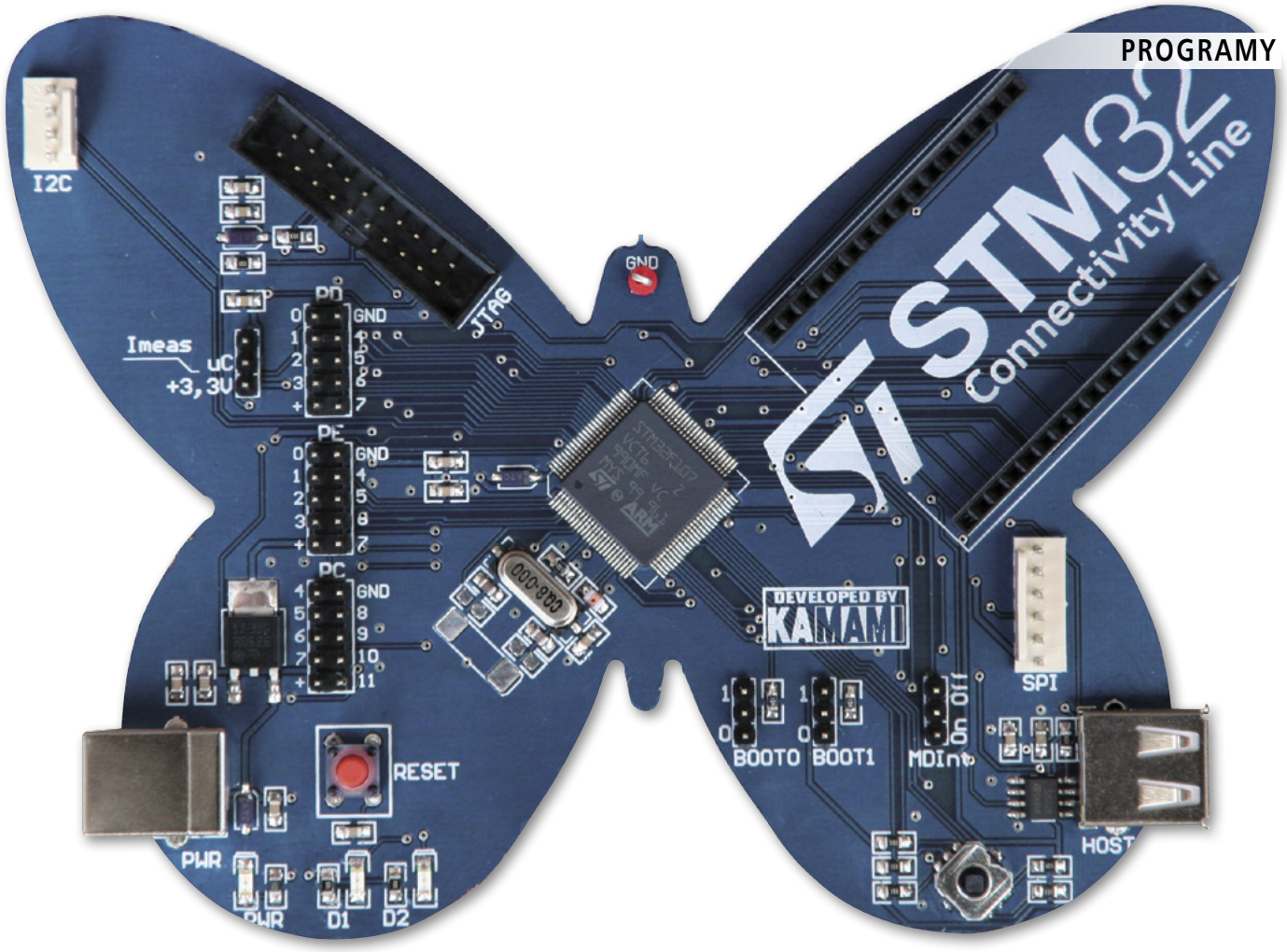


ISIX-RTOS – przykłady w języku C

Miniaturowy system operacyjny ISIX przedstawiliśmy Czytelnikom w EP 7/2010. Przykłady napisane dla niego w języku C++ cieszyły się dużym zainteresowaniem, ale wielu Czytelników EP sygnalizowało chęć zapoznania się z podobnymi aplikacjami napisanymi w C. Odpowiedź na te postulaty przedstawiamy w artykule. Wszystkie przykłady przygotowano dla mikrokontrolera STM32F107 zastosowanego w słynnym zestawie STM32Butterfly.

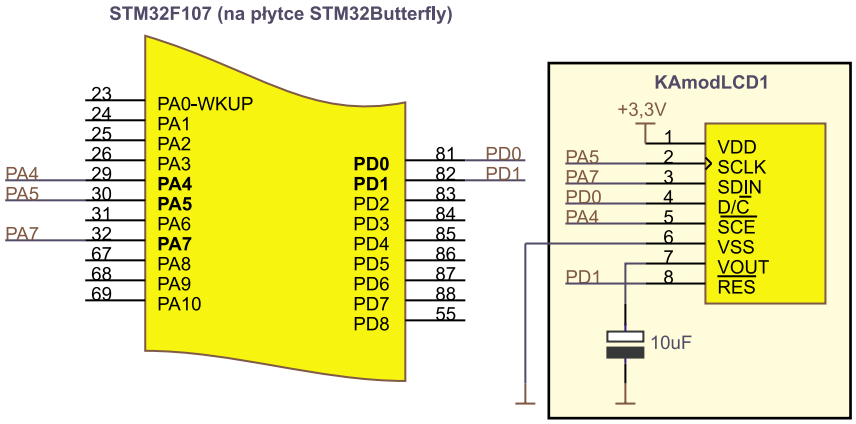
Dodatkowe informacje:
Na CD-EP12/2010 znajdują się pliki źródłowe projektów prezentowanych w artykule oraz systemu ISIX-RTOS.

Dodatkowe materiały na CD i FTP:
ftp://ep.com.pl, user: 16195, pass: 4k17u606

Przykład 1: LED+LCD+ joystick pokładowy

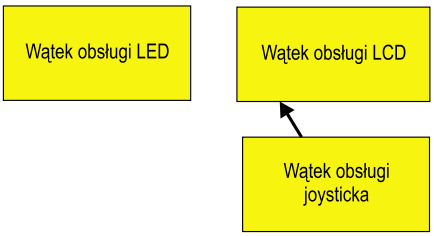
W pierwszym projekcie pokażemy jak tworzyć wątki w języku C z użyciem systemu ISIX oraz w jaki sposób skomunikować je ze sobą. Przedstawimy również możliwość zastosowania wątków do realizacji trzech niezależnych zadań: migania diodą D1 w zestawie STM32Butterfly, sprawdzania stanu joysticka zamontowanego w tym zestawie oraz wyświetlania odpowiednich komunikatów na graficznym wyświetlaczu LCD z telefonu Nokia 3310, którego sposób dołączenia do mikrokontrolera pokazano na **rysunku 1**. Działanie przykładowej aplikacji z uwzględnieniem podziału na wątki przedstawiono na **rysunku 2**. Cykliczne miganie diodą LED odbywa się w niezależnym wątku pokazanym na **listingu 1**.

Funkcję implementującą wątek w systemie ISIX deklarujemy za pomocą makra ISIX_TASK_FUNC, które jako argumenty przyjmuje nazwę funkcji oraz nazwę parametru funkcji. Każda funkcja implementująca zadanie w systemie ISIX ma parametr typu void*, poprzez który możemy przekazać dane podczas jego tworzenia. W wątku najpierw jest konfigurowany port mikrokontrolera PE14, do którego dołączono diodę D1, aby pełnił funkcję linii wyjściowej. Następnie program wchodzi do niekończącej się pętli, w której cyklicznie jest zmieniany stan diody LED. Opóźnienie (tak aby było widoczne miganie diody) jest realizowane za pomocą funkcji isix_wait_ms(), która powoduje wywłaszczenie (uśpienie) wątku na zadaną liczbę milisekund. Pokrewną funkcją jest isix_wait(), która usypia wykonywanie wątku na zadaną liczbę cykli systemu operacyjnego (*ticks*). W systemie ISIX funkcje implementujące wątek muszą zawierać pętlę nieskończoną, a zakończenie wykonywania wątku jest możliwe tylko za pomocą wywołania isix_task_delete(NULL). Tworzenie wątków systemowych odbywa się w funkcji main() (**listing 2**).



Rysunek 1. Schemat podłączenia modułu z LCD do mikrokontrolera STM32F107 w zestawie STM32Butterfly

Dodatkowe informacje, filmy i projekty dla mikrokontrolerów STM32 – w języku polskim! – są dostępne na stronie *www.stm32.eu*



Rysunek 2. Działanie przykładowej aplikacji z uwzględnieniem podziału na wątki

Tworzenie wątków w systemie ISIX odbywa się za pomocą funkcji task_t* **isix_task_create**(task_func_ptr_t task_func, **void** *func_param, **unsigned long** stack_depth, prio_t priority);. Jako pierwszy argument task_func przyjmuje wskaźnik do funkcji tworzącej wątek. 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.

Najpierw tworzony jest wątek blinking_task migający diodą LED. Jako argument dodatkowy przekazujemy mu wartość NULL, ponieważ nie wykorzystujemy żadnych dodatkowych danych oraz nie komunikujemy się z innymi wątkami. Kolejną czynnością jest utworzenie kolejki FIFO, za pomocą której będziemy przekazywać kody wciśniętych klawiszy do wątku wyświetlania. Tworzenie nowej kolejki FIFO odbywa się za pomocą funkcji fifo_t* **isix_fifo_create**(**int** n_elem, **size_t** elem_size). Jako pierwszy argument podajemy liczbę elementów kolejki FIFO, natomiast jako drugi argument przekazujemy wielkość pojedynczego elementu w kolejce. W przypadku pomyślnego utworzenia kolejki FIFO jest zwracany wskaźnik do nowo utworzonej kolejki. W naszym przypadku kolejka składa się z 10 elementów o wielkości sizeof(key_t). Jeżeli kolejka została utworzona pomyślnie (wskaźnik do kolejki fifo_t jest różny od NULL), tworzony jest wątek display_srv_task odpowiedzialny za odbieranie komunikatów z kodami wciśniętych klawiszy oraz keyboard_srv_task odpowiedzialny za odczytywanie stanu klawiszy i wysyłanie kodów. Po utworzeniu wszystkich wątków jako ostatnia wywoływana jest funkcja isix_start_scheduler(), co powoduje uruchomienie planisty zadań (*schedulera*) systemu ISIX. Za odczytywanie stanu joysticka odpowiedzialny jest wątek keyboard_srv_task (**listing 3**).

Wątek rozpoczyna swoje działanie od skonfigurowania linii PE8...PE12, do których podłączono styki joysticka zamontowanego na płytce STM32Butterfly. Parametr entry_param przekazany podczas tworzenia zadania/wątku zawiera wskaźnik na kolejkę FIFO, do której przekazywane będą kody klawiszy, a zatem musi być rzutowany na typ fifo_t. Następnie wątek wchodzi do pętli głównej, w której cyklicznie jest odczytywany stan linii portów i w przypadku wykrycia wciśnięcia klawisza jego kod jest wpisywany do kolejki FIFO za pomocą funkcji isix_fifo_write(). Funkcja ta jako pierwszy argument przyjmuje wskaźnik do kolejki, jako drugi argument – wskaźnik do elementu, który zostanie wpisany do kolejki, natomiast jako ostatni przekazujemy czas oczekiwania na zapisanie elementu do kolejki. W przypadku, gdy kolejka jest zapełniona, wykonywanie wątku zostanie wstrzymane do momentu zwolnienia miejsca w kolejce lub przekroczeniu czasu oczekiwania. W tym przykładzie wystąpienia timeout ustawiono na wartość ISIX_TIME_INFINITE, co oznacza, że czas oczekiwania nie jest ograniczony. Po zapisa-

Listing 1. Wątek odpowiadający za miganie LED w zestawie STM32Butterfly

```
/** Blinking led task function */
static ISIX_TASK_FUNC(blinking_task, entry_param)
{
        RCC->APB2ENR |= RCC_APB2Periph_GPIOE;
        io_config(LED_PORT,LED_PIN,GPIO_MODE_10MHZ,GPIO_CNF_GPIO_PP);
        for(;;)
        {
                //Enable LED
                io_clr( LED_PORT, LED_PIN );
                //Wait time
                isix_wait_ms( BLINK_TIME );
                //Disable LED
                io_set( LED_PORT, LED_PIN );
                //Wait time
                isix_wait_ms( BLINK_TIME );
        }
}
```

Listing 2. Funkcja tworząca wątki systemowe

```
//App main entry point
int main(void)
{
        //Create ISIX blinking task
        isix_task_create( blinking_task, NULL,ISIX_PORT_SCHED_MIN_STACK_DEPTH,
BLINKING_TASK_PRIO);
        fifo_t *kbd_fifo = isix_fifo_create( 10, sizeof(key_t) );
        if(kbd_fifo)
        {
                //Create the display serwer task
                isix_task_create(display_srv_task,kbd_fifo,DISPLAY_TASK_STACK_
SIZE,DISPLAY_TASK_PRIO);

                //Create the keyboyard task
                isix_task_create(keyboard_srv_task, kbd_fifo, KBD_TASK_STACK_
SIZE, KBD_TASK_PRIO);
        }
        //Start the isix scheduler
        isix_start_scheduler();
}
```

niu danych do kolejki wątek jest usypiany na około 25 ms, co pozwala na wyeliminowanie drgań zestyków. Kody wciśniętych przycisków są odbierane przez wątek obsługi wyświetlacza (**listing 4**).

Zadanie/wątek wyświetlacza rozpoczyna swoją prace od inicjalizacji wyświetlacza z telefonu Nokia 3310, wypisaniu w dwóch pierwszych liniach tekstu informacyjnego, a następnie przechodzi do pętli nieskończonej, w której za pomocą funkcji isix_fifo_read()jest odczytywany jest kod klawisza przesłanego przez wątek klawiatury. Funkcja ta jako argumenty przyjmuje wskaźnik do kolejki FIFO, gdzie zostanie skopiowany odebrany element oraz czas oczekiwania na odebranie tego elementu.

Przykład 2: LED+LCD+termometr na I²C

W tym przykładzie pokażemy sposób pisania procedur obsługi przerwań pod kontrolą systemu ISIX na przykładzie obsługi interfejsu I²C, do którego dołączono termometr cyfrowy MCP9801, a uzyskane wyniki pomiaru będą wyświetlane na graficznym wyświetlaczu LCD.

Podobnie jak w pierwszym przykładzie, aplikację podzielono na wątki:

- wątek odczytu temperatury,
- wątek wyświetlania wyniku oraz
- niezależny wątek migający diodą LED.

Wątek odczytu temperatury przekazuje informację o temperaturze z użyciem kolejki FIFO.

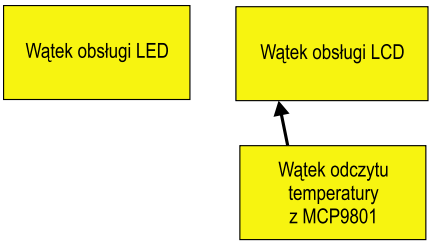
Scalony termometr MCP9801 ma kilkanaście rejestrów kontrolnych i konfiguracyjnych umożliwiających między innymi ustalenie rozdzielczości pomiaru, ustawianie alarmów itp. Ponieważ w przykładzie nie będziemy wykorzystywać pełnych możliwości tego układu, będziemy używać tylko dwóch jego rejestrów:

- ATR (0x00h) umożliwiającego odczyt aktualnej temperatury,
- CONFIG (0x01), który pozwoli nam ustalić rozdzielczość pomiaru temperatury.

Zasadę działania całej aplikacji z uwzględnieniem podziału na wątki przedstawiono na **rysunku 3**.

Wykonywanie programu rozpoczyna się od funkcji main(), pokazanej na **listingu 5**.

Funkcja ta jest bardzo podobna do opisanej w pierwszym przykładzie, w którym



Rysunek 3. Zasada działania aplikacji z uwzględnieniem wątków

Listing 3. Odczyt stanu styków joysticka

```
/** Keyboyard server task */
static ISIX_TASK_FUNC(keyboard_srv_task,entry_params)
{
        //Enable PE in APB2
        RCC->APB2ENR |= RCC_APB2Periph_GPIOE;
        //Set GPIO as inputs
        io_config(KEY_PORT,KEY_OK_BIT,GPIO_MODE_INPUT,GPIO_CNF_IN_FLOAT);
        io_config(KEY_PORT,KEY_UP_BIT,GPIO_MODE_INPUT,GPIO_CNF_IN_FLOAT);
        io_config(KEY_PORT,KEY_DOWN_BIT,GPIO_MODE_INPUT,GPIO_CNF_IN_FLOAT);
        io_config(KEY_PORT,KEY_LEFT_BIT,GPIO_MODE_INPUT,GPIO_CNF_IN_FLOAT);
        io_config(KEY_PORT,KEY_RIGHT_BIT,GPIO_MODE_INPUT,GPIO_CNF_IN_FLOAT);
        fifo_t *kbd_fifo = (fifo_t*)entry_params;
        //Previous key variable
        static key_t p_key = -1;
        for(;;)
        {
                //Get key
                key_t key = get_key();
                //Check if any key is pressed
                if(key!=0 && p_key==0)
                {
                        isix_fifo_write( kbd_fifo, &key, ISIX_TIME_INFINITE );
                }
                //Previous key assignement
                p_key = key;
                //Wait short time
                isix_wait_ms( KBD_DELAY_TIME );
        }
}
```

Listing 4. Odbiór kodów wciśniętych przycisków

```
/** Display server task */
static ISIX_TASK_FUNC(display_srv_task, entry_params)
{
        fifo_t *kbd_fifo = (fifo_t*)entry_params;
        //Initialize LCD
        nlcd_init();
        //Put welcome string
        nlcd_put_string(„www.boff.pl”,0,0);
        nlcd_put_string(„Joy state:”,0,1);
        //Key variable read from
        key_t key;
        for(;;)
        {
                //Read data from fifo
                if(isix_fifo_read( kbd_fifo, &key,ISIX_TIME_INFINITE )==ISIX_
EOK)
                {
                        //Display text based on the key
                        switch(key)
                        {
                        case KEY_LEFT:
                                nlcd_put_string(„** KEY LEFT **”,0,2);
                                break;
                        case KEY_RIGHT:
                                nlcd_put_string(„** KEY RIGHT *”,0,2);
                                break;
                        case KEY_UP:
                                nlcd_put_string(„** KEY UP **  „,0,2);
                                break;
                        case KEY_DOWN:
                                nlcd_put_string(„** KEY DOWN **”,0,2);
                                break;
                        case KEY_OK:
                                nlcd_put_string(„** KEY OK **  „,0,2);
                                break;
                        }
                }
        }
}
```

List. 5. Funkcja main przykładowego programu

```
/** Main func  */
int main(void)
{
        //Create ISIX blinking task
        isix_task_create( blinking_task, NULL,
                        ISIX_PORT_SCHED_MIN_STACK_DEPTH, TASK_PRIO_LED
        );
        //Create fifo msgs
        fifo_t *temp_fifo = isix_fifo_create( 10, sizeof(struct msg) );
        //Initialize I2C bus
        i2cm_init(I2C_SPEED);
        if(temp_fifo)
        {
                //Create isix tasks (temp and disp)
               isix_task_create(temp_read_task,temp_fifo,TASK_STK_SIZE,TASK_
PRIO_TEMP);
               isix_task_create(display_srv_task,temp_fifo,TASK_STK_SIZE,TASK_
PRIO_TEMP);
        }
        isix_start_scheduler();
        //Start the sheduler
}
```

List. 6. Struktura odczytu temperatury
```
//Temperature fractional structure
struct temp
{
        short t;                //temp
        short ft;               // .temp fract
};
```

List. 7. Wątek obsługujący odczyt temperatury oraz przesyłanie wyników pomiarów do wątku wyświetlania
```
/** Temperature read task */
ISIX_TASK_FUNC(temp_read_task,entry_params)
{
        fifo_t *temp_fifo = (fifo_t*)entry_params;
        struct msg msg;
        msg.errno = tempsensor_init();  //Init temp sensor
        if(msg.errno<0) //If cfg fail
        {
                //Push error message and destroy task
                isix_fifo_write( temp_fifo, &msg, ISIX_TIME_INFINITE );
                isix_task_delete(NULL);
        }
        for(;;)
        {
                //Read temp
                msg.errno =  tempsensor_get(&msg.t);
                //Push data
                isix_fifo_write( temp_fifo, &msg, ISIX_TIME_INFINITE );
                //Wait for the next cycle
                isix_wait_ms(MEASURE_WAIT_TIME);
        }
}
```

List. 8. Funkcja realizująca odczyt temperatury
```
//Get current temperature and fill the structure
static int tempsensor_get(struct temp *t)
{
        static const unsigned char temp_reg = MCP9800_TEMP_REG;
        static unsigned char temp[2];
        int ecode;
        //Read the temperature
        ecode = i2cm_transfer_7bit(TEMPSENSOR_I2CADDR,&temp_reg,sizeof(temp_
reg),temp,sizeof(temp));
        //Convert to integer
        if(ecode>=0)
        {
                t->t = temp[0];
                t->ft = (((int)(temp[1]>>4))*10)/16;
        }
        return ecode;
}
```

List. 9. Wątek odpowiadający za wizualizację wyników pomiarów
```
/** Display server task */
static ISIX_TASK_FUNC(display_srv_task, entry_params)
{
        fifo_t *temp_fifo = (fifo_t*)entry_params;
        struct msg msg;                 //Message structure
        bool bstate = false;    //Previous state
        nlcd_init();                    //Initialize LCD
        //Put welcome string
        nlcd_put_string( „www.boff.pl”, 0, 0 );
        nlcd_put_string( „Sensor Temp:”, 0, 1 );
        for(;;)
        {
                //Read data from fifo
                if(isix_fifo_read( temp_fifo, &msg,ISIX_TIME_INFINITE )==ISIX_
EOK)
                {
                        //Display temp or error
                        if(msg.errno>=0)
                                display_temp( 0, 2, &msg.t,”C     „ );
                        else
                                nlcd_put_string(„I2C Error    „,0,2);
                }
                //Blinking *
                if(bstate) nlcd_put_string(„*”,13,0);
                else nlcd_put_string(„ „,13,0);
                bstate=!bstate;
        }
}
```

najpierw utworzono wątek migający diodą LED, następnie kolejkę FIFO o wielkości 10 elementów zawierających informację o aktualnej temperaturze. Wiadomość zawiera strukturę przekazującą dane o temperaturze oraz kod błędu zwracany przez funkcję odczytu czujnika MCP9801. Struktura opisująca aktualną temperaturę została zdefiniowana w sposób pokazany na **listingu 6**.

Składowa *t* określa bezwzględną wartość temperatury, natomiast składowa *ft* określa ułamkową wartość temperatury (zakres 0...9).

Po utworzeniu kolejki komunikatów następuje wywołanie funkcji i2cm_init() inicjalizującej bibliotekę obsługi interfejsu I²C (co dalej będzie szczegółowo opisane). Następnie jest tworzony wątek obsługi odczytu temperatury temp_read_task oraz wątek obsługi wyświetlacza display_srv_task, a na zakończenie wywoływana jest funkcja isix_start_scheduler() uruchamiająca planistę zadań. Wątek obsługi LED jest realizowany przez funkcję implementującą blinking_task i jest on analogiczny do opisanego w poprzednim przykładzie. Za realizację odczytu temperatury oraz przesyłania wyników pomiarów do wątku wyświetlania jest odpowiedzialny wątek temp_read_task() (**listing 7**).

Wątek rozpoczyna pracę od inicjalizacji czujnika temperatury tempsensor_init(). W przypadku niepowodzenia, do kolejki komunikatów serwera wyświetlania wysyłany jest kod błędu, następnie wątek pomiaru temperatury jest usuwany. W przypadku gdy wszystko przebiega pomyślnie, wątek wchodzi do pętli nieskończonej, odczytywana jest wartość bieżącej temperatury, która następnie przesyłana jest jako wiadomość do kolejki FIFO wątku wyświetlania. Procedura inicjalizacji czujnika MCP9801 sprowadza się do wpisania do rejestru konfiguracyjnego układu wartości ustawiającej rozdzielczość pomiaru na 12 bitów (co daje rozdzielczość 0,0625°C), gdyż domyślnie czujnik pracuje z rozdzielczością 9-bitową (0,5°C).

Transfer danych poprzez interfejs I²C odbywa się za pomocą pojedynczego wywołania funkcji i2cm_transfer_7bit, która jako pierwszy argument przyjmuje adres sprzętowy I²C, następnie bufor z danymi przeznaczonymi do wysłania na magistralę oraz jego wielkość. Jako ostatnie dwa parametry należy przekazać bufor oraz wielkość danych, które chcemy odczytać z magistrali. W przypadku, gdy chcemy tylko wysyłać lub odbierać dane, możemy w miejsce odpowiednich parametrów podstawić wartość *NULL* oraz *0*, co będzie skutkować tylko odebraniem lub wysłaniem danych. Za odczyt wartości temperatury odpowiada funkcja tempsensor_get() (**listing 8**), która jako argument przyjmuje wskaźnik na strukturę opisującą temperaturę oraz zwraca kod błędu.

Podobnie jak poprzednio, na początku wywoływana jest funkcja transferu danych za pomocą I²C, która przesyła adres rejestru do odczytania (ATR, adres 0), a następnie odczytuje 2 bajty danych, które po przeliczeniu na część całkowitą i ułamkową są wpisywane do struktury reprezentującej temperaturę. Za wizualizację wyników pomiarów odpowiada wątek *display_srv_task* (**listing 9**).

Powyższy przykład nie mógłby działać prawidłowo, gdyby nie biblioteka obsługi interfejsu I²C, która kryje w sobie znacznie więcej niż trzy niepozorne funkcje składające się na interfejs użytkownika. Obsługa I²C została napisana w taki sposób, aby używać systemu przerwań mikrokontrolera. Dzięki temu transfer danych nie obciąża procesora, ponieważ wątek oczekujący na dane z magistrali jest usypiany, a czas procesora oddawany jest innym wątkom. Biblioteka do

synchronizacji międzyprocesowej oraz procedury obsługi przerwań używają 2 semaforów oraz wspólnego obszaru pamięci.

Na początku funkcji tworzony jest semafor sem_lock, który pełni rolę *mutexa* blokującego dostęp do magistrali I²C. W tym samym czasie na magistrali może mieć miejsce tylko jeden transfer, więc wywołanie funkcji przez inny wątek w trakcie aktywnej transmisji danych powoduje jego uśpienie do czasu zakończenia transferu. W systemie ISIX tworzenie semaforów odbywa się za pomocą funkcji isix_sem_create_limited(), której pierwszy argument określa statyczną strukturę semafora, a w przypadku przekazania wartości *NULL* powoduje zaalokowanie struktury semafora na stercie. Drugi argument określa początkową wartość przypisaną do semafora, natomiast ostatni wyznacza maksymalną wartość, którą może przyjąć semafor. W przypadku powodzenia funkcja zwraca wskaźnik do semafora lub wartość *NULL* w przypadku niemożliwości zaalokowania pamięci. Pokrewną funkcją umożliwiającą utworzenie semafora bez limitu wartości jest isix_sem_create(), który przyjmuje tylko dwa argumenty: wskaźnik na statyczną strukturę semafora oraz jego wartość początkową.

Kolejną czynnością jest utworzenie samafora sem_irq, który sygnalizuje zakończenie transferu danych na magistrali I²C. Po utworzeniu semaforów następuje przypisanie portom GPIO roli wejść-wyjść układu peryferyjnego I2C1. Następnie włączane są sygnały zegarowe dla kontrolera I2C1, który jest ustawiany tak, aby pełnił funkcję 7-bitowego kontrolera I²C *master*. Interfejs transferu danych biblioteki tworzy funkcja I2C_transfer_7bit(), która umożliwia przeprowadzenie kompletnego cyklu zapisu oraz odczytu danych z magistrali I²C, co jest jednym z najczęstszych przypadków spotykanych w praktyce.

W funkcji transferu danych najpierw jest wywoływana funkcja isix_sem_wait(), która powoduje oczekiwanie na zwolnienie semafora, co zapewnia blokadę wywołania funkcji przez inne wątki. Funkcja isix_sem_wait() jako pierwszy argument przyjmuje wskaźnik do semafora, natomiast jako drugi przekazywany jest maksymalny czas oczekiwania, po którym zostanie zgłoszony błąd. W naszym przypadku funkcja będzie oczekiwać na zwolnienie semafora z nieograniczonym limitem czasowym. Po zablokowaniu funkcji następuje ustawienie zmiennych odpowiedzialnych za transfer danych, które będą użyte przez procedurę obsługi przerwania, a następnie konfiguracja kontrolera I²C, odblokowanie przerwania od kontrolera I²C oraz wygenerowanie bitu startu. W tym momencie kontroler I²C rozpoczyna zgłaszanie przerwań, a główna procedura transferu jest realizowana przez procedurę obsługi przerwania I²C.

Kontroler generuje przerwanie w przypadku wystąpienia zdarzenia na magistrali, którego rodzaj jest odczytywany za pomocą funkcji get_last_event() i – w zależności od zdarzenia – podejmowana jest stosowna akcja. Procedura obsługi zdarzeń realizowana jest za pomocą konstrukcji case i niczym nie odbiega od typowej obsługi kontrolera I²C w STM32.

Lucjan Bryndza
www.boff.pl