Obsługa kolorowych wyświetlaczy graficznych w systemie ISIXRTOS

Producenci nowoczesnych urządzeń elektronicznych coraz częściej przyzwyczajają nas do kolorowych wyświetlaczy oraz interfejsów graficznych nawet w urządzeniach, w których wydaje się on zbędny. Aby nadążyć za nowoczesnymi trendami coraz częściej jesteśmy zmuszeni do stosowania kolorowych wyświetlaczy nawet w najprostszych konstrukcjach. Tworzenie oprogramowania kolorowego interfejsu graficznego od podstaw jest często mozolną pracą, której nakład niejednokrotnie przewyższa czas poświęcony na oprogramowanie podstawowej funkcjonalności urządzenia. Wychodząc naprzeciw zapotrzebowaniu użytkowników system operacyjny ISIX-RTOS wyposażono w bibliotekę służącą do budowy kolorowych interfejsów graficznych o nazwie libgfx. Biblioteka ta udostępnia niezależny od sprzętu interfejs klas umożliwiający budowanie własnych interfejsów użytkownika, bez konieczności tworzenia wszystkiego od podstaw.

Biblioteka *libgfx* stanowiąca część systemu ISIX charakteryzuje się następującymi właściwościami:

- Obsługa kolorowych wyświetlaczy graficznych o dowolnej głębi kolorów i rozdzielczości ekranu.
- Pełna niezależność aplikacji od sprzętu dzięki architekturze warstwowej.
- Minimalizacja użycia pamięci RAM dzięki operowaniu na buforze wyświetlacza.
- Budowa obiektowa z wykorzystaniem wszelkich zalet standardu C++ 11 (ISO/IEC 14882:2011).
- Możliwość rozbudowy dzięki łatwości tworzenia własnych komponentów graficznych.

Architektura biblioteki libgfx

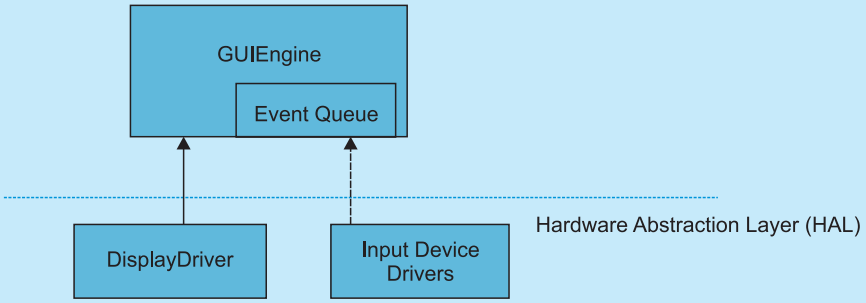
Biblioteka *libgfx* stanowi część systemu ISIX i napisano ją w języku C++11. Dzięki nowym funkcjonalnościom języka udało się w łatwy sposób zaimplementować mechanizm slotów i sygnałów znanych np. z QT. Zastosowanie sygnałów/slotów pozwala na znaczne rozluźnienie powiązań pomiędzy klasami pozwalając na uniknięcie nadmiernego wykorzystania mechanizmu dziedziczenia. Architekturę biblioteki przedstawiającą podział na warstwy przedstawiono na **rysunku 1**.

Dolną warstwę biblioteki stanowi część oddzielająca jądro systemu graficznego od sterowników urządzeń. Sterownik wyświetlacza graficznego udostępnia niezależny od sprzętu interfejs programistyczny umożliwiający rysowanie podstawowych elementów graficznych. Sterowniki urządzeń wejściowych są odpowiedzialne za zgłaszanie zdarzeń (np. wciśnięcie klawisza, dotknięcie ekranu dotykowego itp.), które są umieszczane w kolejce zdarzeń systemu graficznego – to jedne elementy biblioteki, które zależą od użytego sprzętu i muszą zostać dopasowane, do danego urządzenia.

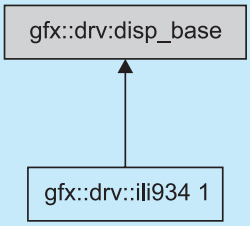
Interfejs sterownika wyświetlacza stanowi wirtualna klasa bazowa *disp_base* umieszczona w przestrzeni nazw *gfx::drv*. Rzeczywisty sterownik wyświetlacza powinien dziedziczyć z klasy bazowej *disp_base* i implementować wszystkie publiczne interfejsy tej klasy. **Rysunek 2** przedstawia hierarchię klas implementującą sterownik dla wyświetlacza *ili9341*, który zostanie przedstawiony w dalszej części przykładu.

W zależności od zastosowanego kontrolera wyświetlacza, część operacji może być wspomagana sprzętowo, natomiast część może być realizowana na drodze programowej, co zależy od rodzaju zastosowanego kontrolera. Do prawidłowej pracy biblioteki graficznej, sterownik musi implementować następujące metody wirtualne:

- *void set_pixel (coord_t x, coord_t y, color_t color)* – rysuje punkt o zadanym kolorze na pozycji wyświetlacza.
- *color_t get_pixel (coord_t x, coord_t y)* – zwraca kolor piksela na zadanej pozycji.
- *void clear (color_t color)* – wypełnia całe pole wyświetlacza zadanym kolorem.



Rysunek 1. Architektura biblioteki przedstawiająca podział na warstwy



Rysunek 2. Hierarchia klas implementująca sterownik dla wyświetlacza ili9341

- *void fill (coord_t x, coord_t y, coord_t cx, coord_t cy, color_t color)* – wypełnia obszar o współrzędnych x, y i wielkości cx, cy zadanym kolorem.
- *void blit (coord_t x, coord_t y, coord_t cx, coord_t cy, coord_t src_y, const color_t *buf)* – przepisuje bitmapę z pamięci do fragmentu wyświetlacza o pozycji x, y i wielkości cx, cy.
- *void vert_scroll (coord_t x, coord_t y, coord_t cx, coord_t cy, int n, color_t c)* – przesuwa zawartość wyświetlacza o pozycji x, y i wielkości cx, cy o n-linii w górę lub w dół, w zależności od tego czy liczba n jest dodatnia, czy ujemna.
- *bool power_ctl (power_ctl_t mode)* – umożliwia sterowanie zasilaniem wyświetlacza oraz jego usypianiem.

Jest to minimalny zestaw metod, który jest niezbędny do prawidłowego działania wyświetlacza, i umożliwia ewentualne wykorzystanie wsparcia sprzętowego do wykonywania niektórych operacji, jeśli sterownik wyświetlacza je obsługuje.

Implementacja obsługi urządzeń wejściowych jest dużo prostsza i sprowadza się, do przekazania przez sterownik urządzenia wejściowego zdarzenia do kolejki komunikatów systemu graficznego. W przypadku klawiatury, należy przekazać zdarzenie *EV_KEY*, w wypadku panelu dotykowego zdarzenie *EV_TOUCH*, a w wypadku myszki zdarzenie *EV_MOUSE*, wraz z odpowiednimi parametrami.

W systemie znajduje się przykładowy sterownik dla taniego wyświetlacza bazującego na kontrolerze *ili9341*, który może stanowić wzorzec podczas tworzenia sterowników urządzeń dla innych typów wyświetlaczy.

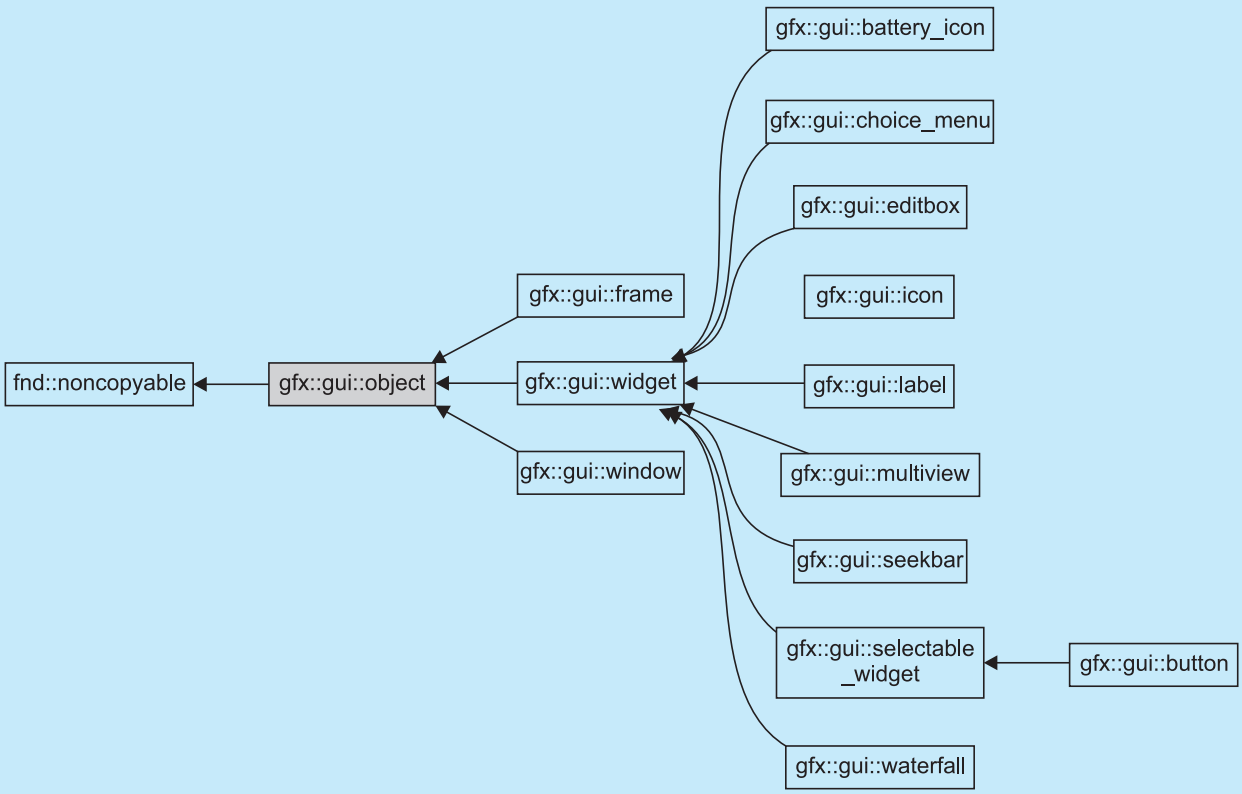
Korzystając z warstwy abstrakcji sprzętu HAL, kolejna warstwa biblioteki udostępnia niezależny od sprzętu zestaw komponentów umożliwiających tworzenie aplikacji graficznych. Hierarchię klas właściwej biblioteki graficznej przedstawiono na **rysunku 3**.

Klasa *object* jest podstawową klasą bazową, z której dziedziczą wszystkie pozostałe klasy biblioteki graficznej. Implementuje ona podstawowy mechanizm slotów i sygnałów wykorzystywanych w aplikacji do podłączania się pod poszczególne zdarzenia zgłaszane przez bibliotekę graficzną. Aby podłączyć się pod dane zdarzenie wystarczy wywołać funkcję *connect(event_signal evt_h, event::evtype evt_t)*, która jako argument przyjmuje obiekt funkcyjny przypisany do zdarzenia, oraz identyfikator zdarzenia, do którego będzie on przypisany. Każde zdarzenie o zgodnym identyfikatorze wygenerowane przez podsystem spowoduje wywołanie odpowiedniej funkcji/metody w momencie jego wystąpienia. Oprócz wymienionych wcześniej zdarzeń związanych bezpośrednio z urządzeniem wejściowym np. *EV_KEY* poszczególne komponenty mogą zgłaszać własne zdarzenia:

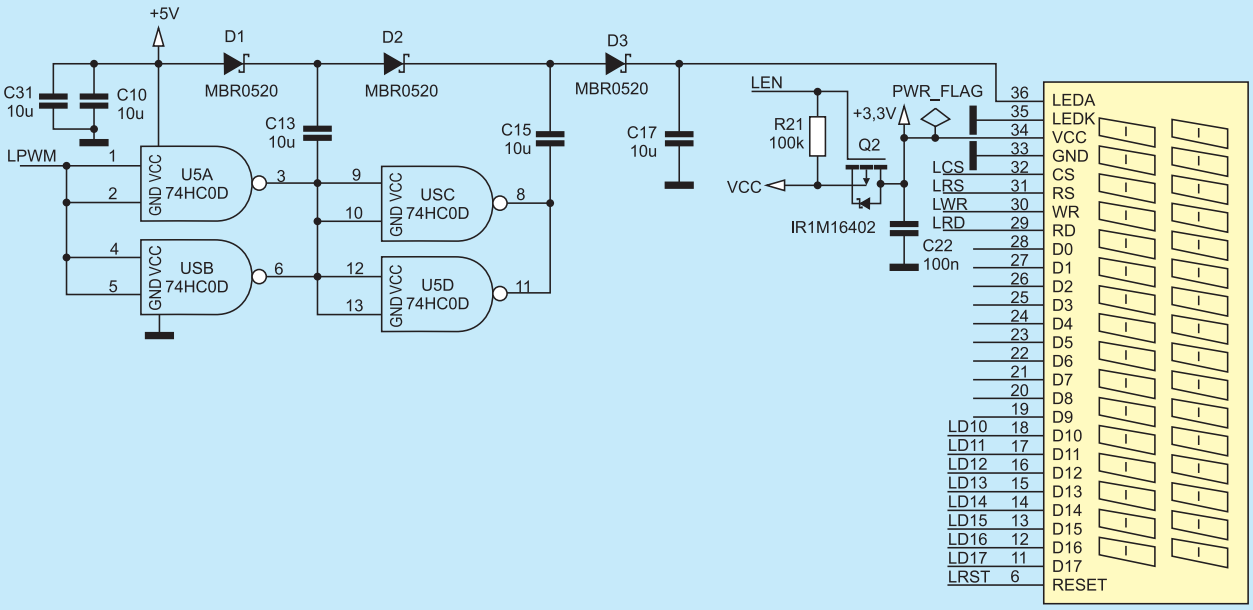
EV_CHANGE – Zdarzenie generowane w wyniku zmiany zawartości danego komponentu np. zmiany zawartości tekstowej w przypadku kontrolek tekstowych.

EV_HOTPLUG – Zdarzenie generowany przez podsystem urządzeń zewnętrznych, np. wyniku dołączenia lub odłączania urządzeń typu *Plug and Play* np. myszy USB.

EV_CLICK – Zdarzenie generowane w wyniku kliknięcia w dany komponent, np. wciśnięcie guzika, wybranie elementu z menu itp.



Rysunek 3. Hierarchia klas właściwej biblioteki graficznej



Rysunek 4. Sposób dołączenia wyświetlacza do zestawu STM32Butterfly

EV_FOCUS – Zdarzenie generowane w wyniku uzyskania widoczności przez okno.

EV_TIMER – Zdarzenie generowane przez klasę timer biblioteki graficznej

EV_WINDOW – Zdarzenie generowane przez użytkownika, kierowane do danego okna

Podstawową klasą, od której należy rozpocząć budowę interfejsu graficznego jest klasa *frame*, która fizycznie jest powiązana ze sterownikiem wyświetlacza oraz urządzeniami wejściowymi. W konstruktorze przyjmuje ona referencję do obiektu klasy *base_display* oraz udostępnia metodę *report_event()* służącą do zgłaszania zdarzeń przez urządzenia wejściowe. Każdy wyświetlacz fizyczny dostępny w urządzeniu powinien mieć przypisany obiekt klasy *frame*. Ta klasa realizuje funkcję menedżera okien znaną z większych systemów graficznych i zawiera zestaw funkcji służących do zarządzania oknami, takim jak: dodawanie okien, usuwanie okien, przesłanianie okien itp. W najprostszym przypadku do obiektu klasy *frame* powinno być przypisane przynajmniej jedno okno. Naturalnie istnieje możliwość dodawania większej liczby okien, które w zależności od fizycznego rozmieszczenia w obiekcie ramki (frame) mogą ulegać przesłonięciu: całkowitemu, częściowemu lub być całkowicie rozdzielne. W danym czasie aktywne może być tylko jedno okno, a jego wybór jest możliwy przez wywołanie metody *set_focus*, która jako argument przyjmuje wskaźnik do okna. Należy pamiętać, że tylko aktualnie aktywne okno otrzymuje zdarzenia związane z urządzeniami wejściowymi, takimi jak np. wciśnięcie klawisza *EV_KEY*. Do okna można dołączyć dowolną liczbę obiektów dziedziczących z klasy bazowej typu *widget*. Obiekty te implementują podstawowe komponenty interfejsu graficznego, takie jak: guziki, menu, ikony, pola edycyjne itp. W bibliotece zdefiniowano wszystkie najczęściej używane widgety: pola edycyjne, ikony, menu, guziki. Możemy również stworzyć własne obiekty dziedziczące z klasy *widget* implementujące inne niestandardowe komponenty, które nie zostały zdefiniowane w bibliotece. Bazując na powyższych komponentach możemy w prosty sposób stworzyć całe środowisko graficzne.

Oprócz komponentów bezpośrednio widocznych na ekranie biblioteka zawiera również odrębną klasę *timer*, której zadaniem jest generowanie zdarzeń związanych z czasem. Mamy tutaj możliwość stworzenia timera jednorazowego (tzw. *one shoot)* lub cyklicznego.

Przykład praktyczny

Celem zapoznania się z możliwościami biblioteki przygotowano aplikację demonstracyjną dla zestawu ewaluacyjnego *STM32Buttefly*. Ten zestaw nie ma żadnego wyświetlacza, więc jest konieczne dołączenie zewnętrznego wyświetlacza do portów GPIO mikrokontrolera. Do demonstracji wykorzystano wyświetlacz o przekątnej ekranu 2,8" z kontrolerem *ILI9341*, który ma rozdzielczość 320×240 pikseli oraz 262 tys. kolorów. Z uwagi na ograniczoną przepustowość magistrali realizowanej w sposób programowy wybrano pracę wyświetlacza z ograniczoną do 65 tys. liczbą kolorów, zapewniając znaczne przyśpieszenie transferu danych. Wyświetlacz z kontrolerem ILI9341 ma niczym niewyróżniające się parametry, jednak jego główną zaletą jest jego bardzo niska cena (poniżej 40 złotych). Do podświetlania tła producent zastosował trzy połączone szeregowo białe diody LED w związku z tym, do jego zasilania konieczna jest dodatkowa przetwornica podwyższająca napięcie. Schemat ideowy pokazujący sposób dołączenia wyświetlacza do zestawu *STM32Butterfly* pokazano na **rysunku 4**.

Napięcie zasilające wyświetlacz (VCC) powinno mieć wartość 3,3 V i jest dołączone za pośrednictwem dodatkowego tranzystora Q2 umożliwiające programowe odłączanie zasilania wyświetlacza. Przetwornicę napięcia dla podświetlania zrealizowano za pomocą klasycznego pojemnościowego potrajacza napięcia na bramkach NAND

Tabela 1. Sposób dołączenia wyświetlacza do GPIO mikrokontrolera

LCS	GPIOC4
LRS	GPIOC5
LWR	GPIOC7
LRD	GPIOC8
LRST	GPIOC9
LD10-LD17	GPIOE0-GPIOE8

Listing 1. Przypisania obiektu sterownika wyświetlacza do menedżera okien *frame*
```
//Constructor
tft_tester()
    : gdisp( gbus ), frame(gdisp)
    {
        start_thread( STACK_SIZE, TASK_PRIO );
    }
```

układu 74HC7400, który jest zasilany napięciem +5 V. Sygnał *LPWM* (linia GPIOC6) taktujący potrajacz jest generowany przez układ czasowo–licznikowy pracujący w trybie PWM z częstotliwością około 100 kHz. Zaletą użycia układu czasowo–licznikowego jest możliwość programowej regulacji jasności podświetlania wyświetlacza poprzez zmianę współczynnika wypełniania sygnału PWM. Nazwy sygnałów z płytki wyświetlacza oraz odpowiadające im porty GPIO mikrokontrolera, do których należy je doprowadzić wymieniono w **tabeli 1**.

Przykład demonstrujący działanie biblioteki graficznej jest dostępny wraz z pozostałymi dla systemu ISIX w katalogu *simple_hw/gfxdemo*. Przykład należy skompilować z wykorzystaniem kompilatora *GCC* (polecenie *make*), w wyniku czego powinien powstać plik *tftdemo.elf*. Następnie do zestawu należy dołączyć JTAG zgodny z interfejsem *oocdlink* i wydać polecenie *make program*, co powinno spowodować zaprogramowanie mikrokontrolera. Do zaprogramowania zestawu możemy również wykorzystać standardowy bootloader i program „*Flash Loader Demonstrator*". Należy jedynie w pliku *Makefile* w polu *FORMAT* zmienić format pliku wynikowego z „*elf*" na „*hex*". Po zaprogramowaniu zestawu program powinien się uruchomić, a na wyświetlaczu powinniśmy ujrzeć efekt działania programu demonstracyjnego. Wykorzystując zintegrowany joystick możemy wybierać poszczególne widgety, edytować ich pola, oraz zapoznać się z poszczególnymi komponentami i sposobem ich działania.

Kod demonstrujący użycie biblioteki graficznej jest zrealizowany w klasie *tft_tester*. Obiekt *ft* klasy *tft_tester* jest tworzony statycznie bezpośrednio w funkcji *main()*. Klasa zawiera prywatne obiekty składowe: obiekt *gdisp* klasy *ili9341* stanowiący sterownik wyświetlacza graficznego oraz obiekt *frame* klasy *frame* stanowiący podstawowy menedżer okien przypisany do wyświetlacza. Przypisanie obiektu sterownika wyświetlacza do menadżera okien *frame* wykonywane jest w liście inicjalizującej konstruktora, co pokazano na **listingu 1**.

W konstruktorze równocześnie tworzony jest wątek odpowiedzialny za obsługę interfejsu graficznego. Główny kod aplikacji realizowany jest przez metodę *windows_test()* wywołaną z metody głównej *main()* implementującej wątek, obsługi interfejsu graficznego (**listing 2**).

Pierwszą czynnością jest włączenie oraz inicjalizacja sterownika wyświetlacza graficznego poprzez wywołanie metody *power_ctl* z argumentem *ctl_t::on*. Następnie jest tworzony obiekt okna głównego aplikacji *win*, o wielkości 200×300 pikseli i początku w punkcie 10,10. Następnie tworzone są przykładowe widgety demonstracyjne: *but-*

Listing 2. Główny kod aplikacji realizowany przez metodę *windows_test()*
```
void windows_test()
{
    gdisp.power_ctl( power_ctl_t::on );
    window win( rectangle( 10, 10, 200, 300), frame, window::flags::border |window::flags::selectborder );
    button btn( rectangle(20, 20, 100 , 40), layout(), win );
    choice_menu choice1( rectangle(20, 147, 178, 120), layout(), win, choice_menu::style::normal );
    seekbar seek( rectangle(20, 65, 160 , 20), layout(), win );
    editbox edit( rectangle(20, 110, 160 , 30), layout(), win );
    edit.readonly(true);
    m_edit = &edit;
    seek.value( 50 );
    btn.caption(„Button");
    edit.value(„Text edit value");
    choice1.items( menu1 );
    //Connect windows callback to the main window
    win.connect(std::bind(&tft_tester::window_callback,this,std::placeholders::_1), event::evtype::EV_KEY); //1
    btn.connect(std::bind(&tft_tester::on_click,this,std::placeholders::_1),event::evtype::EV_CLICK); //[2]
    btn.pushkey( gfx::input::kbdcodes::enter);
    choice1.connect(std::bind(&tft_tester::on_select_item,this,std::placeholders::_1),event::evtype::EV_CLICK);
    seek.connect(std::bind(&tft_tester::on_seek_change,this,std::placeholders::_1),event::evtype::EV_CLICK);
    frame.set_focus( &win ); //[3]
    frame.execute(); // [4]
}
```

Listing 3. Zmiana aktywnego widgetu w wyniku wciśnięcia klawisza
```
bool window_callback( const gfx::gui::event &ev )
{
    using namespace gfx::input;
    bool ret {};
    if( ev.keyb.stat == detail::keyboard_tag::status::DOWN )
    {
        auto win = static_cast<gfx::gui::window*>(ev.sender);
        if( ev.keyb.key == kbdcodes::os_arrow_left && !m_edit_mode)
        {
            win->select_prev();
            ret = true;
        }
        else if( ev.keyb.key == kbdcodes::os_arrow_right && !m_edit_mode)
        {
            win->select_next();
            ret = true;
        }
        if( ev.keyb.key == kbdcodes::enter && m_edit == win->current_widget())
        {
            m_edit_mode = !m_edit_mode;
            m_edit->readonly( !m_edit_mode );
            dbprintf(„Toggle edit mode %i", m_edit_mode );
        }
    }
    return ret;//   Need redraw
}
```

ton, choice_menu, editbox, seekbar w określonych pozycjach w oknie oraz przypisywane są im przykładowe wartości początkowe. Po zakończeniu inicjalizacji pod zdarzenie *EV_KEY* obiektu *win* jest podłączana metoda *window_callback*, której zadaniem jest zmiana aktywnego widgetu w wyniku wciśnięcia odpowiedniego klawisza (**listing 3**).

Działanie tej metody sprowadza się do sprawdzenia czy wybrano kierunek *up* / *down* na joysticku i w zależności od tego odpowiednio następuje wybór kolejnego widgetu w oknie głównym poprzez wywołanie metody *select_next()*/*select_prev()* na rzecz obiektu *win*. Zmiana widgetu jest blokowana wówczas, gdy zmienna *edit_mode* ma wartość *true*, co oznacza, że okno jest w trybie edycyjnym. Zmiana wartości *edit_mode* następuje w wyniku wciśnięcia klawisza *ENTER*, czyli wciśnięciu joysticka. Blokada przełączania widgetów ma na celu umożliwienie obsługi interfejsu ograniczoną ilością przycisków joysticka.

Wróćmy teraz do metody *window_test*, gdzie w **[2]** do poszczególnych widgetów naszego przykładu do zdarzeń *EV_CLICK* podłączane są odpowiednie metody, których zadaniem jest wyświetlenie informacji, że dany widget został wciśnięty na konsoli szeregowej. W rzeczywistej aplikacji, w tym miejscu zazwyczaj będą wykonywana jakieś inne czynności związane z wyborem danego elementu. Po podłączeniu wszystkich wymaganych metod obsługi zdarzeń, w **[3]** jest wywoływana metoda *focus* na rzecz obiektu *frame*, która jako parametr przyjmuje adres obiektu klasy *win*, w wyniku czego następuje wybranie danego okna jako głównego. Ostatnią czynnością jest wywołanie na rzecz obiektu *frame* metody *execute()*, która powoduje rozpoczęcie przetwarzania komunikatów zdarzeń, a więc uruchomi obsługę interfejsu GUI w kontekście procesu, z którego została wywołana. Metoda ta nigdy nie powraca, dlatego należy pamiętać, aby w wątku obsługi interfejsu graficznego wywołać ją jako ostatnią.

Zakończenie

Dzięki uniwersalnej bibliotece graficznej *libgfx* tworzenie nawet skomplikowanych interfejsów użytkownika staje się stosunkowo łatwe. Do dyspozycji mamy szereg podstawowych komponentów graficznych, użytecznych w najczęściej spotykanych przypadkach, a gdy staną się one niewystarczające w prosty sposób możemy rozszerzyć funkcjonalność biblioteki tworząc nowe elementy. Dodatkowa warstwa abstrakcji sprzętu umożliwia całkowite uniezależnienie od zastosowanego wyświetlacza graficznego zapewniając podczas cyklu życia produktu szybką wymianę wyświetlacza na inny model. Jedyną czynnością konieczną do wykonania jest przygotowanie odpowiedniego sterownika urządzenia. Jest to o tyle istotne, że często zdarza się, iż cykl produkcji danego typu wyświetlacza jest stosunkowo krótki i często stajemy przed koniecznością wymiany jednego typu wyświetlacza na inny.

Lucjan Bryndza EP (SQ5FGB)