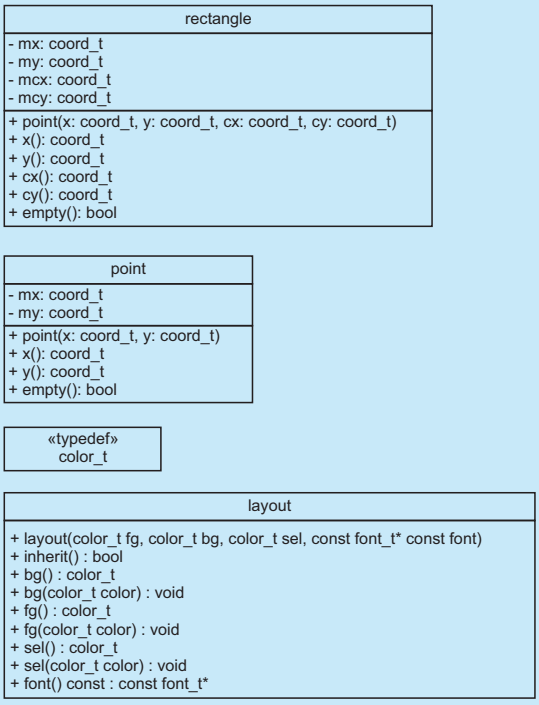
ISIXRTOS v3 mini system operacyjny dla mikrokontrolerów rodziny M0/M3/M4/M7 (9)

Biblioteka libgfx

W dwóch ostatnich odcinkach opracowaliśmy odrębne sterowniki wyświetlacza oraz panelu dotykowego dla biblioteki graficznej libgfx. W ostatnim odcinku kursu dotyczącym tematów graficznych połączymy wszystko w jedną całość, aby ostatecznie otrzymać w pełni funkcjonalny program demonstracyjny reagujący na dotyk. Omówimy również szczegóły budowy biblioteki graficznej od strony użytkownika, tak aby pokazać w jaki sposób stworzyć aplikację graficzną.

Podstawowa hierarchia klas

Biblioteka graficzna **libgfx** służy do tworzenia prostych interfejsów graficznych na niewielkich ekranach TFT LCD i została napisana w języku C++17, przy czym szczególny nacisk położono na minimalizację użycia pamięci. Do prawidłowego działania wystarczy mały mikrokontroler z rdzeniem Cortex-M3 oraz pamięcią RAM o wielkości min. 32 kB oraz pamięcią FLASH o wielkości min. 64 kB. Motywacją do opracowania biblioteki było niezadowolenie z aktualnie dostępnych rozwiązań, które albo były napisane w języku C, albo były dostępne na niewolnych licencjach bez dostępu do kodu źródłowego, lub wymagały stosunkowo dużych zasobów pamięciowych.



Rysunek 1. Podstawowe elementy odpowiedzialne za rozmieszczenie komponentów graficznych oraz zarządzanie kolorem

Biblioteka została napisana w oparciu o model obiektowy, ułatwiający znacząco tworzenie aplikacji graficznych, przy czym zdecydowano się na tworzenie interfejsu graficznego bezpośrednio w kodzie bez żadnych dodatkowych generatorów, które są zwyczajowo źródłem problemów, w szczególności, gdy chcemy zrealizować jakieś nietypowe zadanie. Dzięki takiemu podejściu interfejs biblioteki jest stosunkowo prosty oraz umożliwia tworzenie aplikacji graficznych w łatwy sposób, poprzez tworzenie obiektów klas odpowiadających wizualnie poszczególnym interfejsom graficznym na ekranie.

Podstawowymi elementami odpowiedzialnymi za rozmieszczenie komponentów graficznych oraz zarządzanie kolorem są obiekty pokazane na **rysunku 1**. Klasa **rectangle** służy do określenia rozmiaru oraz wielkości elementów graficznych na ekranie. Punkt początkowy określany jest za pomocą zmiennych **x** oraz **y**, natomiast rozmiar określany jest za pomocą zmiennej przechowującej szerokość obiektu **cx**, oraz zmiennej **cy** przechowującej wysokość obiektu. Klasa **point** określa pozycję pojedynczego punktu na ekranie i przechowuje współrzędne punktu w postaci zmiennych **x** oraz **y.** Definicja **color_t** służy do przechowywania koloru poszczególnych elementów.

Kolor możemy zdefiniować za pomocą funkcji:

```
static inline constexpr color_t rgb( unsigned char R, unsigned char G, unsigned char B );
```

Funkcja jako argumenty przyjmuje składowe koloru w postaci nasycenia składowej czerwonej, zielonej, oraz niebieskiej. Kolor możemy również określić za pomocą zestawu predefiniowanych zestawów **color** znajdujących się w przestrzeni nazw **gfx**.

Wygląd okien oraz widżetów opisywany jest za pomocą pomocniczej klasy **layout**, która zawiera podstawowe informacje o wyglądzie okna, kolorze tła, kolorze czcionki, kolorze przycisków, kolorze obramowań, oraz przechowuje wskaźnik do obiektu czcionki używanej do tworzenia napisów.

Kompletny diagram klas biblioteki graficznej od strony interfejsu użytkownika pokazano na **rysunku 2**. Koncepcja tworzenia aplikacji graficznych z użyciem biblioteki opiera się na tworzeniu hierarchii

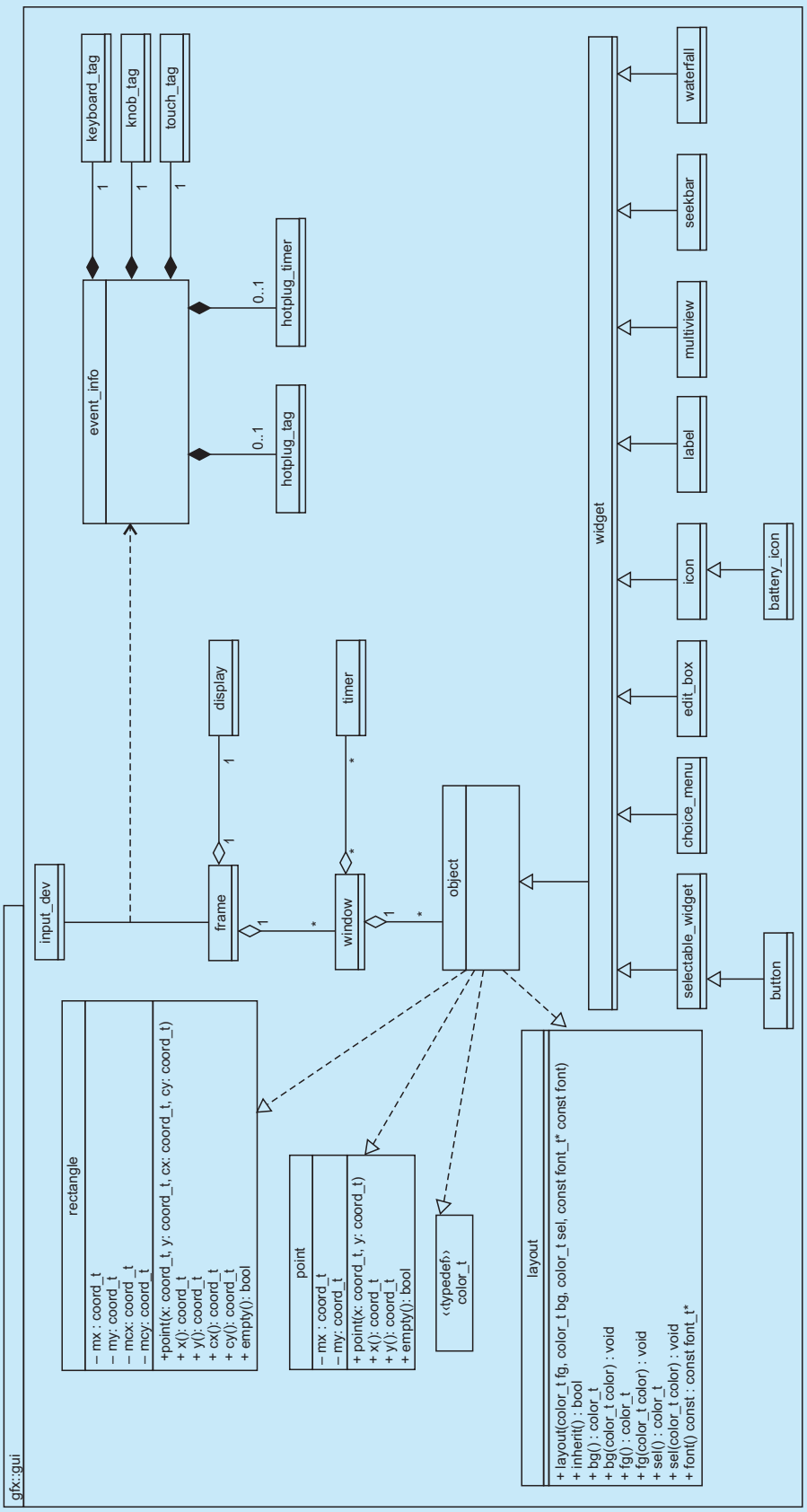
komponentów, takich jak: ramki, okna oraz elementy graficzne zwane widżetami. Podstawowym obiektem będącym właścicielem okien jest obiekt klasy **frame**, który powiązany jest z klasą bazową fizycznego sterownika, realizującego wyświetlanie na ekranie. Jedna instancja obiektu **frame** może być fizycznie powiązana tylko z jedną instancją klasy sterownika graficznego. Poza wyświetlaniem danych potrzebna jest również interakcja aplikacji z użytkownikiem za co odpowiedzialna jest metoda:

```
int report_event( const
input::event_info &event );
```

Metoda ta jest wywoływana przez sterowniki urządzeń wejściowych w wyniku reakcji na zdarzenia np. wciśnięcie klawisza, dotknięcie ekranu, co zostało omówione w poprzednim odcinku. Obiekt tej klasy odpowiedzialny jest również za zarządzanie oknami, do którego może być przypisane wiele okien (obiektów klasy **window**). Okna mogą być widoczne na ekranie lub mogą być ukryte. W tym samym czasie tylko pojedyncze okno może być aktywne (*Focus*) i tylko do aktywnego okna trafiają wszystkie zdarzenia pochodzące od urządzeń wejściowych. Do poszczególnych okien mogą być również przypisane obiekty klasy **timer**, które umożliwiają generowanie zdarzeń w ściśle określonych interwałach czasowych. W poszczególnych oknach mogą być umieszczane właściwe komponenty graficzne odpowiedzialne za integrację z użytkownikiem. Podstawową klasę bazową dla wszystkich widżetów stanowi klasa **object**, która jest odpowiedzialna za obsługę zdarzeń. Natomiast podstawową klasą bazową dla wszystkich komponentów graficznych stanowi klasa **widget**. Stanowi ona podstawowy interfejs odpowiedzialny za realizację operacji graficznych przez klasy podrzędne. Do dyspozycji twórcy aplikacji graficznej mamy zestaw następujących klas podrzędnych:

- Klasa **button** umożliwia utworzenie przycisków (klawiszy),
- Klasa **choice_menu** umożliwia tworzenie obiektów z napisami z możliwością wyboru opcji z listy,
- Klasa **edit_box** umożliwia tworzenie pól edycyjnych,
- Klasa **icon** odpowiedzialna jest za tworzenie prostych piktogramów (ikon),
- Klasa **battery_icon** umożliwia tworzenie inteligentnych wskaźników stanu naładowania baterii,
- Klasa **label** umożliwia tworzenie etykiet tekstowych z dowolnym kolorem tła oraz kolorem napisu na ekranie,
- Klasa **multiview** umożliwia utworzenie wieloliniowego pola edycyjnego z automatycznym przewijaniem tekstu do tyłu oraz do przodu,
- Klasa **seekbar** umożliwia tworzenie suwaków które mogą służyć do regulacji różnych wartości (np. poziom głośności itp.),
- Klasa **waterfall** umożliwia tworzenie wykresów wodospadowych, korzystających z transformaty Fouriera.

Pokazane komponenty często są spotykane w innych środowiskach graficznych, jednak, gdy będziemy potrzebować innego komponentu, który nie występuje na liście gotowych obiektów możemy w łatwy sposób przygotować własne implementacje.



Rysunek 2. Kompletny diagram klas biblioteki graficznej od strony interfejsu użytkownika

Zdarzenia wejściowe

Biblioteka graficzna **libgfx** jest oparta o system zdarzeń obsługiwany w wątku głównym klasy **frame**. Zdarzenia generowane są przez urządzenia wejściowe i przekazywane do obiektów w wyniku reakcji użytkownika np. wciśnięcia klawisza, dotknięcia ekranu itp. Sterowniki systemu operacyjnego mogą również generować zdarzenia np. w wyniku włożenia pendrive do złącza USB, są to tzw. zdarzenia systemowe. Wątek główny przekazuje zdarzenia podchodzące od urządzeń wejściowych do wszystkich aktualnie widocznych okien, odpowiedzialny jest również za rysowanie poszczególnych widocznych okien i komponentów na ekranie.

Biblioteka nie narzuca ilości dostępnych ekranów. Jedynym warunek jest taki, żeby każda klasa **frame** odpowiadała za generowanie obrazu dla pojedynczego wyświetlacza. Tworząc odpowiednią ilość takich obiektów możemy obsługiwać dowolną liczbę wyświetlaczy w pojedynczym systemie. Poza zdarzeniami zewnętrznymi w implementacji biblioteki generowane są również zdarzenia wewnętrzne np. informacja o uzyskaniu widoczności przez okno, informacja o utracie widoczności, informacja o odrysowaniu elementu. Listę dostępnych zdarzeń biblioteki pokazano w **tabeli 1**. Zdarzenia mogą być wykorzysane wewnętrznie przez komponenty graficzne np. zdarzenie od wciśnięcia klawisza może wpisywać kolejne litery w polu edycyjnym. Istnieje również możliwość podłączenia lub odłączenia własnego handlera związanego z określonym zdarzeniem do dowolnego obiektu biblioteki, co realizowane jest za pomocą metod klasy bazowej **object**:

```
event_handle connect( event_signal evt_h,
event::evtype evt_t  );
void disconnect( event_handle evt );
```

obiekt typu **event_signal** zdefiniowano z użyciem mechanizmu **std::function<>** :

```
using event_signal = std::function<bool(const event&)>
```

Dzięki temu rozwiązaniu jako funkcję obsługi zdarzeń możemy wykorzystać funkcję lambda, funkcje klasyczne, a także dzięki zastosowaniu **std::bind** metody klas, których funkcja jako argument przyjmuje referencję do klasy event oraz powinna zwracać wartość *true*, jeśli wybrany komponent ma zostać odrysowany na ekranie.

Anatomia klas frame oraz window oraz widget

Po zapoznaniu się z ogólną koncepcją biblioteki przechodzimy do szczegółowego omówienia interfejsów klas przydatnych podczas tworzenia własnych aplikacji. Pierwszym krokiem jest utworzenie klasy **frame**, a następnie dodanie do niej okna głównego aplikacji lub ewentualnie kilku okien. Przy tworzeniu instancji klasy **frame** należy skorzystać z konstruktora:

```
frame( drv::disp_base &display, color_t color =
color::Black )
```

Jako argumenty należy przekazać referencję do obiektu sterownika wyświetlacza oraz kolor tła, którym będzie wypełniony wyświetlacz, gdy nie będzie widoczne na nim okno. Po utworzeniu obiektu należy przekazać referencję do niego sterownikowi lub sterownikom urządzeń wejściowych, które w reakcji na zdarzenie zewnętrzne powinny wywołać metodę:

```
int report_event( const input::event_info &event );
```

do której powinny przekazać wypełnioną strukturę event opisującą określone zdarzenie.

Po zakończeniu inicjalizacji wszystkich potrzebnych obiektów należy pamiętać, aby wywołać metodę:

```
/** Execute gui main loop */
void execute();
```

której zadaniem jest obsługa głównej pętli zdarzeń biblioteki graficznej.

Kolejnym krokiem po utworzeniu ramki jest dodanie lub usuwanie okien powiązanych z danym wyświetlaczem co możemy zrealizować za pomocą metod:

```
//Add widget to frame
void add_window( window* window );
//Delete the widget
void delete_window( window* window );
```

Do ramki możemy dodać dowolną liczbę okien którymi będziemy zarządzać. Okna mogą zachodzić na siebie, pokrywać się częściowo lub całkowicie, przy czym w danym czasie tylko jedno okno może otrzymać Focus oraz otrzymywać zdarzenia od urządzeń wejściowych.

Do najważniejszych metod służących do zarządzania oknami należą:

```
int set_focus( window* win, window* back_win =
nullptr );
int pop_focus();
bool stack_empty() const;
window* get_active_window() const;
```

Pierwsza metoda ustawia wybrane okno jako widoczne na wierzchu, jednocześnie pozwalając na opcjonalne przekazanie wskaźnika do innego okna, które będzie zapisane na stosie okien.

Druga metoda umożliwia zdjęcie ze stosu poprzednio odłożonego okna oraz ustawia zdjęte okno jako widoczne. Metoda **stack_empty**(), zwraca wartość **true**, jeżeli stos okien jest pusty, natomiast ostatnia metoda pozwala na uzyskanie wskaźnika do aktualnie aktywnego okna.

Kolejną klasą, która stanowi właściwy kontener na komponenty graficzne jest klasa **window**. Obiekt tej klasy możemy utworzyć za pomocą konstruktora:

```
window( const rectangle &coord, frame &frm,
unsigned flags = 0 )
```

Jako pierwszy argument należy przekazać referencję do klasy **rectangle** określającej umiejscowienie okna w ramce, jako drugi argument należy przekazać referencję do obiektu ramki, która będzie właścicielem okna. Jako ostatni argument możemy przekazać opcjonalne argumenty określające atrybuty okna. Atrybuty powinny stanowić kombinację bitów określających wypełnienie oraz obramowanie okien – **listing 1**. Flaga **fill** określa, czy okno powinno być wypełniane kolorem tła czy powinno pozostać niezmienione. Flaga **border** określa czy należy rysować obramowanie okna czy też nie. Flaga **selectboder** definiuje czy należy rysować obramowanie aktualnie aktywnego okna. Dodatkowa flaga **double_border** określa, czy obramowanie powinno być podwójnej grubości, czy nie.

Tabela 1. Lista dostępnych zdarzeń biblioteki

Zdarzenie	Struktura	Typ	Opis
EV_PAINT	window*	Wewn.	Okno zostało odrysowane
EV_WINDOW	window*	Wewn.	Zdarzenie wygenerowane przez użytkownika
EV_KEY	keyboard_tag	Zewn.	Wciśnięto klawisz
EV_KNOB	knob_tag	Zewn.	Zdarzenie od enkodera obrotowego
EV_TOUCH	touch_tag	Zewn.	Zdarzenie od ekranu dotykowego
EV_HOTPLUG	hotplug	Zewn.	Zdarzenie systemowe (dołączenie lub odłączenie urządzenia)
EV_CLICK		Wewn.	Kliknięcie w wybrany widżet
EV_CHANGE	argument	Wewn.	Zmiana zawartości widżetu
EV_FOCUS	window*	Wewn.	Uzyskanie lub utrata widoczności
EV_TIMER	timer	Wewn.	Zdarzenie pochodzące od licznika
EV_JOYSTICK	argument	Zewn.	Zdarzenie od zewnętrznego joysticka (USB)

Listing 1. Ostatni argument obiektu klasy window definiujący atrybuty okna stanowi kombinację bitów określających wypełnienie oraz obramowanie okien

```
enum  : unsigned {
        //Fill background
        fill = 0x01,
        //Draw border
        border = 0x02,
        //Select border
        selectborder = 0x04,
    };
};

struct style {
    enum  : unsigned {
        //Single border
        single_border = 0,
        //Double border
        double_border = 1,
    };
};
```

Po utworzeniu okna możemy dodawać lub usuwać poszczególne komponenty graficzne z wykorzystaniem metod:

```
//! Add widget
void add_widget( widget * const w );
//! Delete widget
void delete_widget( widget * const w );
```

W ramach aktywnego okna możemy również aktywować wybrany widżet za pomocą metod:

```
//! Select next item
void select_next();
//! Select prev item
void select_prev();
//! Select wiget directly
void select( widget * const w );
//! Select widget by coord
void select( const point& p );
```

Korzystając z powyższych metod możemy wybierać aktywne komponenty do przodu lub do tyłu według porządku ułożenia w oknie. Możemy również aktywować konkretny komponent podając wskaźnik do tego komponentu lub punkt na ekranie, w którym jest on zawarty.

Należy zaznaczyć, że zarówno zmiana aktywnego okna, jak i wybór widżetu w danym oknie nie jest zdefiniowany przez bibliotekę graficzną, ponieważ w przeciwieństwie do klasycznych komputerów, w systemach wbudowanych zestaw urządzeń wejściowych, czy ilość i rodzaj przycisków nie jest zazwyczaj określona. Twórca aplikacji powinien zarejestrować własne procedury obsługi zdarzeń dla wybranych zdarzeń wejściowych korzystając z metody *connect()*, należącej do obiektu klasy **frame**. Z metody tej należy wywołać stosowne funkcje służące do aktywacji wybranego okna czy widżetu w ramach danego okna. Takie podejście kosztem niewiele większego skomplikowania zapewnia w pełni uniwersalne podejście niezależne od dostępnych przycisków.

Ostatnim etapem tworzenia interfejsu graficznego jest dodanie poszczególnych widżetów do wybranych okien. W tym przypadku każdy widżet zawiera zestaw własnych metod w zależności od tego jaką funkcjonalność realizuje. Część metod jest wspólna dla wszystkich komponentów i pochodzi z klasy bazowej **widget** oraz **object**. Metoda **selectable()** umożliwia określenie czy dany komponent jest wybieralny czy nie, co jest przydatne np. w przypadku komponentów typu ikony, etykiety itp, które nie realizują żadnych interakcji z użytkownikiem:

```
void selectable(bool select_mode);
bool selectable() const
```

Metoda **is_modified()**, określa czy wybrany component uległ zmianie i czy konieczne będzie jego ponowne odrysowanie

```
bool is_modified() const;
```

Spójrzmy dla przykładu na klasę **button**, której zadaniem jest narysowanie przycisku. Przycisk tworzymy za pomocą następującego konstruktora:

```
button( rectangle const& rect,layout const& layout
,window &win )
```

Jako pierwszy argument przekazujemy pozycję, w oknie na której przycisk zostanie umieszczony, jako drugi argument przekazujemy referencje do obiektu layout określający wygląd przycisku, natomiast jako ostatni argument należy przekazać referencję do obiektu okna, która jest właścicielem przycisku. Tak utworzony klawisz naturalnie powinien posiadać napis, który możemy ustawić za pomocą metody:

```
void caption( const T caption );
```

Stanem specyficznym dla przycisku, jest jego stan wciśnięcia. Możemy zarówno zmienić jak i odczytać aktualny stan za pomocą metod:

```
void pushed( bool pushed );
bool pushed() const;
```

Przycisk po wciśnięciu przez użytkownika jest źródłem zdarzenia **EV_CLICK**, do którego możemy podłączyć własną procedurę z pomocą metody ***connect()***. Dzięki temu po wciśnięciu klawisza zostanie wywołana nasza metoda/funkcja, którą możemy wykorzystać do celów określonych przez aplikację. Naturalnie inne komponenty graficzne będą zawierać inne metody oraz zgłaszać inne zdarzenie specyficzne dla poszczególnych typów widżetów.

Przykład

Do zaprezentowania możliwości biblioteki graficznej przygotowano proste demo, którego efekt działania pokazuje **fotografia 1**. Ekran składa się z pojedynczego okna, w którym umiejscowiono najczęściej spotykane widżety: przycisk, suwak, pole edycyjne oraz pole wyboru. Za pomocą pola dotykowego możemy wybierać poszczególne elementy. Efektem działania aplikacji jest wyświetlanie na diagnostycznym terminalu szeregowym informacji o dotknięciu poszczególnych komponentów oraz inne specyficzne informacje dla danego widżetu. Główną część kodu stanowią dwa pliki, *tft_livedemo.cpp* oraz *tft_livedemo.hpp*, które implementują klasę **tft_livedemo (listing 2).** Składowe prywatne klasy zawierają omówiony w poprzednich odcinkach obiekt sterownika wyświetlacza oraz ekranu dotykowego, z którymi współpracuje obiekt **m_frame** stanowiący instancję klasy **frame**. Główny kod programu graficznego realizowany jest przez metodę **thread()** wykonywaną przez system ISIX w postaci oddzielnego wątku (**listing 3**).

Na początku wykonywana jest metoda inicjalizująca sterownik magistrali I²C, która jest potrzebna do obsługi panelu dotykowego. Jeśli inicjalizacja kontrolera nie powiedzie się wątek kończy działanie.

Listing 2. Implementacja klasy *tft_livedemo*

```
class tft_livedemo
{
    static const unsigned STACK_SIZE = 2048;
    static const unsigned TASK_PRIO = 3;
public:
    //Constructor
    //Start the first task
    void start() noexcept;
private:
    //! Main thread
    void thread() noexcept;
    //! Lib test
    void windows_test() noexcept;
    //A window calllback for select item
    bool window_callback( const gfx::gui::event &ev );
    //Buttons callback
    bool on_click( const gfx::gui::event &ev );
    //On select item
    bool on_select_item( const gfx::gui::event &ev );
    //On seek change
    bool on_seek_change( const gfx::gui::event &ev );
    //! Setup i2c bus
    int setup_i2c_bus() noexcept;
private:
    periph::display::bus::dsi m_dsi { „dsi” };
    periph::display::fbdev m_fb { „ltdc” };
    periph::display::otm8009a displl {m_dsi, „display”};
    gfx::drv::dsi_fb m_disp { m_fb,displl };
    gfx::gui::frame frame { m_disp };
    gfx::gui::editbox* m_edit {};
    bool m_edit_mode = false;
    periph::drivers::i2c_master m_i2c { „i2c1” };
    gfx::drv::ft6x06_touch m_touch { „display”, m_i2c, frame };
    isix::thread m_thr;
};
```

Listing 3. Główny kod programu graficznego realizowany jest przez metodę *thread()* wykonywaną przez system ISIX w postaci oddzielnego wątku

```
void tft_livedemo::thread() noexcept {
    if(setup_i2c_bus()) {
        return;
    }
    //Start touch screen only when init is ok
    m_touch.start();
    // Start the demo
    windows_test();
}
```

W kolejnym kroku wywoływana jest metoda **start()** klasy panelu dotykowego, co powoduje rozpoczęcie przetwarzania zdarzeń wejściowych. Po zakończeniu inicjalizacji wywoływana jest metoda **windows_test(),** która realizuje właściwy proces tworzenia aplikacji graficznej (**listing 4**).

Pierwszym etapem jest włączenie podświetlania wyświetlacza, co jest realizowane za pomocą metody **power_ctl()** klasy display. Następnym krokiem jest utworzenie okna głównego reprezentowanego przez obiekt **win**, które jest pomniejszone o 10 pikseli w stosunku do rozmiaru całego ekranu. W konstruktorze przekazujemy rozmiary oraz lokalizację okna, referencję, do ramki która jest właścicielem okna, oraz flagi które włączają obramowanie. Po utworzeniu okna przystępujemy do tworzenia poszczególnych elementów graficznych: przycisku, suwaka, pola edycyjnego oraz pola wyboru. Tworząc powyższe obiekty przekazujemy relatywną pozycję elementów w oknie, rozmiar obiektów, wygląd obiektu, oraz referencję do okna, które jest właścicielem widżetów. W kolejnym kroku za pomocą metody **caption()** uzupełniamy opis przycisku, za pomocą metody **value()** ustawiamy suwak w połowie, oraz dodatkowo pole wyboru uzupełniamy listą linii. Mamy już w zasadzie utworzone wszystkie elementy graficzne na ekranie pozostało nam jeszcze uzupełnienie aplikacji o metody obsługi zdarzeń, tak aby program wykonywał jakieś akcje w interakcji z użytkownikiem. Jednym z istotniejszych kroków jest podłączenie metody **window_callback()** pod zdarzenia **EV_KEY** (wciśnięcie klawisza), oraz **EV_TOUCH** zdarzenie dotykowe, w której należy oprogramować sposób wyboru (*Focus*) poszczególnych komponentów po dotknięciu ekranu lub wciśnięciu klawisza. W kolejnych krokach w poszczególnych komponentach podłączamy metody pod zdarzenia **EV_CLICK**, które będą wysyłane do aplikacji po wykonaniu określonych czynności. Ostatnim krokiem jest ustawienie focusa na okno główne aplikacji, oraz wywołanie metody **execute()** na rzecz obiektu **m_frame**. Od tego momentu wszystkie interakcje ze środowiskiem graficznym odbywają się za pomocą pętli zdarzeń.

Najbardziej istotną metodą jest metoda **window_callback(),** obsługująca wybór aktualnie aktywnego komponentu (**listing 5**). W pierwszym kroku zajmujemy się obsługą zdarzeń od klawiatury, której warunkiem początkowym jest wykrycie zdarzenia **EV_KEY**. Następnie sprawdzamy który klawisz został wciśnięty i jeżeli jest to strzałka w lewo, wówczas na obiekcie okna wywołujemy metodę

Listing 4. Metoda *windows_test()*, która realizuje właściwy proces tworzenia aplikacji graficznej

```
//Base buttons and windows demo
void tft_livedemo::windows_test() {
    using namespace gfx::gui;
    using namespace gfx::drv;
    m_disp.power_ctl( power_ctl_t::on );
    window win( rectangle( 10, 10, m_disp.get_width()-20, m_disp.get_height()-20),
            frame, window::flags::border |window::flags::selectborder );
    button btn( rectangle(40, 20, 400 , 40), layout(), win );
    seekbar seek( rectangle(40, 75, 400 , 40), layout(), win );
    editbox edit( rectangle(40, 150, 400 , 30), layout(), win );
    choice_menu choice1( rectangle(40, 220, 400, 250), layout(), win, choice_menu::style::normal );
    edit.readonly(true);
    m_edit = &edit;
    seek.value( 50 );
    btn.caption(„Button”);
    edit.value(„Text edit value”);
    static constexpr choice_menu::item menu1[] = {
            { 1, „linia 1” },
            { 20, „linia 20” },
            choice_menu::end  //Termination
    };
    choice1.items( menu1 );

    //Connect windows callback to the main window
    win.connect(std::bind(&tft_livedemo::window_callback,this,std::placeholders::_1), event::evtype::EV_KEY);
    win.connect(std::bind(&tft_livedemo::window_callback,this,std::placeholders::_1),     event::evtype::EV_TOUCH);
    btn.connect(std::bind(&tft_livedemo::on_click,this,std::placeholders::_1),event::evtype::EV_CLICK);
    btn.pushkey( gfx::input::kbdcodes::enter);    choice1.connect(std::bind(&tft_livedemo::on_select_item,this,std::placeholders::_1),
            event::evtype::EV_CLICK);
    seek.connect(std::bind(&tft_livedemo::on_seek_change,this,std::placeholders::_1),event::evtype::EV_CLICK);
    frame.set_focus( &win );
    frame.execute();
}
```

Listing 5. Metoda *window_callback()* obsługująca wybór aktualnie aktywnego komponentu

```
//A window calllback for select item
bool tft_livedemo::window_callback(const gfx::gui::event &ev) {
    using namespace gfx::input;
    using namespace gfx::gui;
    bool ret{};
    auto win = static_cast<gfx::gui::window *>(ev.sender);
    if (ev.type==event::evtype::EV_KEY && ev.keyb.stat == gfx::input::detail::keyboard_tag::status::DOWN) {
        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);
            dbg_info(„Toggle edit mode %i”, m_edit_mode);
        }
    }
    //Handle touch screen by selection
    if(ev.type==event::evtype::EV_TOUCH && ev.touch.eventid==touchevents::press_down) {
        win->select({ev.touch.x, ev.touch.y});
    }
    return ret; //  Need redraw
}
```

select_next() co spowoduje wybranie kolejnego komponentu. Jeśli jest to strzałka w prawo to analogicznie wybieramy metodę **select_prev()** w wyniku czego zaznaczony będzie poprzedni komponent. Dodatkowo, jeśli znajdujemy się w polu edycyjnym, klawisz **ENTER** powoduje wejście lub wyjście z trybu edycji polegającej na naprzemiennym włączaniu oraz wyłączaniu trybu tylko do odczytu (metoda ***read_only()***).

Znacznie prostsza jest obsługa zdarzeń dotykowych. W tym przypadku sprawdzamy jedynie czy przyszło zdarzenie **EV_TOUCH** i wywołujemy metodę ***select()*** na rzecz obiektu okna przekazując miejsce dotknięcia ekranu. Metoda ta automatycznie uaktywni dany widżet, jeśli stwierdzi, że punkt dotknięcia znajduje się w obrębie danego komponentu.

Kolejnymi metodami obsługi zdarzeń są zdarzenia **EV_CLICK** podpięte do poszczególnych komponentów. Dla porządku przedstawimy jedynie metodę reakcji na zdarzenie wciśnięcia przycisku ***on_click()*** (**listing 6**). W wyniku wciśnięcia klawisza podsystem graficzny wywoła metodę **on_click()** przekazując referencję do opisu zdarzenia (*event_info*). W pierwszym kroku rzutujemy pole **ev.sender** będące wskaźnikiem do typu **object** na typ obiektu **button**, ponieważ mamy pewność, że wskaźnik do tego obiektu jest pożądanego typu. W kolejnym kroku za pomocą funkcji wypisującej na porcie szeregowym wyświetlamy identyfikator (adres) okna, oraz tytuł (tekst) znajdujący się na przycisku. Każda metoda obsługi zdarzenia powinna zwracać wartość **true**, jeśli dany obiekt został zmieniony i chcemy, aby został odrysowany.

Podczas pisania własnych procedur obsługi zdarzeń należy pamiętać, aby procedury obsługi były nieblokujące, ponieważ wykonywane są w kontekście wątku graficznego. Zablokowanie metody na elemencie synchronizacyjnym spowoduje zatrzymanie wątku graficznego i brak reakcji na interakcję z użytkownikiem.

Aby uruchomić wcześniejszy przykład należy podłączyć do komputera zestaw *stm32f469i-disco* za pomocą kabla mini USB oraz pobrać kod źródłowy z repozytorium wpisując polecenie:

```
git clone -b pub/ep0120 –recursive https://www.boff.pl/cgit/public/isixsamples
```

W kolejnym kroku należy skompilować program oraz zaprogramować płytkę ewaluacyjną tak powstałym kodem maszynowym, co możemy zrealizować wydając następujące polecenia:

```
waf configure --crystal-hz=8000000 --debug waf
waf
waf program
```

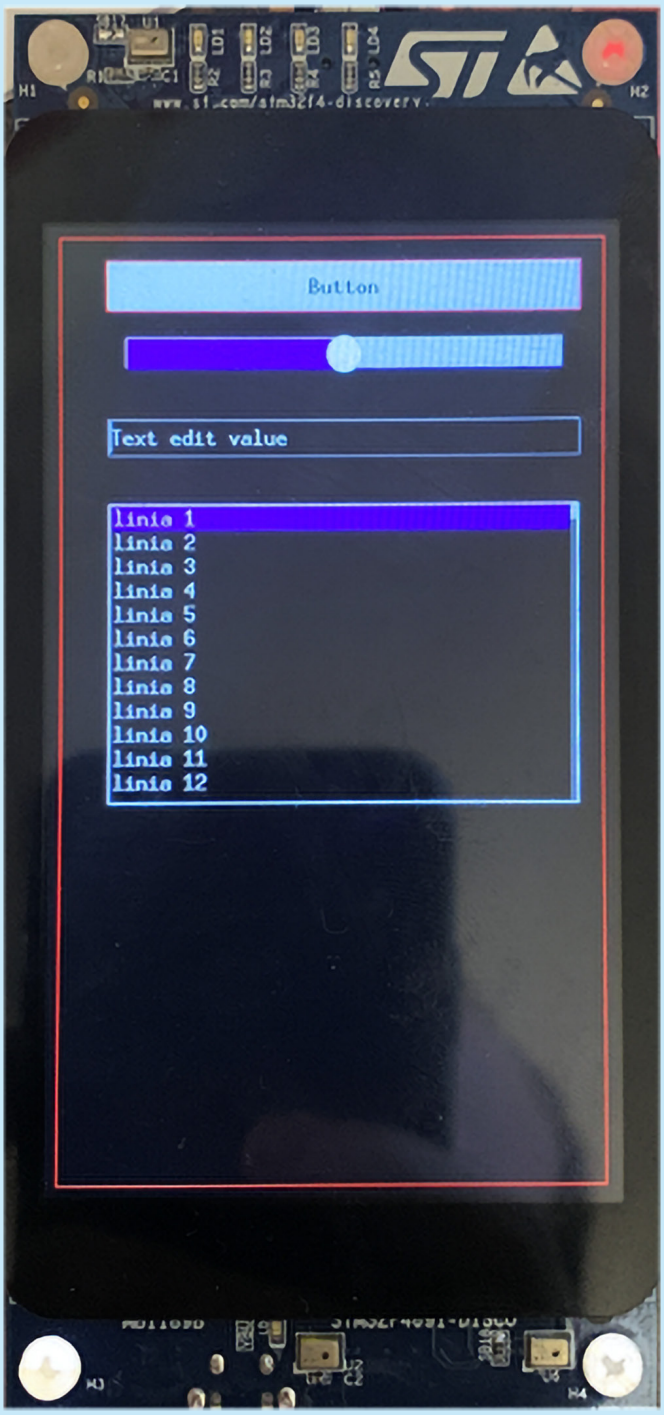
Równocześnie do wirtualnego portu szeregowego utworzonego przez programator należy podłączyć dowolny program terminalowy ustawiając następujące parametry transmisji: **115200,n,8,1.**

Po wykonaniu powyższych czynności na wyświetlaczu powinno pojawić się okno z demonstracyjnymi widżetami biblioteki graficznej. W przeciwieństwie do poprzedniego przykładu widżety powinny reagować na dotyk, a wciskanie odpowiednich elementów będzie powodować wywoływanie odpowiednich metod do nich przypisanych, co będziemy mogli zaobserwować na terminalu szeregowym.

Lucjan Bryndza, EP
lucjan.bryndza@boff.pl

Listing 6. Metoda reakcji na zdarzenie wciśnięcia przycisku *on_click()*

```
//Buttons callback
bool tft_livedemo::on_click( const gfx::gui::event &ev ) {
    auto btn = static_cast<gfx::gui::button*>(ev.sender);
    dbg_info(„Button %p clicked with desc %s”, ev.sender, btn->caption().c_str());
    return false;
}
```



Fotografia 1. Prezentacja możliwości biblioteki graficznej za pomocą prostego programu demonstracyjnego

Poprzednie części kursu i dodatkowe materiały dostępne są na stronie www.media.avt.pl