ISIX-RTOS
Obsługa przerwań

Wątki w ISIX-RTOS mogą komunikować się ze sobą za pomocą semaforów lub kolejek komunikatów. Korzystanie z nich może powodować usypianie procesu (sleep state) w wyniku oczekiwania na pozyskanie zasobu. W przypadku przerwań uśpienie procedury obsługi przerwania nie jest możliwe z uwagi na to, że przerwania nie są wykonywane w kontekście procesu.

W związku z tym dla procedur obsługi przerwań należy wywołać tylko metody nieblokujące. W systemie ISIX, w kontekście obsługi przerwań mogą być wywoływane jedynie metody z sufiksem _isr. Sposób obsługi komunikacji pomiędzy zadaniami a procedurami obsługi przerwań pokażemy na przykładzie obsługi magistrali I²C z podłączonym scalonym zegarem RTC M41T56C64 (na przykład z modułu KAmodRTC). Działanie aplikacji sprowadzać się będzie do odczytania godziny z zegara RTC, przetworzenia go na komunikat oraz przesłania do serwera wyświetlania, zaprezentowanego w poprzednim przykładzie. Kolejny niezależny wątek podobnie jak w poprzednim przykładzie będzie migał diodą LED D1 zamontowaną w zestawie STM32Butterfly.

W przykładzie zastosowano także wyświetlacz od Nokii3310 (KAmodLCD1). Sposób dołączenia wyświetlacza i układu RTC do mikrokontrolera pokazano na **rysunku 1**.

Układ M41T56C64 firmy ST jest bardzo interesującym opracowaniem, integrującym w jednej obudowie zegar czasu rzeczywistego RTC z wbudowanym rezonatorem kwarcowym 32768 Hz oraz pamięć EEPROM AT24C64. Układ charakteryzuje się poborem prądu rzędu 400 nA, co umożliwia mu pracę z małej baterii litowej nawet 10 lat. Dzięki zintegrowaniu w układzie rezonatora kwarcowego nie musimy dołączać z zewnątrz praktycznie żadnych elementów, dodatkowo rezonator ten jest kalibrowany w procesie produkcji przez firmę ST, która gwarantuje dokładność rzędu ±5 ppm.

Przykładowy program pokazuje na wyświetlaczu LCD aktualną godzinę oraz miga diodą LED D1 znajdującą się na płytce STM32Butterfly. Sposób działania aplikacji z podziałem na wątki przedstawiono na **rysunku 2**.

Wątek obsługi LED pracuje zupełnie niezależnie od pozostałych wątków, co pozwala pokazać ich wzajemną niezależność. Wątek obsługi LCD, który z punktu widzenia pozostałych wątków jest serwerem wyświetlania, odpowiada za odbiór rozkazów oraz odpowiednie sterowanie wyświetlaczem. Wątek RTC jest odpowiedzialny za odczytanie aktualnej godziny z zegara RTC poprzez interfejs I²C, sformatowanie tekstu, następnie wygenerowanie i przesłanie komunikatu dla serwera wyświetlania. Aplikacja została napisana w sposób obiektowy w języku C++, hierarchię klas przedstawiono na **rysunku 3**.

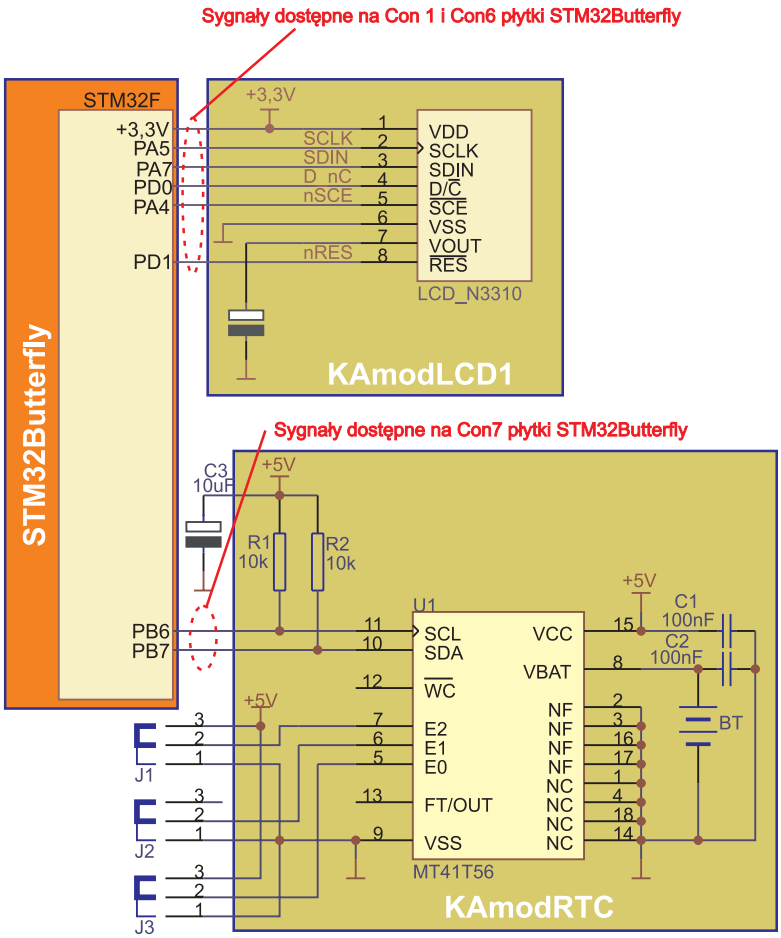
Hierarchia klas jest bardzo podobna do przykładu 2, ponieważ wykorzystano opisaną w nim architekturę serwera wyświetlania. Klasa the_application jest klasą aplikacji, w której zawarto wszystkie pozostałe obiekty. Klasa led_blink jest odpowiedzialna za cykliczne miganie diodą LED i jest dziedziczona z klasy isix::task_base implementującej obsługę wątków. Klasa display_server jest odpowiedzialna za odbiór komunikatów z kolejki FIFO oraz fizyczne sterowanie kontrolerem wyświetlacza za pomocą klasy nokia_display. Klasa i2c_host jest uniwersalną klasą sterownika, implementującą obsługę magistrali I²C z wykorzystaniem sprzętowego kontrolera I2C1 w trybie 7-bitowym. Została ona napisana w taki sposób, aby można było ją rozwinąć o obsługę dodatkowych sprzętowych kontrolerów I²C, występujących w mikrokontrolerach rodziny STM32F. Deklaracja klasy znajduje się w pliku *i2c_host.hpp* (**listingu 1**).

Ustawienie zwór adresowych J1…
J3 na rysunku 1 ma wpływ tylko na adres pamięci EEPROM zintegrowanej w układzie M41T56C64, adres zegara RTC jest niezmienny.

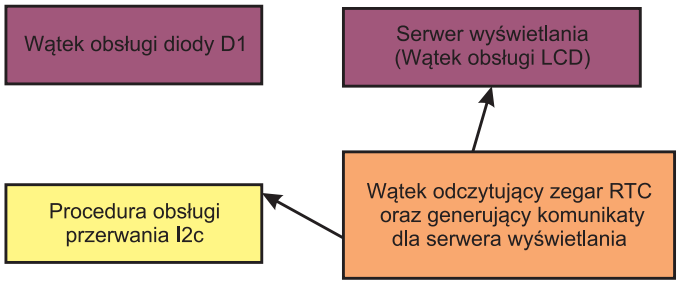
Klasa została zaprzyjaźniona z funkcją i2c1_ev_isr_vector stanowiącą wektor obsługi przerwania od kontrolera I2C1. Wszystkie procedury obsługi przerwań muszą mieć linkowanie typu C (*extern „C”*). Funkcje wywoływane są w momencie wystąpienia przerwania bez żadnych dodatkowych parametrów, co wymusza istnienie dostępu do instancji klasy kontrolera I²C poprzez wskaźnik lub referencję globalną. Wskaźnik do obiektu i2c umożliwiający dostęp do obiektu kontrolera I²C przez funkcję obsługi przerwania umieszczono w nienazwanej przestrzeni nazw w pliku implementacji klasy (*i2c_host.cpp*), przez co dostęp do niej jest możliwy tylko w obrębie tego modułu.

Zadeklarowanie przyjaźni funkcji umożliwia wywoływanie dowolnych metod chronionych klasy z funkcji zaprzyjaźnionej, co zostało wykorzystane do wywołania metody *isr()* stanowiącą wektor obsługi przerwania. Do synchronizacji wątku z procedurą obsługi przerwania wykorzystano semafor sem_irq, natomiast semafor sem_lock jest wykorzystywany do blokowania sterownika w przypadku, gdy jest on zajęty obsługą magistrali I²C.

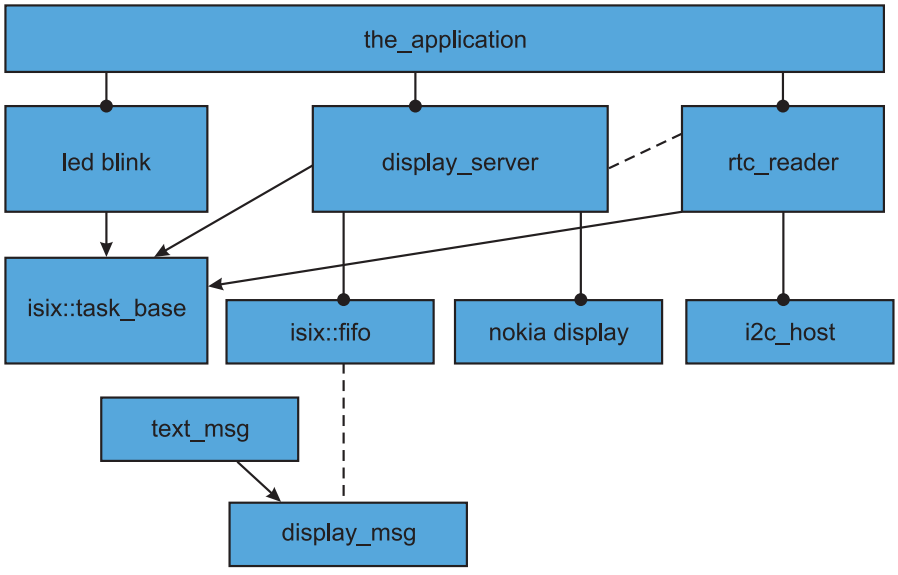
Konstruktor klasy obiektu *i2c_host* przyjmuje dwa argumenty: adres wybranego kontrolera I²C (np. I2C1, I2C2) oraz szybkość taktowania magistrali I²C wyrażoną w hercach,



Rysunek 1. Sposób dołączenia zegara RTC i wyświetlacza LCD do zestawu STM32Butterfly



Rysunek 2. Podział aplikacji na wątki



Rysunek 3. Hierarchia klas projektu

której domyślna wartość wynosi 100000. Zadaniem konstruktora jest inicjalizacja kontrolera sprzętowego I²C, uruchomienie przerwań oraz przypisanie wskaźnika this utworzonego obiektu do wskaźnika wykorzystywanego przez procedurę obsługi przerwania.

Na początku sprawdzany jest numer kontrolera I²C (obecnie zaimplementowana jest tylko obsługa kontrolera I2C1), następnie konfigurowane są porty GPIO, tak aby pełniły funkcję wyjścia układu peryferyjnego. Następnie konfigurowany jest sprzętowy kontroler I²C, tak aby pełnił rolę sterownika *master* z 7-bitowym adresowaniem. Do wskaźnika na obiekt wykorzystanego przez procedurę obsługi przerwania przypisywany jest wskaźnik this aktualnie tworzonego obiektu. Przyjęto założenie, że istnieje tylko jedna instancja klasy i2c_host, ponieważ do danego kontrolera magistrali może być przypisany tylko jeden sterownik. Na zakończenie włączane są przerwania w kontrolerze I²C oraz konfigurowane są przerwania w kontrolerze NVIC.

Jedyną metodą interfejsu użytkownika klasy sterownika I²C jest metoda i2c_transfer_7bit (**listing 3**) umożliwiająca przeprowadzenie na magistrali transakcji I²C.

Jako parametry przyjmuje ona kolejno adres sprzętowy układu I²C, z którym chcemy przeprowadzić transakcję, wskaźnik do bufora z danymi oraz długość bufora, z którego dane chcemy przesłać magistralą I²C. Następnie przekazujemy wskaźnik do bufora, gdzie będą przesłane dane odebrane z magistrali I²C oraz liczba danych, jaką chcemy odczytać. Jeśli chcemy wykonać tylko pojedynczy zapis danych lub odczyt, do nieużywanego wskaźnika na bufor należy przypisać wartość *NULL*. Metoda blokuje wykonanie bieżącego wątku do chwili zakończenia transakcji I²C oraz w przypadku powodzenia zwraca wartość *ERR_OK*. Na początku funkcji wywoływana jest metoda wait() na głównym semaforze blokującym, w wyniku czego proces może zostać zablokowany, jeżeli sterownik jest zajęty przez inny proces. Do zmiennych klasy przypisywane są wskaźniki do buforów oraz wielkości tych buforów i jest generowany sekwencja *start*. Następnie oczekujemy na semaforze sygnalizującym zakończenie pracy przez procedurę obsługi przerwania. Cała obsługa cyklu magistrali wykonywana jest w przerwaniu. Oczekiwanie na semafor kończy się w momencie sygnalizacji przez procedurę obsługi przerwania lub w wyniku przekroczenia czasu oczekiwania, co informuje nas o wystąpieniu błędu. W przypadku powodzenia zwracany jest status *ERR_OK* oraz podnoszony jest semafor blokujący sem_lock za pomocą wywołania metody sem_lock.signal(). Generacja bitu startu powoduje rozpoczęcie działania kontrolera I²C. W wyniku wystąpienia poszczególnych zdarzeń zgłaszane

Listing 1.

```
//I2c host class
class i2c_host
{
        //Friend interrupt class
        friend void i2c1_ev_isr_vector(void);
public:
        enum errno
        {
                ERR_OK = 0,                                                    //All
is ok
                ERR_BUS = -5000,                                      //Bus error
                ERR_ARBITRATION_LOST = -5001,
                ERR_ACK_FAILURE = -5002,
                ERR_OVERRUN = - 5003,
                ERR_PEC = - 5004,                                     //Parity check
error
                ERR_BUS_TIMEOUT = -5005,                //Bus timeout
                ERR_TIMEOUT = - 5006,                   //timeout error
                ERR_UNKNOWN = - 5007
        };
        //Default constructor
        i2c_host(I2C_TypeDef * const _i2c, unsigned clk_speed=100000);
        //I2c transfer main function
        int i2c_transfer_7bit(uint8_t addr, const void* wbuffer, short wsize,
void* rbuffer, short rsize);
private:
        //Interrupt service routine
        void isr();
        //Configuration data
        static const unsigned TRANSFER_TIMEOUT = 1000;
        static const unsigned IRQ_PRIO = 1;
        static const unsigned IRQ_SUB = 7;
        //Rest of the data
        static const unsigned CR1_ACK_BIT = 0x0400;
        static const unsigned CR1_START_BIT = 0x0100;
        static const unsigned CR1_STOP_BIT = 0x0200;
        static const uint16_t I2C_IT_BUF = 0x0400;
        static const uint16_t I2C_IT_EVT = 0x0200;
        static const uint16_t I2C_IT_ERR = 0x0100;
        static const uint16_t CR1_PE_SET = 0x0001;
        //Get last i2c event
        uint32_t get_last_event()
        {
                static const uint32_t sflag_mask = 0x00FFFFFF;
                return ( static_cast<uint32_t>(i2c->SR1) |
                        static_cast<uint32_t>(i2c->SR2)<<16 )
                        & sflag_mask;
        }
        //Send 7 bit address on the i2c bus
        void send_7bit_addr(uint8_t addr)
        {
                i2c->DR = addr;
        }
        //Send data on the bus
        void send_data(uint8_t data)
        {
                i2c->DR = data;
        }
        //Read data from the bus
        uint8_t receive_data()
        {
                return i2c->DR;
        }
        //CR1 reg enable disable
        void cr1_reg(unsigned bit, bool en)
        {
                if(en) i2c->CR1 |= bit;
                else i2c->CR1 &= ~bit;
        }
        //ACK ON control
        void ack_on(bool on)
        {
                cr1_reg( CR1_ACK_BIT, on );
        }
        //Generate start
        void generate_start(bool en=true)
        {
                cr1_reg( CR1_START_BIT, en );
        }
        //Generate stop
        void generate_stop(bool en=true)
        {
                cr1_reg( CR1_STOP_BIT, en );
        }
        //Clear data flags (dummy read)
        void clear_flags()
        {
                static_cast<void>(static_cast<volatile uint16_t>(i2c->SR1));
                static_cast<void>(static_cast<volatile uint16_t>(i2c->DR));
        }
        //Control enabling disabling int in the device
        void devirq_on(bool en=true)
        {
                if(en)
                        /* Enable I2C interrupt */
                        i2c->CR2 |=  I2C_IT_EVT| I2C_IT_ERR;
                else
                        /* diasable I2C interrupt */
                         i2c->CR2 &=  ~(I2C_IT_EVT | I2C_IT_ERR);
        }p
        //Set bus speed
        void set_speed(unsigned speed);
        //Translate error to the error code
        int get_hwerror();


private:        //Data
        //I2c device number
        I2C_TypeDef *i2c;
        //Tx buffer pointer
        const uint8_t *tx_buf;
        //Rx buffer pointer
        uint8_t *rx_buf;
        //Busy semaphore
        isix::semaphore sem_lock;
        //Read semaphore
        isix::semaphore sem_irq;
        //Bus address
        volatile uint8_t bus_addr;
        //Bus error flags
        volatile uint8_t err_flag;
        //Tx counter
        volatile short tx_bytes;
        //Rx counter
        volatile short rx_bytes;
        //Position in the buffer
        volatile short buf_pos;

private:        //Noncopyable
        i2c_host(i2c_host &);
        i2c_host& operator=(const
i2c_host&);
};
```

jest przerwanie, którego obsługa realizowana jest przez procedurę i2c1_ev_isr_vector (**listing 4**).

Funkcja ta jest zaprzyjaźniona z klasą i2c_host. Na rzecz obiektu klasy i2c_host jest wywoływana funkcja isr(), która jest właściwą procedurą obsługi przerwania (**listing 5**).

Kontroler I²C działa na zasadzie zdarzeń, zatem procedura obsługi przerwania sprowadza się do odczytania zdarzenia, a następnie wykonania określonej akcji przypisanej dla tego zdarzenia. W przypadku transmisji danych do układu podłączonego do magistrali I²C, po wysłaniu sekwencji *start* otrzymujemy zdarzenie informującej o przejęciu arbitrażu nad magistralą I²C przez kontroler. W wyniku wystąpienia zdarzenia I2C_EVENT_MASTER_MODE_SELECTED wywołujemy funkcję send_7bit_addr(), co powoduje przesłanie adresu sprzętowego dla urządzenia. W wyniku wysłania adresu sprzętowego dostajemy zdarzenie I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED, po którym możemy rozpocząć przesyłać dane. Wraz z każdym przesłanym bajtem otrzymujemy zdarzenie I2C_EVENT_MASTER_BYTE_TRANSMITTED. Oba zdarzenie obsługiwane są przez ten sam fragment programu, którego zadaniem jest wysłanie danych z bufora. W przypadku przesłania ostatniego bajtu, jeżeli nie ma konieczności odbierania danych, jest wywoływana metoda powodująca wygenerowanie sekwencji *stop*, następnie jest podnoszony semafor sem_irq informujący wątek o zakończeniu obsługi przerwania. Realizacja podnoszenia semafora odbywa się za pomocą metody sem_signal_isr(), która jest przeznaczona do wywołania z procedur obsługi przerwań. Jeżeli po zakończeniu nadawania konieczne jest odbieranie danych z urządzenia, generowany jest ponownie bit startu oraz przesyłany jest adres sprzętowy, z najmłodszym bitem ustawionym do odczytu. W wyniku przesłania adresu sprzętowego otrzymujemy zdarzenie I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED, w którym sprawdzamy, czy mamy do odebrania jeden bajt. Jeżeli tak jest, to wyłączamy generowanie bitu ACK, a następnie przechodzimy do odbioru kolejnych danych. W przypadku odebrania bajtu otrzymujemy zdarzenie I2C_EVENT_MASTER_BYTE_RECEIVED, gdzie realizujemy przepisywanie bajtów do bufora odbiorczego. Gdy skończymy odbieranie przedostatniego bajtu zgodnie ze specyfikacją I²C wyłączamy, generowanie bitu potwierdzenia ACK, a po odebraniu ostatniego bajtu wysyłamy polecenie wyge-

Listing 2.

```
// TODO Add configuration for i2c2 device support
        if(_i2c==I2C1)
        {
            //GPIO configuration
            RCC->APB2ENR |= I2C1_GPIO_ENR;
            io_config(I2C1_PORT,I2C1_SDA_PIN,GPIO_MODE_50MHZ,GPIO_CNF_ALT_OD);
            io_config(I2C1_PORT,I2C1_SCL_PIN,GPIO_MODE_50MHZ,GPIO_CNF_ALT_OD);
            io_set(I2C1_PORT,I2C1_SCL_PIN);
            io_set(I2C1_PORT,I2C1_SDA_PIN);
            //I2C module configuration
            RCC->APB1ENR |= I2C1_ENR;
        }
        /* Enable I2C module*/
        i2c->CR1 |= CR1_PE_SET;
        /* Reset the i2c device */
        i2c->CR1 |= CR1_SWRST;
        nop();
        i2c->CR1 &= ~CR1_SWRST;

         uint16_t tmpreg = i2c->CR2;
         /* Clear frequency FREQ[5:0] bits */
         tmpreg &= CR2_FREQ_RESET;
         tmpreg |= static_cast<uint16_t>(config::PCLK1_HZ/1000000);
         i2c->CR2 = tmpreg;

         //Set speed
         set_speed(clk_speed);

         /* CR1 configuration */
         /* Get the I2Cx CR1 value */
         tmpreg = i2c->CR1;
         /* Clear ACK, SMBTYPE and  SMBUS bits */
         tmpreg &= CR1_CLEAR_MASK;
         /* Configure I2Cx: mode and acknowledgement */
         /* Set SMBTYPE and SMBUS bits according to I2C_Mode value */
         /* Set ACK bit according to I2C_Ack value */
         tmpreg |= I2C_MODE_I2C | I2C_ACK_ENABLE;
         /* Write to I2Cx CR1 */
         i2c->CR1 = tmpreg;

         /* Set I2Cx Own Address1 and acknowledged address */
         i2c->OAR1 = I2C_AcknowledgedAddress_7bit;

         i2c->SR1 = 0; i2c->SR2 = 0;
         /* Enable I2C interrupt */
         devirq_on();
         //Assign as the global object
         if(_i2c==I2C1)
         {
                 i2c1_obj = this;
         }
         /* Enable interrupt controller */
         nvic_set_priority( I2C1_EV_IRQn, IRQ_PRIO, IRQ_SUB);
         nvic_irq_enable(I2C1_EV_IRQn,true);
}
```

Listing 3.

```
int i2c_host::i2c_transfer_7bit(uint8_t addr, const void* wbuffer, short
wsize, void* rbuffer, short rsize)
{
        int ret;
        if( (ret=sem_lock.wait(isix::ISIX_TIME_INFINITE))<0 )
        {
                return ret;
        }
        //Disable I2C irq
        devirq_on(false);
        if(wbuffer)
        {
                bus_addr = addr & ~I2C_BUS_RW_BIT;
        }
        else if(rbuffer)
        {
                bus_addr = addr | I2C_BUS_RW_BIT;
        }
        tx_buf =  static_cast<const uint8_t*>(wbuffer);
        rx_buf =  static_cast<uint8_t*>(rbuffer);
        tx_bytes = wsize;
        rx_bytes = rsize;
        buf_pos = 0;
        //ACK config
        ack_on(true);
        //Enable I2C irq
        devirq_on();
        //Send the start
        generate_start();
        //Sem read lock
        if( (ret=sem_irq.wait(TRANSFER_TIMEOUT)) <0  )
        {
                if(ret==isix::ISIX_ETIMEOUT)
                {
                        sem_irq.signal();
                        sem_lock.signal();
                        return ERR_TIMEOUT;
                }
                else
                {
                        sem_irq.signal();
                        sem_lock.signal();
                        return ret;
                }
        }
```

Listing 3. c.d.

```
        if( (ret=get_hwerror())  )
        {
                err_flag = 0;
                sem_lock.signal();
                return ret;
        }
        sem_lock.signal();
        return ERR_OK;
}
```

Listing 4.

```
//Call to the global c function
extern "C"
{

void i2c1_ev_isr_vector(void) __attribute__ ((interrupt));
void i2c1_ev_isr_vector(void)
{
        if(i2c1_obj) i2c1_obj->isr();
}
```

Listing 5.

```
//I2c interrupt handler
void i2c_host::isr()
{
        uint32_t event = get_last_event();
        switch( event )
        {

        //Send address
        case I2C_EVENT_MASTER_MODE_SELECT:              //EV5
                send_7bit_addr(bus_addr);
        break;

        //Send bytes in tx mode
        case I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED:        //EV6
        case I2C_EVENT_MASTER_BYTE_TRANSMITTED: //EV8
                if(tx_bytes>0)
                {
                        send_data(tx_buf[buf_pos++]);
                        tx_bytes--;
                }
                if(tx_bytes==0)
                {
                        if(rx_buf)
                        {
                                //Change address to read only
                                bus_addr |= I2C_BUS_RW_BIT;
                                ack_on(true);
                                generate_start();
                                buf_pos = 0;
                        }
                        else
                        {
                                generate_stop();
                                sem_irq.signal_isr();
                        }
                }
        break;

        //Master mode selected
        case I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED:   //EV7
                if(rx_bytes==1)
                {
                        ack_on(false);
                }
        break;
        //Master byte rcv
        case I2C_EVENT_MASTER_BYTE_RECEIVED:
                if(rx_bytes>0)
                {
                        rx_buf[buf_pos++] = receive_data();
                        rx_bytes--;
                }
                if(rx_bytes==1)
                {
                        ack_on(false);
                        generate_stop();
            }
                else if(rx_bytes==0)
                {
                        generate_stop();
                        sem_irq.signal_isr();
                }
        break;

        //Stop generated event
        default:
                if(event & EVENT_ERROR_MASK)
                {
                        err_flag = event >> 8;
                        i2c->SR1 &= ~EVENT_ERROR_MASK;
                        sem_irq.signal_isr();
                }
                else
                {
                        clear_flags();
                }
        break;
        }
}
```

nerowania sekwencji *stop* oraz podnosimy semafor sem_isr informujący o zakończeniu procedury odbioru. Po wygenerowaniu sekwencji *stop* magistrala I²C zostaje zwolniona.

Klasa sterownika i2c wykorzystywana jest przez klasę rtc_reader, której zadaniem jest odczytanie bieżącej godziny z zegara RTC oraz przygotowanie i przesłanie komunikatów do obiektu serwera wyświetlacza. Działanie wątku odpowiedzialnego za to zadanie realizowane jest przez metodę wirtualną pokazaną na **listingu 6**.

Listing 6.

```
//Main rtc reader core task
void rtc_reader::main()
{
        static const uint8_t pgm_regs[] =
        {
                0x01,           //Sec
                0x02,           //Min
                0x03,           //Hour
                0x04,           //Day num
                0x05,           //day
                0x06,           //Month
                0x07,           //Year
                0x00            //Config
        };

        //Software address
        static const uint8_t sw_addr = 0;
        static uint8_t buf[3];
        static time_msg tmsg( 3,2 );
        int status;
        //Send configuration registgers
        i2c_bus.i2c_transfer_7bit(I2C_RTC_ADDR,pgm_regs,sizeof(pgm_
regs),NULL,0);
        //Main task loop
        for(;;)
        {
                //Send configuration registgers
                status = i2c_bus.i2c_transfer_7bit(I2C_RTC_ADDR,&sw_
addr,sizeof(sw_addr),buf, sizeof(buf) );

                if(status>=0)
                {
                        //If no error display time
                        tmsg.set_time( buf[2]&0x3f, buf[1]&0x7f, buf[0]&0x7f
);
                }
                else
                {
                        //If error display it
                        tmsg.set_text("I2C ERR");
                }
                //Send message to the i2c device
                disp_srv.send_message(tmsg);
                //Refresh screen delay
                isix::isix_wait(200);
        }
}
```

Listing 7.

```
class time_msg : public text_msg
{
public:
        time_msg(short xpos=0,short ypos=0)
         :text_msg("",xpos,ypos)
         {

         }
        //Set text
        void set_time(short h, short m, short s)
        {
                conv_hex(sbuf,h,2);
                sbuf[2] = ':';
                conv_hex(&sbuf[3],m,2);
                sbuf[5] = ':';
                conv_hex(&sbuf[6],s,2);
                set_text(sbuf);
        }

private:
        void strrev(char *str, int len);
        const char* conv_hex(char *txt, unsigned value,int zeros);

private:
        char sbuf[9];
};
```

Pierwszą czynnością jest ustawienie wartości początkowej daty i czasu. Wartości początkowe zdefiniowane są w tablicy pgm_regs. W rzeczywistej aplikacji należałoby zadbać o możliwość odczytania danych z interfejsu użytkownika. Przesłanie wartości początkowych jest realizowane za pomocą funkcji i2c_transfer_7bit. Następnie wchodzimy do pętli głównej programu, gdzie realizowany jest odczyt bieżącej godziny z zegara RTC. Procedura odbywa się poprzez wywołanie pojedynczej metody i2c_transfer-7bit(). Po odczytaniu danych z magistrali I²C sprawdzany jest status błędu i w przypadku jego wystąpienia jest wysyłany komunikat tekstowy. Jeżeli odczyt danych z magistrali I²C wykonano pomyślnie, jest tworzony komunikat klasy time_msg, zawierający informację o czasie w formie tekstowej, który następnie przesyłany jest do serwera wyświetlania. Klasa time_msg dziedziczy z klasy text_msg, zatem może być wysłana bezpośrednio do serwera wyświetlania (**listing 7**).

Utworzenie komunikatu tekstowego zawierającego aktualny czas odbywa się przez wywołanie metody set_time(), która jako argumenty przyjmuje aktualną godzinę, minutę i sekundę w formacie BCD.

Lucjan Bryndza, EP
lucck@boff.pl