ISIX-RTOS

Wymiana danych pomiędzy wątkami

W artykule przedstawiamy komunikację pomiędzy procesami ISIX-RTOS z wykorzystaniem wątków, na przykładzie prostej aplikacji dla zestawu STM32Butterfly. Jej działanie przejawia się miganiem diody LED (jeden wątek) oraz wyświetlaniem grafik odpowiadających kierunkom wychylania dźwigni joysticka zamontowanego na płytce STM32Butterfly (drugi wątek).

W przykładzie zastosowano wyświetlacz LCD z telefonu Nokia 3310 zamontowany na module KAmodLCD1 (**fotografia 1**). Dołączono go do płytki STM32Butterfly w sposób pokazany na **rysunku 2**. Dane przesyłane przez sprzętowy interfejs SPI1, linie sterujące pracą wyświetlacza LCD dołączono bezpośrednio do innych linii GPIO mikrokontrolera.

Cykliczne miganie LED jest realizowane przez wątek pracujący niezależnie od reszty aplikacji. Część odpowiedzialną za wyświetlanie komunikatów po wciśnięciu pozycji joysticka zrealizowano jako dwa oddzielne wątki: wątek serwera wyświetlania oraz wątek obsługi joysticka. Zadaniem wątku serwera wyświetlania jest odbiór rozkazów poprzez kolejkę FIFO z innych wątków, zwanych dalej klientami serwera wyświetlania, ich interpretację i fizyczne sterowanie wyświetlacza LCD. Wątek obsługi joysticka z punktu widzenia serwera wyświetlania jest klientem. Jego zadaniem jest odczyt stanu styków joysticka oraz, w wyniku interpretacji stanu styków joysticka, wysyłanie odpowiednich rozkazów do serwera wyświetlania, np. wypisania tekstu. Aplikacja została napisana w C++. Hierarchię klas projektu przedstawiono na **rysunku 3**.

Obiekt **the_application** jest głównym obiektem, którego instancja statyczna tworzona jest w funkcji głównej **main() – listing 1**.

Klasa **the_application** zawiera klasy odpowiedzialne za realizację poszczególnych zadań. W klasie tej tworzone są pojedyncze instancje klas: **led_blink**, **display_server**, **graph_key**. Dzięki zawarciu aplikacji w oddzielnej klasie mamy możliwość utworzenia instancji klasy w dowolny sposób, np. na stercie, na stosie czy statycznie, bez konieczności zmiany pozostałych części.

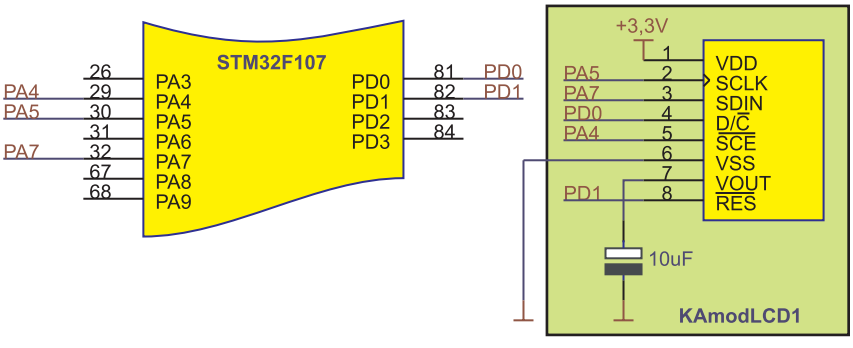
Klasa **ledblink** odpowiedzialna jest za miganie diodą LED i jest dziedziczona z obiektu **isix::task_base** implementującej obsługę wątków. Implementacja klasy jest praktycznie identyczna jak w poprzednio omówionym przykładzie dotyczącym diod LED, dlatego zostanie tutaj pozostawiona bez szerszego komentarza. Miganie diodami LED odbywa się niezależnie od pozostałej części aplikacji, więc klasa ta nie komunikuje się z innymi obiektami.

Dodatkowe informacje:
Artykuł poświęcony systemowi operacyjnemu ISIX-RTOS opublikowaliśmy w EP 3/2010.

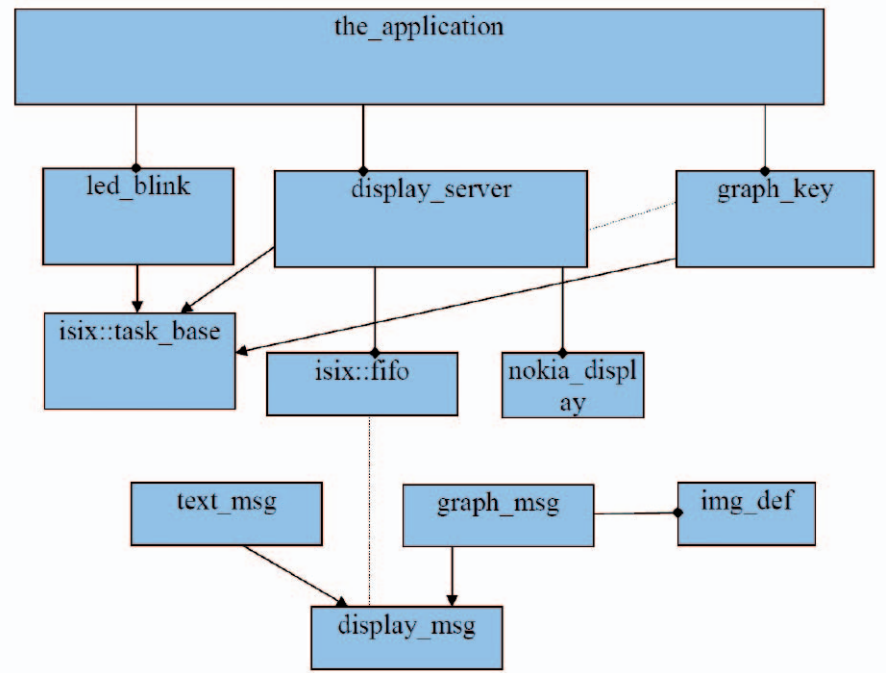
Dodatkowe materiały na CD i FTP:
ftp://ep.com.pl, user: 11825, pass: 81036471

Klasa **display_server** odpowiedzialna jest bezpośrednio za odbiór komunikatów z kolejki FIFO oraz wyświetlanie ich na wyświetlaczu. Klasa **display_msg** jest klasą bazową dla klas **text_msg** oraz klasy **graph_**

Fotografia 1. Wygląd modułu KAmodLCD1



Rysunek 2. Sposób dołączenia wyświetlacza KAmodLCD1 do płytki STM32Butterfly



Rysunek 3. Hierarchia klas przykładu

Listing 1.

```
//App main entry point
int main()
{
    //Application object
    static app::the_application application;

    //Start scheduler
    isix::isix_start_scheduler();
}
```

Listing 2.

```
//Images namespace
namespace images
{
        //Image definition from pure C a SPGL library.
        struct img_def
        {
                int width;
                int height;
                const unsigned char *data;
        };
}
```

Listing 3.

```
//Nokia display class
class nokia_display
{
public:
        //Constructor
        nokia_display();
        //Put char
        void put_char(char code);
        //Set position
        void set_position(unsigned x, unsigned y);
        //Put string
        void put_string(const char *str, unsigned  x, unsigned y);
        //Put bitmap
        void put_bitmap(const images::img_def &image);
        //Clear display
        void clear();
private:
        inline void dc_pin(bool en);
        inline void res_pin(bool en);
        inline void sel_pin(bool en);
        void hw_init();
        void hw_spi_write(unsigned char byte);
        void lcd_init();
        void busy_delay(unsigned delay);
};
```

Listing 4. Konstruktor klasy nokia_display odpowiedzialny za inicjalizację obiektu

```
//Display constructor
nokia_display::nokia_display()
{
        //Initialize hardware
        hw_init();
        //Initialize LCD
        lcd_init();
}
```

Listing 5.
```
//Write character at current position
void nokia_display::put_char(char code)
{
    std::size_t code_point;
         //Data
    dc_pin(1);
         //Char in array calculation
    code_point = static_cast<std::size_t>(code) * 6;
         //Write character.
         sel_pin(0);
    hw_spi_write(cg_rom[code_point++]);
         hw_spi_write(cg_rom[code_point++]);
         hw_spi_write(cg_rom[code_point++]);
         hw_spi_write(cg_rom[code_point++]);
         hw_spi_write(cg_rom[code_point++]);
         hw_spi_write(cg_rom[code_point]);
    //Write end
    sel_pin(1);
}
```

Listing 6.
```
//Set cursor at selected pos (14 cols, 6 rows)
void nokia_display::set_position(unsigned x, unsigned y)
{
         x=x*6;
         //Write activation
         sel_pin(0);
         //Instr
         dc_pin(0);
         hw_spi_write(y|0x40);       //set row position
         hw_spi_write(x|0x80);       //set col position
         // Write stop
         sel_pin(1);
         dc_pin(1);
}
```

msg, dzięki czemu do kolejki FIFO możemy przekazywać zarówno wskaźniki do obiektów klas **text_msg** (komunikat tekstowy), jak i **display_msg** (komunikat graficzny). Klasa **text_msg** jest prostym wrapperem przechowującym wskaźnik do C-stringu **const char***, natomiast klasa **display_msg** jest wrapperem przechowującym strukturę **img_def** (**listing 2**).

Struktura ta pochodzi bezpośrednio z biblioteki graficznej **SPGL** (napisanej w „czystym" C) i przechowuje informację na temat bitmap zawartych w pamięci Flash mikrokontrolera. Wygenerowanie odpowiednich struktur umożliwia narzędzie konwertera umożliwiającego tworzenie na podstawie plików graficznych plików wynikowych *.*c* zawierających definicję obrazków. Wszystkie wygenerowane bitmapy zawarte w projekcie znajdują się w pliku *images.cpp*, natomiast ich definicje zawarte są w pliku *images.hpp* i zostały zawarte w przestrzeni nazw **images**.

Klasa **nokia_display** implementuje fizyczną obsługę wyświetlacza graficznego z telefonu NOKIA-3310. Deklaracja klasy zawarta jest w pliku *nokia_display.hpp* (**listing 3**).

Klasa zawiera metody zapewniające mechanizmy podstawowej obsługi wyświetlacza, takie jak wyświetlanie tekstu, wyświetlanie bitmap czy ustawienie kursora wyświetlacza na zadanej pozycji. Konstruktor klasy (**listing 4**) jest odpowiedzialny za inicjalizację obiektu.

Wywołuje on metodę prywatną **hw_init()**, która jest odpowiedzialna za inicjalizację układów peryferyjnych mikrokontrolera (portów GPIO oraz SPI1). Następnie jest wywoływana metoda prywatna **lcd_init()**, odpowiedzialna za fizyczną inicjalizację wyświetlacza oraz zerowanie pamięci obrazu. Metoda **put_char()** odpowiada za wypisanie pojedynczego znaku na wyświetlaczu LCD (**listing 5**).

Kontroler wyświetlacza nie ma wbudowanego generatora znaków, dlatego tablica znaków została utworzone w kodzie programu, gdzie pojedynczy znak zajmuje 6 bajtów pamięci Flash. Na podstawie numeru porządkowego znaku wyznaczany jest początek znaku w tablicy znaków, a następnie poszczególne bajty znaku przepisywane są do pamięci obrazu wyświetlacza. Wyświetlacz domyślnie jest skonfigurowany w trybie poziomym, więc zapisanie każdego bajtu powoduje zapalenie lub zgaszenie 8 pionowych pikseli. Ustawienie wyświetlania na zadanej pozycji umożliwia metoda *set_position(unsigned x, unsigned y)* – **listing 6**.

Ustawienie wybranej pozycji kursora w pamięci obrazu sprowadza się do wysłania do wyświetlacza komendy **Set Row Position** oraz **Set Col Position**. Działanie klasy **display_server** (wątku serwera) sprowadza się do odczytywania komunikatów (wskaźników do klas komunikatów) z kolejki FIFO, a następnie odpowiednie wykorzystanie metod klasy **nokia_display** do sterowania wyświetlaczem. Zadanie to realizowane jest przez metodę wirtualną **main()** klasy **display_server** (**listing 7**).

W pętli głównej wątku wywoływana jest metoda **pop()** obiektu kolejki FIFO **(io_fifo)**. Metoda ta blokuje wykonywanie wątku do momentu odebrania od innego zadania komunikatu przekazanego za pomocą metody **push()**. W przypadku, gdy kolejka FIFO zawiera jakiś komunikat, wówczas metoda **pop()** wybudza wątek i zwraca kod błędu **isix::ISIX_EOK**. W przypadku odebrania prawidłowego komunikatu, zawartość wyświetlacza LCD jest czyszczona metodą **clear()** obiektu wyświetlacza **display**, a następnie sprawdzany jest typ przekazanego komunikatu (*obiektu*) za pomocą metody **get_type()**

Listing 7. Metoda wirtualna main() klasy display_server
```
//Main thread
void display_server::main()
{
         const display_msg *msg = NULL;

         for(;;)
         {
         //Try read message from the fifo
         if(io_fifo.pop(msg) == isix::ISIX_EOK)
         {
                  //Validate message
                  if( msg )
                  {
                           //Clear display
                           display.clear();
                           //If text message
                           if(msg->get_type()==display_msg::MSG_TEXT)
                           {
                                    //Cast to the text message class
                                    const text_msg *t = static_cast<const text_
msg*>(msg);
                                    if(t->get_text())
                                    {
                                             //Write text on the display
                                    display.put_string(t->get_text(),t-
>get_x(),t->get_y());
                                    }
                                    }
                                    //If graphics message
                                    else if(msg->get_type()==display_msg::MSG_
GRAPHICS)
                                    {
                                             //Get image
const images::img_def *img = static_cast<const graph_msg*>(msg)->get_image();
                                    if(img)
                                    {
                                             //Display bitmap
                                    display.put_bitmap(*img);
                                    }
                           }
                  }
         }
}
}
```

klasy bazowej **display_msg**. Wykrywanie typów klas można zrealizować mechanizmem RTTI (*Run Time Type Information*) i operatorem **typeid()**, jednak z uwagi na dużą zajętość pamięci Flash mechanizm **RTTI** został wyłączony za pomocą flagi kompilatora *-fno-rtti* Jeżeli przekazany obiekt jest typu tekstowego (**text_msg**) wówczas jest on rzutowany na wskaźnik do klasy **text_msg**, i następnie wywoływane są metody pobierające wskaźnik do napisu oraz pozycję tekstu na ekranie. Następnie wywoływana jest metoda obiektu wyświetlacza (**display**), wypisująca tekst na wyświetlaczu LCD na zadanej pozycji. W przypadku, gdy mamy do czynienia z obiektem typu graficznego (**graph_msg**), wówczas jest on rzutowany na wskaźnik do tego obiektu, a następnie wywoływana jest metoda pobierająca wskaźnik na strukturę opisującą obrazek (**img_def**). Następnie wskaźnik ten jest przekazywany do metody **put_bitmap()** obiektu wyświetlacza, który jest odpowiedzialny za wyświetlenie bitmapy. Obiekt **keys** klasy **graph_key** jest odpowiedzialny za odczyt stanu joysticka oraz wysyłanie komend do serwera wyświetlania w zależności od kierunku przechylenia joysticka.

Obiekt inicjalizowany jest za pomocą konstruktora, który jako argument przyjmuje referencję do klasy **display_server**. W konstruktorze inicjalizowane są porty GPIO, do których podłączono poszczególne styki joysticka. Odczyt stanu joysticka oraz wysyłanie komunikatów jest realizowane przez metodę główną wątku **main() – listing 8**.

Wykrywanie kierunku przechylenia joysticka jest realizowane w pętli głównej wątku na zasadzie wykrywania zbocza na linii GPIO, do której dołączono styki joysticka. W przypadku wykrycia zbocza opadającego na którymś z wejść program ustala, jaki kierunek wskazuje joystick, następnie do kolejki FIFO serwera wyświetlania jest przekazywany obiekt komunikatu metodą **send_message()**.

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

Listing 8.

```
//Task/thread method
void graph_key::main()
{
        //Previous key variable
        static short p_key = -1;
        //Graphics message class
        static graph_msg gmsg;
        //Text message class
        static text_msg tmsg(„Wcisnales OK”);
        for(;;)
        {
                //Get key
                short key = get_key();
                //Check if any key is pressed
                if(key!=0 && p_key==0)
                {
                        switch(key)
                        {
                        case KEY_OK:
                                disp_srv.send_message(tmsg);
                                break;
                        case KEY_LEFT:
                                gmsg.set_image(images::isixlogo_bmp);
                                disp_srv.send_message(gmsg);
                                break;
                        case KEY_RIGHT:
                                gmsg.set_image(images::disk_bmp);
                                disp_srv.send_message(gmsg);
                                break;
                        case KEY_UP:
                                gmsg.set_image(images::printer_bmp);
                                disp_srv.send_message(gmsg);
                                break;
                        case KEY_DOWN:
                                gmsg.set_image(images::scriba_bmp);
                                disp_srv.send_message(gmsg);
                                break;
                        }
                }
                //Previous key assignement
                p_key = key;
                //Wait short time
                isix::isix_wait( isix::isix_ms2tick(DELAY_TIME) );
        }
}
```

Dodatkowe informacje na temat systemu ISIX-RTOS, nowości ze świata STM32 oraz prezentacje multimedialne są dostępne na stronie *www.stm32.eu*.