Obsługa wyświetlaczy tekstowych w systemie ISIX RTOS

Zamiennik dla popularnych wyświetlaczy HD44780

W elektronice konsumenckiej prym wiodą kolorowe wyświetlacze graficzne oraz interfejsy dotykowe. Jednak z drugiej strony, w urządzeniach niewymagających kolorowych interfejsów użytkownika w dalszym ciągu chętnie są używane wyświetlacze znakowe z kontrolerem HD44780. W czasach wszechobecnie panujących mikrokontrolerów ARM – zasilanych napięciem 3,3 V lub niższym – stosowanie wyświetlaczy z kntrolerem HD44780 staje się kłopotliwe z uwagi na konieczność zapewnienia dodatkowego napięcia zasilającego. Problematyczne jest również użycie magistrali równoległej, co często powoduje konieczność użycia układu w większej obudowie a zatem droższego. Poszukując innego rozwiązania dla prostych interfejsów użytkownika opartych o interfejs znakowy), znalazłem wyświetlacz LCD podobny wymiarami i zbliżony cenowo do klasycznych wyświetlaczy tekstowych, który doskonale może pełnić rolę zamiennika.

Wyświetlacz LCD-AG-C128032R-DIW W/KK E6 PBF może być zasilany napięciem 3,3 V, a dodatkowo można go sterować za pomocą wybranej magistrali szeregowej, co zmniejsza liczbę połączeń potrzebnych do sterowania wyświetlaczem. Jest to wyświetlacz graficzny o rozdzielczości 128×32 piksele, wyposażony w popularny kontroler UC1601. Wymiarami jest bardzo zbliżony do typowych wyświetlaczy znakowych o rozdzielczości 2×16 znaków. Kontroler umożliwia sterowanie wyświetlaczem za pomocą magistral szeregowych np. SPI czy I²C oraz równoległych np. Intel 8080.

Do sterowania wspomnianym wyświetlaczem powstała biblioteka zgodna z API dla wyświetlaczy tekstowych systemu ISIX-RTOS. Bibliotekę zrealizowano w oparciu o następujące założenia:

- Zgodność interfejsu z API strumieniowym dla wyświetlaczy tekstowych.
- Minimalne zużycie pamięci RAM (biblioteka nie używa dodatkowych buforów obrazu i operuje bezpośrednio na pamięci wyświetlacza).
- Rozszerzenia w stosunku do biblioteki podstawowej np. wyświetlanie ikon.
- Zgodność programowa z kontrolerem UC1601.
- Wyświetlanie dodatkowych komponentów np. ikon.
- Możliwość zmiany czcionki.
- Możliwość dalszej rozbudowy o dodatkowe typy wyświetlaczy i kontrolerów.

API wyświetlaczy tekstowych w systemie ISIX – biblioteka *libfoundation*

Obsługa wyświetlaczy znakowych w systemie ISIX jest realizowana przez część biblioteki *libfoundation*, którą zaprojektowano obiektowo na wzór biblioteki *iostream* stanowiącej część biblioteki standardowej C++. Dostęp do wyświetlacza obywa się za pomocą przeciążonego operatora **<<** sprawiając naturalne wrażenie wysyłania strumienia danych w kierunku wyświetlacza. Istotną zaletą tego podejścia (w przeciwieństwie do klasycznego interfejsu podobnego do *printf()*) jest to, że odpowiednie funkcje potrzebne do obsługi danego formatu, np. konwertowanie liczb na wartość tekstową, ustalane są na etapie kompilowania programu. W wypadku klasycznego interfejsu *printf()* wymagane funkcje konwertujące nie są znane w momencie kompilowania kodu programu, a dopiero ustalane w momencie jego wykonania poprzez wyszukiwanie tagów formatujących %. Użycie tego typu interfejsu powoduje istotny wzrost wielkości kodu wynikowego programu wynikający z konieczności linkowania wszystkich funkcji formatujących (np. całej biblioteki zmiennoprzecinkowej), ponieważ na etapie kompilowania nie są znane tagi formatujące, które mogą być użyte. W wypadku zastosowania interfejsu strumieniowego C++ jedynie faktycznie używane funkcje konwertujące będą wybrane przez kompilator i umieszczone w kodzie programu już podczas kompilowania. Na przykład, jeśli będziemy wyświetlali tylko łańcuchy tekstowe – bez używania liczb – kod odpowiedzialny za konwersję liczb na tekst nie będzie umieszczony w programie.

Z uwagi na to, że wyświetlacze znakowe najczęściej używane będą z mikrokontrolerami mającymi raczej skromne zasoby, stanowi to istotną przewagę interfejsu strumieniowego opartego na przeciążonym operatorze << nad rozwiązaniami klasycznymi znanymi z języka C. Inną istotną zaletą jest większe bezpieczeństwo podczas

korzystania ze strumieni, ponieważ eliminujemy możliwość popełnienia pomyłki niewykrywalnej na etapie kompilowania. Kompilator stricte przestrzega i sprawdza zgodność typów podczas wyboru odpowiedniego operatora przeciążonego, co nie jest możliwe do zrealizowania w wypadku, gdy mamy do czynienia z funkcją o dowolnej i nieustalonej liczbie parametrów, taką jak *printf*. Nietrudno np. wyobrazić sobie, co się stanie, gdy używając *printf* wpiszemy *%s*, a zamiast łańcucha tekstowego podamy wartość typu *int*. Wtedy kompilator nie jest nas w stanie przed niczym ochronić, gdyż używając operatora … rozluźniamy reguły sprawdzania do granic możliwości.

Na **rysunku 1** przestawiono hierarchię klas biblioteki obsługi wyświetlaczy tekstowych systemu ISIX. Obsługę wyświetlaczy tekstowych zawarto w bibliotece *libfoundation* stanowiącej integralną część systemu ISIX. Zawiera ona jedynie algorytmy obsługi wyświetlaczy oraz formatowania tekstu, przez co jest całkowicie niezależna od sprzętu i systemu. Nic nie stoi na przeszkodzie, aby podsystem obsługi wyświetlaczy znakowych oraz bibliotekę *libfoundation* zastosować w innych projektach, niezależnie od systemu ISIX. Dostęp do warstwy sprzętowej jest realizowany poprzez oddzielne klasy dostępu do sprzętu *uc1601_bus* oraz *hd44780 display*. Nie stanowią one części biblioteki i muszą zostać zaimplementowane oddzielnie, dla danej platformy sprzętowej. Używanie biblioteki jest banalnie proste i sprowadza się do utworzenia obiektu klasy wyświetlacza oraz przesyłanie do niego danych do wyświetlenia za pomocą przeciążonego operatora <<. Opcjonalnie możemy użyć dodatkowych klas – tagów formatujących umożliwiających zmianę sposobu formatowania liczb. Używanie biblioteki jest bardzo łatwe. Najpierw należy utworzyć klasę magistrali, następnie utworzyć klasę wyświetlacza podając w konstruktorze referencję do obiektu magistrali oraz fizyczne rozmiary wyświetlacza.

```
dev::dispbus_i2c m_disp_bus; //Display bus
fnd::lcd::uc1601_display m_lcd {m_disp_bus,
128, 32};  //Disp device
```

Na początku należy również wybrać czcionkę, ponieważ wyświetlacz UC1601 nie ma wbudowanego generatora znaków i czcionka jest generowana w sposób programowy:

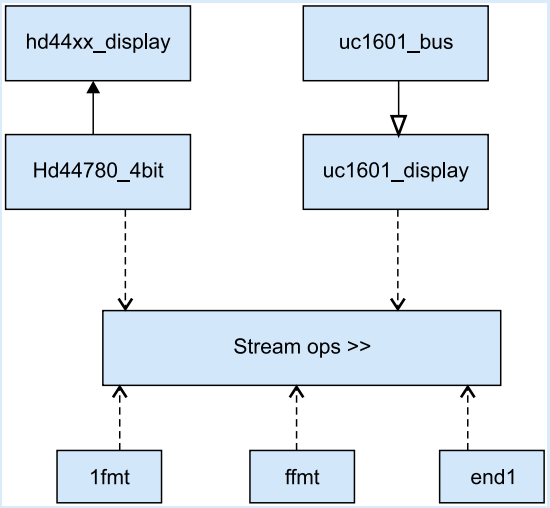
```
m_lcd.set_font( &res::font_default );
```

Mając obiekt utworzony w ten sposób możemy odwoływać się do niego wywołując operacje strumieniowe za pomocą przeciążonego operatora <<, na przykład:

```
m_lcd << pos(0,0) << „Value „ << 12 <<
endl();
```

W powyższym przykładzie najpierw do wyświetlacza LCD jest kierowany obiekt *pos*, którego zadaniem jest wysłanie do wyświetlacza rozkazu ustawienia kursora w pozycji x, y=(0,0). Następnie jest przesłany łańcuch tekstowy o treści „*value*”, a po nim liczba 12. Ostatnią czynnością jest przekazanie do wyświetlacza obiektu *endl()*, co spowoduje wyczyszczenie ekranu począwszy od bieżącej pozycji kursora do końca linii. W wyniku wykonania powyższych instrukcji, w pierwszej linii wyświetlacza zostanie wyświetlony komunikat *Value 12*.

Podobnie wygląda obsługa wyświetlacza znakowego *HD44780* z tą jedyną różnicą, że implementację dostępu do magistrali zrealizowano poprzez dziedziczenie obiektów, zamiast korzystania z dodatkowej klasy pomocniczej, więc w tym wypadku wystarczy stworzyć obiekt klasy hd44780, bez konieczności podawania referencji do obiektu klasy magistrali. Nie trzeba również podawać wielkości wyświetlacza, ponieważ kontroler HD44780 tego nie wymaga.



Rysunek 1. Hierarchia klas biblioteki wyświetlaczy tekstowych systemu ISIX

Oprócz danych wraz ze strumieniem do wyświetlenia – jak zauważyliśmy w przykładzie – są przesyłane tagi formatujące, czyli obiekty klas, które potrafią zmienić sposób formatowania lub wykonać czynności dodatkowe. Obecnie biblioteka ma zaimplementowane następujące tagi formatujące:

- Klasa ***lfmt*** – umożliwiająca dodatkowe formatowanie liczby stałoprzecinkowej. Konstruktor klasy zdefiniowano w następujący sposób: lfmt(unsigned val_, int fmt_, char fmtch_='0', short base_=dec). Jako pierwszy argument jest podawana formatowana liczba, jako drugi argument przyjmowana jest liczba znaków wiodących. Następne argumenty są opcjonalne, kolejno, *fmtch_* określa znak formatujący, a zmienna *base_* określa format wyświetlania jako dziesiętny (dec) lub szesnastkowy (hex). Zatem możemy albo za pomocą operatora << wyświetlić bezpośrednio liczbę stałoprzecinkową bez formatowania, albo opakować ją w dodatkową klasę formatująca, która zmieni jej sposób wyświetlania.
- Klasa ***ffmt*** służy do formatowania liczb zmiennoprzecinkowych i pełni podobną rolę jak *lfmt*. Jej konstruktor został zdefiniowany następująco: ffmt(T _val, int _prec). Pierwszym argumentem jest liczba zmiennoprzecinkowa do sformatowania, natomiast drugim liczba zer wiodących, która będzie wyświetlona.
- Klasa ***pos*** służy do ustawienia kursora wyświetlacza na zadanej pozycji, a jej konstruktor zdefiniowano następująco: pos(int x_, int y_). Argumentami jest pozycja kursora na wyświetlaczu. Te współrzędne zależą od klasy implementującej dany wyświetlacz. W wypadku wyświetlacza HD44780 będą to współrzędne znakowe, np. znak 1 linia 1, jest równoważne wywołaniu *pos(1, 1)*, natomiast w wypadku wyświetlacza graficznego będą to współrzędne graficzne wyrażone w pikselach z uwagi na to, że jest używana czcionka o zmiennej szerokości znaków.
- Klasa ***endl*** jest tagiem formatującym służącym do poinformowania wyświetlacza o chęci wyczyszczenia

Listing 1. Struktura *icon_t*
```
struct icon_t
{
  unsigned char pg_width;
  unsigned char height;
  const unsigned char* data;
};
```

obrazu od aktualnej pozycji kursora do końca linii; jej konstruktor nie przyjmuje żadnych argumentów.

W bibliotece, oprócz operatorów dla klas formatujących, wbudowano następujący zestaw przeciążonych operatorów **<<** dla typów prostych:

```
template <typename D>
  D& operator<<( D &o,const char *str)
template <typename D>
  D& operator<<(D &o,unsigned value)
template <typename D>
  D& operator<<(D &o,int value)
template <typename D>
  D& operator<<(D &o,double value)
```

Dzięki ich zastosowaniu mamy możliwość wyświetlania na wyświetlaczu wartości typów *POD (Plain Old Data)*. Użytkownik wzorując się na powyższych operatorach może napisać własne wersje operatorów dla innych typów, dzięki czemu możemy np. bezpośrednio przesyłać do wyświetlacza obiekty własnych typów, tak jakbyśmy mieli do czynienia z typami wbudowanymi, np. *m_disp << moj_obiekt << endl()*. Podstawowe wersje operatorów << zdefiniowano w pliku *display_operators.hpp*. Przykład implementacji operatora dla typu *int* wygląda nastepująco:

```
template <typename D>
  D& operator<<(D &o,int value)
  {
    char buf[12];
    fnd::fnd_itoa(buf,value,1,'0');
    o << buf;
    return o;
  }
```

Jako pierwszy argument jest przyjmowana referencja do obiektu wyświetlacza, natomiast jako kolejny wartość do wyświetlenia – w tym wypadku typu *int*. Następnie, w implementacji operatora jest wywoływana funkcja *fnd_itoa*, której zadaniem jest konwersja liczby stałoprzecinkowej na łańcuch tekstowy. Po skonwertowaniu na wartość tekstową jest wywoływany operator << (const char*), który realizuje wyświetlenie łańcucha tekstowego.

Bazując na kodzie operatorów dla typów POD w prosty sposób możemy utworzyć rodzinę własnych operatorów, które będą pokazywały inne typy danych użytkownika bezpośrednio na wyświetlaczu za pomocą prostego przypisania.

Oprócz obsługi trybu tekstowego, obie klasy tj. *hd44780* jak i *uc1601*, umożliwiają różne niestandardowe operacje (np. wyświetlanie ikon lub ustawienie czcionki), które mogą być zrealizowane bezpośrednio poprzez odwołanie się do odpowiedniej metody obiektu wyświetlacza. Za najistotniejsze metody klasy *uc1601_display* możemy uznać:

- ***int clear();***. Metoda odpowiedzialna za kasowanie wyświetlacza. Nie przyjmuje żadnych argumentów i zwraca kod błędu,
- ***int error() const***. Funkcja zwracająca kod błędu ostatniej operacji wykonanej na wyświetlaczu.
- ***int box(int x1, int y1, int cx, int cy, box_t type = box_t::clear);***. Funkcja rysuje prostokąt o wymiarach zaczynających się punkcie *x1, y1* o wielkości określonej przez zmienne *cx, cy*. Opcjonalny argument *type* określa rodzaj wypełnienia, gdzie: *box_t::clear* oznacza kolor tła, *box_t::fill* oznacza kolor wypełnienia, *box_t::frame* spowoduje pustej ramki z obrysem.
- ***int progress_bar(int x1, int y1, int cx, int cy, int value, int max = 100);***. Funkcja wyświetla pasek postępu w formie prostokąta. Jako argumenty *x1, y1* przyjmuje punkt początkowy, *cx, cy* – wielkość paska, *value* – aktualna wartość (przekłada się na długość „zaczernionego" paska), opcjonalny *max* – maksymalna wartość zmiennej *value*. Funkcja zwraca kod błędu.
- ***void set_font(const font_t * font);***. Funkcja przypisuje bieżącą czcionkę do obiektu wyświetlacza. Od momentu przypisania czcionki wszystkie teksty będą wyświetlane za jej pomocą.
- ***int show_icon(int x1, int y1, const icon_t *icon);***. Funkcja rysuje ikonę na pozycji *x1, y1*. Ikona/obrazek jest przekazywana jako argument *icon*. Funkcja zwraca kod błędu wykonanej operacji.

Klasę *uc1601_display* napisano w taki sposób, aby ograniczyć użycie pamięci RAM. Operuje bezpośrednio na pamięci danych wyświetlacza. W związku z tym ma ona pewne ograniczenia – wszystkie współrzędne osi pionowej *y* lub *cy* muszą być podzielne przez 8. Zatem w osi rzędnych mamy możliwość pozycjonowania napisów i komponentów z dokładnością do 8 pikseli.

Ikona jest reprezentowana poprzez strukturę *icon_t* pokazaną na **listingu 1**.

Pole *pg_width* zawiera wielkość obrazu wyrażoną w 8-pikselowych kolumnach. Co oczywiste, dla wielkości 32 będzie to wartość 4. Pole *height* zawiera szerokość obrazu w pikselach, natomiast pole *data* jest wskaźnikiem do bitmapy w reprezentacji zgodnej z organizacją pamięci wyświetlacza.

Kontroler UC1601 nie posiada wbudowanego generatora znaków z tego powodu litery muszą być przygotowane w sposób programowy. Wada ta jest również zaletą, ponieważ, dzięki temu istnieje możliwość przygotowania ładnie wyglądającej czcionki o zmiennej szerokości znaków. Struktura definiująca czcionke zdefiniowana jest następująco:

Pole *height* zawiera wysokość czcionki. Pole *first_char* zawiera pierwszy kod znaku występującego w czcionce. Wewnętrzna struktura chr_desc_t zawiera wskaźnik do tablicy opisującej kod znaku i indeks mapy bitowej gdzie umieszczony jest dany znak. Natomiast pole bmp zawiera samą mapę bitową definującą znaki.

Przygotowanie odpowiedniej bitmapy jest kłopotliwe dlatego najlepiej do tego celu wykorzystać będzie gotowy program. Doskonale do tego celu nadaje się darmowy „The Dot Factory", który na podstawie czcionek syste-

Listing 2. Struktura definiująca czcionkę
```
struct font_t
{
  unsigned char height;
  char first_char;
  char last_char;
  unsigned char spc_width;
  struct chr_desc_t
  {
    unsigned char  width;
    unsigned short offset;
  } const *chr_desc;
  const unsigned char *bmp;
};
```

mowych, oraz plików BMP potrafi wygenerować odpowiednie struktury oraz dane, zawierające opis czcionek oraz bitmap. Program jest bardzo elastyczny i umożliwia wygenerowanie map bitowych w dowolnym układzie w zależności od organizacji pamięci wyświetlacza Zrzut ekranu okna konfiguracji odpowiednią dla naszego wyświetlacza typu UC1601 przedstawiono na rysunku (n).

Przykład praktyczny. Wyświetlacz LCD-AG-C128032R-DIW W/KK E6 oraz mikrokontroler STM32F100R6T6B

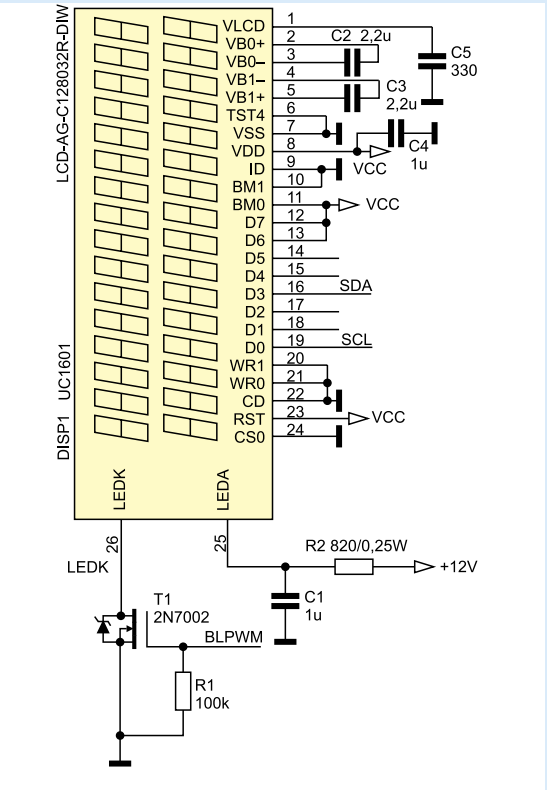
Po zapoznaniu się z interfejsem biblioteki zaprezentujemy w jaki sposób podłaczyć wyświetlacz LCD-AG-C128032R-DIW W/KK E6 do mikrokontrolera STM32F100R6T6B, tak aby sprawdzić jej działanie w praktyce.

Wyświetlacz został wykonany w technologi **COG**, dzięki czemu charakteryzuje się niską ceną, oraz ma rozdzielczość **128×32** piksele. Wymiarami przypomina klasyczny wyświetlacz znakowy 2×16. Jego najistotniejszą zaletą jest zasilanie napięciem 3,3 V co odpowiada standardowi zasilania współczesnych mikrokontrolerów, zatem odpada problem z dopasowywaniem poziomów logicznych, oraz dostarczeniem odrębnego napięcia do zasilania wyświetlacza.

Sposób dołączenia wyświetlacza do mikrokontrolera

Wyświetlacze z kontrolerem UC1601, mogą być dołączone do systemu zarówno za pomocą magistrali równoległej jak i magistral szeregowych takich jak **I²C** czy **SPI**. Z uwagi na stosunkowo niewielką ilość przesyłanych danych pod rozwagę bierzemy jedynie interfejsy szeregowe. W ostatecznym rozważaniu zdecydowano się na sterowanie za pomocą magistrali **I²C** z uwagi na najmniejszą liczbę wyprowadzeń. Co prawda w stanardowym wykonaniu magistrala pracuje z jedynie częstotliwością 100 kHz, a w wersji przyspieszonej 400 kHz, jednak producent kontrolera **uc1601** deklaruje że wyświetlacz potrafi obsłużyć magistralę I²C dla prędkości ponad 3 MHz. Biorąc pod uwagę wymaganą do przesłania ilość danych prędkość ta jest zadowalająca. Wyświetlacz do prawidłowego działania potrzebuje jedynie połączenia mikrokontrolera za pomocą dwóch przewodów **SDA** i **SCL**. Sposób dołaczenia wyświetlacza do mikrokontrolera STM32F100 przedstawiono na **rysunku 2**.

Z uwagi na możliwość pracy równoległej, oraz możliwość konfiguracji różnych trybów pracy wyświetlacz posiada stosunkowo dużo wyprowadzeń. Niemniej jednak mają one stosunkowo duży rozstaw i większość połączeń wykonana jest w pobliżu wyświetlacza. Po podłączeniu wszelkich linii konfiguracyjnych do odpowiednich linii zasilania wyświetlacz sterowany jest jedynie za pośrednictwem dwóch linii **SDA**, **SCL**. Do prawidłowego działania konieczne jest dołączenie zestawu kondensatorów **C1-C4**, które potrzebne są dla pompy ładunkowej przetwornicy wewnętrznej. Oprócz kondensatorów konieczne jest dołączenie niektórych portów wyświetlacza do odpowiednich linii zasilania, tak aby skonfigurować wyświetlacz do pracy z wybraną magistralą. Aby kontroler **UC1601** pracował w trybie magistrali I²C konieczne jest wykonanie następującej



Rysunek 2. Dołączenie wyświetlacza za pomocą I²C

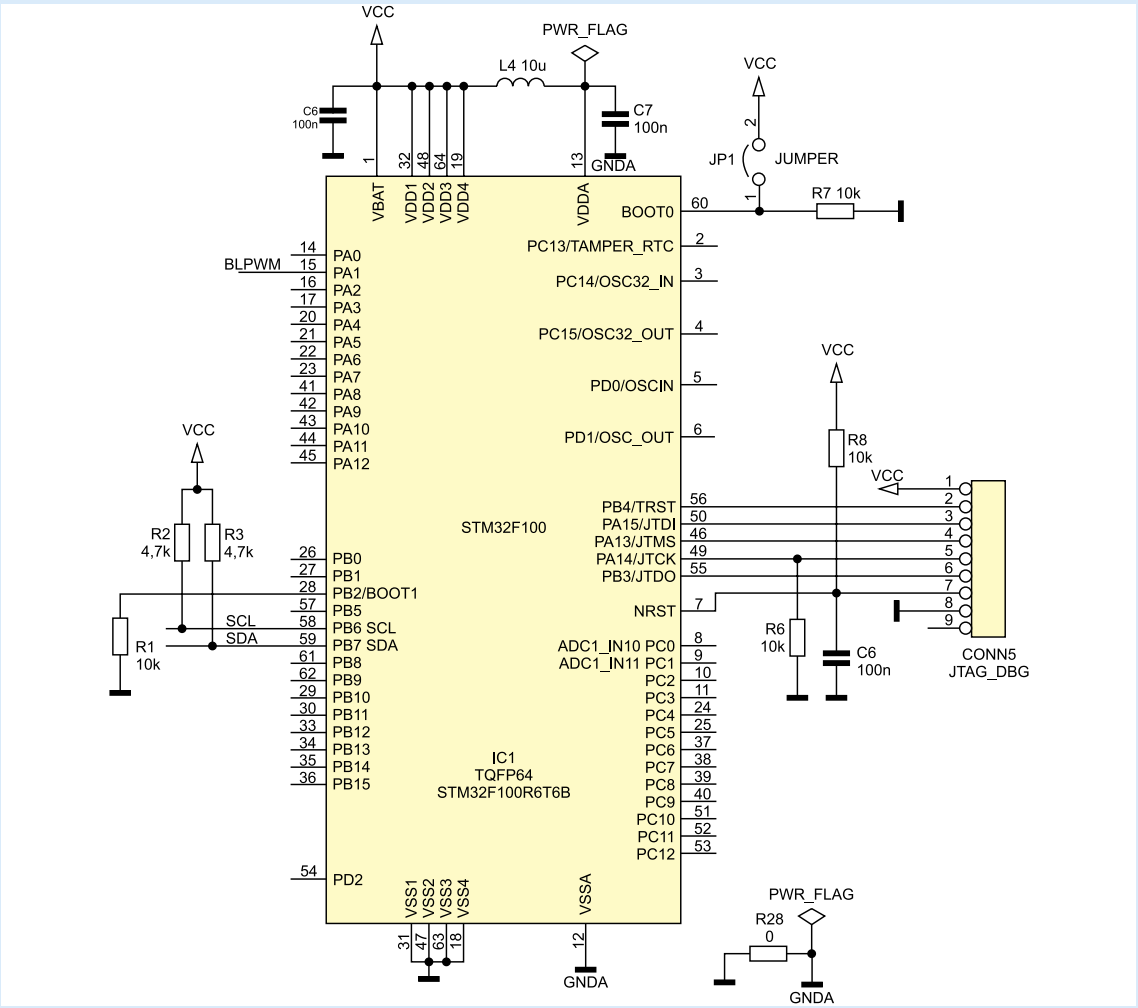
konfiguracji wyprowadzeń wyświetlacza: **DB[7:6]=11b CD=0 WR[1:0]=00b :CS[1:0]=00b BM[1:0]=10b**. Linie danych **SDA** i **SCL** stanowią odpowiednio wyprowadzenia **DB3** oraz **DB0** wyświetlacza, które dołączone są do portów **PB7** i **PB6** mikrokontrolera. Porty PB6, PB7 stanowią wyprowadzenia sprzętowego kontrolera magistrali I2C1. Do linii SDA oraz SCL podłączono również rezystory podciągające je do dodatniej szyny zasilania zgodnie ze specyfikacją magistrali. Z uwagi na to że wyświetlacz jest typu **FSTN** do prawidłowej pracy konieczne jest dołączenie podświetlania. Podświetlanie wyświetlacza stanowi biała dioda LED, która do prawidłowego działania wymaga zasilenia jej prądem o natężeniu około 10-15 mA. W przedstawionym układzie zasilenie podświetlania wykonywane jest bezpośrednio z napięcia zasilającego układu +12 V oraz resystor R1. Dodatkowy tranzystor T1 umożliwia odłączenie wyświetlacza, w przypadku programowej obsługi wyłączania urządzenia, lub umożliwia sterownie intensywnością podświetlania za pomocą sygnału PWM. Zasilanie kontrolera wyswietlacza o wartości 3.3 V jest wspólne dla całego systemu mikroprocesorowego. Kontroler wyświetlacza zadowala się prądem o natężeniu **~300 µA**, zatem jeśli planujemy wykorzystanie wyświetlacza w układach energooszczędnych możemy użyć wersji **PSTN**, i zrezygnować z podświetlania. Schemat części procesorowej z mikrokontrolerem STM32F100 stanowi klasyczną notę aplikacyjną układu, wraz z wyprowadzonym interfejsem JTAG do wygodnego programowania i debugowania pokazano na rysunku 3. Zaprezentowany przykład po drobnych modyfikacjach będziemy mogli również uruchomić bez problemu na innych mikrokontrolerach rodziny STM32, jedynym wymaganym urzadzeniem peryferyjnym jest sprzętowy sterownik I2C1, w który wyposażony jest chyba prawie każdy układ rodziny stm32.

Przykład praktyczny

Na bazie powyższej platformy sprzętowej przygotowano prosty przykład demonstrujący możliwości biblioteki oraz wyświetlacza. Kod został napisany w języku C++ (kompilator gcc) i jest zgodny z zatwierdzonym nowym standardem języka **ISO/IEC 14882:2011** Główny kod programu realizowany jest przez klasę **application**. Obiekt **app** klasy **application** tworzony jest jako statyczny bezpośrednio w funkcji main, a następnie wywoływana jest metoda **process()** realizująca główną część programu. Kod klasy demonstrującej możliwości biblioteki przedstawiono na listingu (n)

```
namespace app {

        class application {
                static constexpr auto
BLPORT = GPIOA;
                static constexpr auto
BLPWM = 1;
        public:
            //Constructor
            application() {
                //Enable backlight pin
                stm32::gpio_abstract_
config( BLPORT, BLPWM, stm32::AGPIO_MODE_
OUTPUT_PP, stm32::AGPIO_SPEED_HALF );
                stm32::gpio_set( BLPORT,
BLPWM );
                m_lcd.set_font(
&res::font_default );
                }
                //The main application processs
                void process() {
                    //Namespace used in method
                    using namespace fnd::lcd;
                    using namespace dev;
                    constexpr auto C_about_
timeout = 5000;
                    //Show icon
                    m_lcd.show_icon(0, 0,
&res::boff_logo_img);                    //
[2]
                    tim::timer_wait( C_about_
timeout  );
                    m_lcd.clear();
                    //Display integer
                    m_lcd << pos(0,0) << „Int
„ << tim::read_tick() << endl();      //[3]
                    //Display float
                    static constexpr auto
float_val  = 1.23f;
                    m_lcd << pos(0,16) <<
„Float val „ << float_val << endl() ;  //[4]
                    tim::timer_wait( C_about_
timeout  );
                    //Display progress bar
using direct interface
                    //and update it every 50ms
period per 10%
```



Rysunek 3.

```
                m_lcd.clear();

                    static constexpr auto
progress_x = 18;
                    static constexpr auto
progress_cx = 128 - progress_x - 2;
                    static constexpr auto
progress_y = 16;
                    static constexpr auto
progress_cy = 16;
                static constexpr auto
percent_tout = 50;
//[5]
                for(auto percent=0;
percent<=100; ++percent) {
                    m_lcd.progress_bar(
progress_x, progress_y, progress_cx,
progress_cy, percent );
                    tim::timer_wait(
percent_tout  );
                }
            }
        private:
            dev::dispbus_i2c m_disp_bus;
//Display bus
            fnd::lcd::uc1601_display m_
lcd {m_disp_bus, 128, 32};        //Disp
device
        };
}
```

Klasa **application** zawiera jako składowe prywatne dwie klasy: Klasę magistrali **dev::dispbus_i2c m_disp_bus** dziedziczącą po klasie interfejsu **uc1601_bus** która jest odpowiedzialna za fizyczną komunikację z wyświetlaczem za pomocą magistrali I²C, oraz **uc1601 display** która realizuje algorytmy ogólne związane z obsługą kontrolera. Zmiana interfejsu sterowania wyświetlaczem możliwa jest poprzez zdefiniowanie innej klasy kontrolera np. komunikującego się z wyświetlaczem za pomocą magistrali SPI bez konieczności zmiany i rekompilacji kodu klasy **uc1601_display**. Obiekt klasy wyswietlacza **m_lcd** w konstruktorze przyjmuje referencję do obiektu magistrali **m_disp_bus**, oraz dwie dodatkowe zmienne całkowitoliczbowe, które reprezentują fizyczne rozmiary wyświetlacza wyrażone w pikselach. W konstruktorze klasy **application::application()** konfigurowana jest linia podświetlania w kierunku wyjścia, a następnie jest ustawiana w stan wysoki, co powoduje załączenie podświetlania. Następnie za pomocą metody **set_font** obiektu **m_lcd** ustawiana jest czcionka podstawowa używana w trybie znakowym. Główny kod programu realizowany jest przez metodę **process()** wywołaną z funkcji **main().** Na początku w celu skrócenia zapisu definiowane są wykorzystywane w metodzie przestrzenie nazw. Następnie w **[2]** wyświetlane jest przykładowe logo, po czym po około 5 sekundach w **[3]** za pomocą wspomnianych wcześniej operatorów **<<** reprezentujących strumienie wyświetlany jest łańcuch tekstowy przedstawiający liczbe milisekund jaka mineła od włącznia zasilania urządzenia. Liczba ta jest typu unsigned int , a więc zostanie wywołany przeciążony operator dla liczb stałoprzecinkowych bez znaku. Nastepnie w drugiej linii (**[4]**) celem przetestowania działania operatora przeciążonego dla liczb zmiennoprzecinkowych wykonane jest wyświetlenie przykładowej zmiennej typu **float**. Po odczekaniu ponownie około 5 sekund **[5]** wywoływany jest fragment demonstrujący działanie paska postępu. Jest to prosta pętla for, która zwiększa wartość zmennej w zakresie 0...100. Wewnątrz pętli za pomocą metody klasy **uc1601_display** w dolnej linii wyświetlany jest pasek postępu, dodawane jest krótkie opóźnienie (50ms), tak byśmy mogli zauważyć na ekranie zwiększanie się paska postępu.

Po wypełnieniu paska postępu do 100% program kończy działanie.

Klasa obsługi dostepu do wyświetlacza za pomocą magistrali I²C **dispbus_i2c** została zrealizowana za pomocą sprzętowego kontrolera **I2C1** który został zmuszony do pracy z prędkoścla około 2 MHz pomimo iż producent deklaruje pracę jedynie w trybie fast (400 kHz) W klasie wykorzystano wszystkie walory sprzętowego kontrolera włącznie z transmisją DMA oraz wykorzystaniem systemu przerwań. Szczegóły implementacyjne wykraczają poza łamy niniejszego artykułu a zainteresowani mogą prześledzić jej kod w dołączonym przykładzie.

Implementując własną klasę obsługi magistrali np. SPI należy odziedziczyć interejs z klasy bazowej **uc1601bus** oraz zaimplementować we własnym zakresie nastepujące metody wirtualne:

```
virtual int data_wr( const uint8_t *buffer,
size_t len ) = 0;
```

Metoda zapisuje dane przekazane jako parametr **buffer** o rozmiarze **len** do pamięci danych wyświetlacza. Zwracając 0 w przypadku powodzenia lub inną wartość w przypadku błędu

```
virtual int data_rd(  uint8_t *buffer,
size_t len ) = 0;
```

Metoda odczytuje dane z pamięci wyświetlacza o rozmiarze określonym jako parametr **len** i zapisuje je pod adresem przekazanym w argumencie **buffer.** Zwraca 0 w przypadku powodzenia lub inną wartość w przypadku błędu.

```
virtual void mdelay( unsigned timeout ) = 0;
```

Funkcja opóźnienia, która odczekuje zadaną liczbę milisekund określoną przez argument **timeout**

```
virtual int command_( int cmd1, int cmd2 )
= 0;
```

Funkcja przesyła komedę do wyświetlacza przekazaną jako argument **cmd1** oraz opcjonalnie w przypadku komendy dwubajtowej drugi bajt komendy z argumentu **cmd2**. W przypadku komendy 1 bajtowej parametr **cmd2** przyjmuje wartość **CMD_EMPTY.**

Zakończenie

Opisana biblioteka zapewnia dużą elastyczność konfiguracji, oraz możliwość dostosowania do obsługi dowolnego wyświetlacza znakowego. Dzięki mechanizmowi przeciążania operatorów funkcje konwertujące wykrywane są już na etapie kompilacji, a nie w czasie wykonania programu. Pozwala to zaoszczędzić dużą ilość pamięci FLASH i RAM w porównaniu do klasycznej implementacji a-la **printf().** Pomimo iż biblioteka wydaje się stosunkowo skomplikowana w praktyce jej użycie jest bardzo proste i sprowadza się napisania kilkunastu linii kodu. Wspomniane wcześniej mechanizmy dedukcji typów i wywołania przeciążonych operatorów przekładając się na bardzo efektywny kod maszynowy. Pomimo iż biblioteka formalnie stanowi część systemu ISIXRTOS, nie zawiera ona odwołań do funkcji systemowych **ISIXA**, dzięki czemu może być wykorzystana niezależnie.

Lucjan Bryndza, EP
SQ5FGB