# 32 bity jak najprościej (4)

## STM32F0 – nieblokująca obsługa wyświetlacza LCD ze sterownikiem rodziny HD44780

**W kolejnym odcinku serii przedstawimy całkiem nietypową realizację bardzo typowego składnika oprogramowania systemu mikroprocesorowego – obsługi wyświetlacza ciekłokrystalicznego ze sterownikiem rodziny HD44780. Taki sposób obsługi wyświetlacza może być zastosowany dla dowolnego mikrokontrolera, również 8-bitowego.**

Przykład opisywany w artykule korzysta oczywiście z STM32F0, a projekt został uruchomiony na płytce z najmniejszym dostępnym aktualnie mikrokontrolerem tej serii – STM32F030F4, opisanej w poprzedniej części artykułu. Po zmianie definicji portów i operacji na nich, umieszczonych w pliku f030exp1.h, oprogramowanie można uruchomić np. na płytce serii Discovery wyposażonej w dowolny procesor rodziny STM32F0.

Głównym problemem, jaki napotyka programista chcąc efektywnie zaprojektować obsługę wyświetlacza LCD ze sterownikiem klasy HD44780 jest brak mechanizmu asynchronicznego informowania o gotowości sterownika przy użyciu przerwania. Zwykle wyświetlacz taki jest obsługiwany programowo przy użyciu aktywnego oczekiwania, co może powodować niepożądane opóźnienia w pracy oprogramowania i uniemożliwia modyfikowanie zawartości wyświetlacza przez procedury obsługi przerwań. W rozwiązaniach programowych niekorzystających z systemów operacyjnych czasu rzeczywistego wyświetlacz ze sterownikiem zgodnym z HD44780 obsługuje się zazwyczaj w pętli zdarzeń. W przypadku zastosowania RTOS obsługę wyświetlacza realizuje się w jednym zadaniu o niskim priorytecie. W obu rozwiązaniach obsługa wyświetlacza niepotrzebnie angażuje moc obliczeniową procesora. Aktywne oczekiwanie na gotowość wyświetlacza uniemożliwia realizację oprogramowania w postaci pozbawionej pętli zdarzeń, co utrudnia redukcję poboru mocy przez urządzenie poprzez usypianie procesora. Podobnie zastosowanie interfejsu jednokierunkowego i rezygnacja z testowania gotowości w typowych spotykanych realizacjach programowych skutkuje wydłużonym i niepotrzebnym aktywnym oczekiwaniem na upłynięcie czasu gwarantującego wykonanie polecenia przez sterownik wyświetlacza.

Proponowane rozwiązanie programowe umożliwia obsługę wyświetlacza bez programowego oczekiwania, dzięki czemu zawartość wyświetlacza może być asynchronicznie modyfikowana przez procedury obsługi przerwań. Minimalizuje ono również narzut czasu procesora na obsługę wyświetlacza, co skutkuje wzrostem wydajności obliczeniowej systemu i szybszą reakcją na zdarzenia. Zastosowanie przedstawionego rozwiązania eliminuje pętle oczekiwań i umożliwia zaprojektowanie oprogramowania w strukturze bez pętli zdarzeń lub z pustą pętlą zdarzeń. W ten sposób wyeliminowane zostają praktycznie wszystkie problemy związane z oprogramowaniem wyświetlacza występujące w typowych projektach oprogramowania korzystających z alfanumerycznych wyświetlaczy LCD.

### Zasady współpracy ze sterownikiem klasy HD44780

Zasady podłączania i programowania sterownika opisane są w dokumencie „Hitachi HD44780U (LCD-II)(Dot Matrix Liquid Crystal Display Controller/Driver)", dostępnym w Internecie. Oprogramowanie obsługi wyświetlacza składa się zwykle z kilku składników – procedur, w tym procedury inicjującej wyświetlacz, procedur zapisu poleceń oraz procedur zapisu wyświetlanych znaków.

Procedura inicjująca wymaga przesłania do wyświetlacza kilku poleceń w określonych minimalnych odstępach czasowych. Po zainicjowaniu zazwyczaj korzysta się tylko z dwóch funkcji sterownika – ustawienia pozycji kursora i zapisu danych.

Każde z poleceń wykonywanych przez sterownik ma wyspecyfikowany przez producenta maksymalny czas wykonania. Oprogramowanie musi zapewnić, że nowe polecenie nie zostanie przesłane przed zakończeniem wykonywania poprzedniego. Można to osiągnąć na dwa sposoby: przez odczekanie czasu nie krótszego od gwarantowanego czasu wykonania polecenia lub przez testowanie gotowości sterownika na przyjęcie kolejnego polecenia poprzez odczyt rejestru stanu.

Odczyt stanu wymaga sterowania przez mikrokontroler linii R/–W oraz zmiany kierunku portu danych, co jest niekiedy dość niewygodne, dlatego w wielu projektach interfejs sterownika LCD jest jednokierunkowy, a synchronizacja jest osiągana przez odczekiwanie czasu niezbędnego do wykonania polecenia przez sterownik.

Dokładna lektura dokumentacji sterownika odsłania kilka nieoczywistych jego cech, np. inkrementacja pozycji zapisywanego znaku jest wykonywana przez sterownik już po zgłoszeniu gotowości sterownika i może zająć do 4 µs, przez który to czas sterownik nie może jeszcze przyjąć kolejnego znaku, pomimo

sygnalizacji stanu gotowości. Wpływa to na budowę fragmentów oprogramowania odpowiedzialnych za testowanie gotowości – muszą one zagwarantować dodatkowy odstęp czasu po stwierdzeniu gotowości sterownika na podstawie bitu BUSY w rejestrze stanu, co wydłuża czas ich wykonania oraz całkowity czas działania procedur programowych zajmujących się wyświetlaniem napisów. Takie zachowanie sterownika stanowi dodatkowy argument za rezygnacją z programowego testowania gotowości i korzystania z interfejsu dwukierunkowego.

Do mikrokontrolera zasilanego napięciem 3,3 V można dołączyć sterownik LCD zasilany napięciem 3,3 V lub 5 V – poziomy wejściowe sterownika przy zasilaniu 5 V są zgodne z poziomami wyjściowymi mikrokontrolera.

## Założenia dla projektu oprogramowania

Przedstawiony projekt oprogramowania ma na celu całkowitą eliminację oczekiwania na gotowość sterownika i umożliwienie aktualizacji zawartości wyświetlacza przez procedury obsługi przerwań, a w konsekwencji – realizację całego oprogramowania w postaci samych procedur obsługi przerwań, z pustą pętlą zdarzeń. Struktura oprogramowania powinna ograniczyć opóźnienia programowe do niezbędnego minimum, wynikającego z konieczności zachowania parametrów czasowych protokołu szyny w cyklach zapisu poleceń i danych do sterownika.

Przyjęto, że zarówno aktualizacja zawartości wyświetlacza jak i jego inicjowanie zostaną zrealizowane tą samą metodą – interakcja ze sterownikiem wyświetlacza będzie następować wyłącznie w procedurze obsługi przerwania timera.

W ten sposób zostanie wyeliminowane oczekiwanie programowe podczas inicjowania sterownika i zapisu danych. Rozwiązanie takie umożliwia również łatwe reinicjowanie sterownika w czasie normalnej pracy urządzenia, co ma istotne znaczenie dla niezawodności urządzenia – przy wahaniach napięcia zasilającego sterowniki LCD mają tendencję do zawieszania się lub samoczynnego reinicjowania w domyślnym trybie pracy (jednowierszowym), który na ogół nie jest zgodny z zastosowanym wyświetlaczem.

## Interfejs programowy obsługi wyświetlacza

Komunikacja pomiędzy oprogramowaniem określającym zawartość wyświetlacza i procedurami obsługi wyświetlacza zachodzi przy użyciu struktury danych umieszczonej w pamięci RAM mikrokontrolera, zawierającej zadaną przez oprogramowanie zawartość całego wyświetlacza oraz znaczniki poleceń zrealizowane jako zmienne logiczne.

Ponieważ podczas aktualizacji zawartości przed przesłaniem kodów znaków konieczne jest również ustawienie kursora na początku wiersza, w przypadku zastosowania wyświetlaczy wielowierszowych sensowne jest zdefiniowanie oddzielnych znaczników konieczności modyfikacji dla poszczególnych wierszy wyświetlacza. Znacznik jest ustawiany po wpisaniu przez oprogramowanie nowych danych dla danego wiersza wyświetlacza. Aktualizacja zawartości wyświetlacza w procedurze obsługi przerwania timera polega na skopiowaniu zawartości całego wiersza z bufora danych w pamięci do sterownika.

**Listing 1. Plik lcd.h**

```
#ifndef LCD_H_
#define LCD_H_

#include <stdint.h>

#define LCDROWS 2
#define LCDCOLS 16

struct lcdctrl_ {
    uint8_t screen[LCDROWS][LCDCOLS];  // screen content
    struct {
        volatile _Bool init, upd[LCDROWS];
    } req;  // request flags
};

extern struct lcdctrl_ lcd;
void lcdhandler(void);
#endif
```

W celu zainicjowania sterownika wyświetlacza oprogramowanie ustawia znacznik żądania inicjowania. Powoduje to wyzwolenie sekwencji akcji wykonywanych w procedurze obsługi przerwania timera.

W najprostszym przypadku dla wyświetlacza dwuwierszowego potrzebne są trzy polecenia: inicjowania oraz aktualizacji pierwszego i drugiego wiersza. Oprogramowanie użytkowe wpisuje nową zawartość wyświetlacza do bufora, a następnie ustawia znaczniki poleceń. Zawartość struktury danych jest zadeklarowana w pliku lcd.h (**listing 1**). Składają się na nią:

- bufor ekranu w postaci tablicy dwuwymiarowej o rozmiarach odpowiadających pojemności wyświetlacza,
- znacznik żądania reinicjowania wyświetlacza.
- wektor znaczników żądania aktualizacji poszczególnych wierszy.

W razie potrzeby struktura może zostać rozszerzona przez programistę o inne dane i znaczniki żądania wykonania innych akcji, np. definiowania znaków użytkownika. Ważną cechą takiego interfejsu programowego jest to, że fragmenty oprogramowania, które określają zawartość wyświetlacza, nie wywołują żadnych procedur obsługi sterownika wyświetlacza, a jedynie nadają wartości danym – polom struktury lcd. W celu zainicjowania sterownika wyświetlacza należy ustawić znacznik lcd.req.init. Aby zaktualizować zawartość wyświetlacza należy wpisać nowe dane na odpowiednie pozycje bufora, a następnie, w zależności od potrzeb, ustawić znaczniki żądania modyfikacji odpowiednich wierszy wyświetlacza lcd.req.upd[n] .

Znaczniki akcji są zerowane przez procedurę obsługi wyświetlacza z chwilą rozpoczęcia wykonania zleconej akcji. Dzięki przyjęciu takiej konwencji mamy gwarancję, że zawartość wyświetlacza będzie zawsze aktualna, również w sytuacji, gdy zapis nowego tekstu nastąpi zanim poprzednie zmiany zawartości dotrą do wyświetlacza. Nie ma też potrzeby sprawdzania stanu znaczników przez oprogramowanie zlecające modyfikację – kolejne polecenia będą zawsze wykonywane prawidłowo bez potrzeby czekania na zakończenie wcześniej zleconej akcji. Taki interfejs programowy jest więc całkowicie asynchroniczny i nie wymaga jakiegokolwiek oczekiwania w oprogramowaniu określającym zawartość wyświetlacza.

**Listing 2. Plik lcd.c**

```
#include „f030exp1.h”
#include „lcd.h”

struct lcdctrl_ lcd = {
    {„                „’ ”                „},
    {1}  // init rq
};

// cursor locate commands for each row
static const uint8_t clcmd[] = {0x80, 0xc0};

static const struct {
// LCD init commands
    uint8_t init8[4], init4[6];
} lcdcmd = {
    // 8 bit i/f - send 4 bits only
    {0x30, 0x30, 0x30, 0x20},
    // 4 bit i/f
    {
        0x28,  // mode set
        0x08,  // off
        0x01,  // clear
        0x06,  // entry mode
        0x0c,  // on, cursor off
        0x02  // home
    }
};

static void send4(uint8_t v)
{
    LCD_DATA_SET(v);
    E_L_DELAY;
    LCD_E_SET;
    E_H_DELAY;
    LCD_E_CLR;
}

// LCD service called from timer interrupt routine
void lcdhandler(void)
{
    static enum {LS_RDY, LS_INIT, LS_CMD, LS_DATA} lcdstate = LS_RDY;
    static uint8_t ccnt, dcnt;  // data counter
    static const uint8_t *cptr, *dptr;  // command, data ptr
    static uint8_t lcdtimer = 255;  // initial delay
    if (lcdtimer) lcdtimer --;
    if (!lcdtimer)
    {
        if (lcdstate == LS_RDY)
        {
            if (lcd.req.init)
            {
                lcd.req.init = 0;
                cptr = lcdcmd.init8;
                ccnt = sizeof(lcdcmd.init8);
                 LCD_RS_CLR;
                 lcdstate = LS_INIT;
            }
            else
            {
                static uint8_t lastrow = 0;
                uint32_t i;
                // check for update requests
                for (i = 1; i <= LCDROWS; i ++)
                {
                    uint32_t ridx = (lastrow + i) % LCDROWS;
                    if (lcd.req.upd[ridx])
                    {
                        lcd.req.upd[ridx] = 0;  // clear request
                        cptr = &clcmd[ridx];  // set cursor cmd
                        ccnt = 1;  // 1 command byte
                        dptr = lcd.screen[ridx];
                        dcnt = LCDCOLS;  // no. of data bytes
                        lastrow = ridx;
                        lcdstate = LS_CMD;
                        break;
                    }
                }
            }
        }
        switch (lcdstate)
        {
            case LS_INIT:  // initial commands in 8-bit mode
                send4(*cptr ++ >> 4);      // high nibble only
                lcdtimer = DLY_5MS;
                if (-- ccnt == 0)
                {  // continue init in 4-bit mode
                    ccnt = sizeof(lcdcmd.init4);
                    lcdstate = LS_CMD;
                }
                break;
            case LS_CMD:  // normal commands in 4-bit mode
               if (*cptr < 0x04)  // clear or home command - up to 5 ms
               lcdtimer = DLY_5MS;
            case LS_DATA:  // data phase
                send4(*cptr >> 4);  // high nibble
                send4(*cptr ++);  // low nibble
                if (-- ccnt == 0)
                {  // end of command phase
                    if (dcnt)
                    {  // enter data phase
                        cptr = dptr;
```

## Parametry czasowe i zasady obsługi wyświetlacza w przerwaniu timera

Współpracą ze sterownikiem wyświetlacza zajmuje się wyłącznie procedura obsługi przerwania timera. Określenie parametrów czasowych obsługi wyświetlacza bazuje na kompromisie pomiędzy wydajnością i prostotą budowy procedur obsługi. Przyjęto, że poszczególne akcje nie będą wykonywane z maksymalną możliwą szybkością. Czasy ich wykonania powinny jedynie zagwarantować, że użytkownik nie zauważy przebiegu wykonywania czynności związanych z obsługą wyświetlacza, w tym aktualizacji wyświetlacza – w postaci powolnego pisania kolejnych znaków. W praktyce oznacza to, że całkowity czas wykonania dowol-

**Listing 2. c.d.**

```
                        ccnt = dcnt;
                        dcnt = 0;
                        LCD_RS_SET;
                        lcdstate = LS_DATA;
                    }
                    else
                    {
                        LCD_RS_CLR;
                        lcdstate = LS_RDY;
                    }
                }
                break;
            default: ;  // only to avoid warnings
        }
    }
}
```

**Listing 3. Definicje związane z LCD w pliku f030exp1.h**

```
#define LCD_PORT  GPIOA
#define LCD_E_BIT  4
#define LCD_RS_BIT  6
#define LCD_E_SET  LCD_PORT->BSRR = 1 << LCD_E_BIT
#define LCD_E_CLR  LCD_PORT->BRR = 1 << LCD_E_BIT
#define LCD_RS_SET  LCD_PORT->BSRR = 1 << LCD_RS_BIT
#define LCD_RS_CLR  LCD_PORT->BRR = 1 << LCD_RS_BIT
#define LCD_DATA_OUT(v)  LCD_PORT->BSRR = ((v) & 0xf) | (0xf << 16)
//=============================================================
// LCD timing: E_L + data_setup = E_H = 500 ns
// 8 MHz SYSCLK
#define E_L_DELAY
#define E_H_DELAY  __NOP();__NOP();__NOP()
// no. of lcd timer ticks per 5 ms
#define DLY_5MS  8  // 1600 Hz SysTick
```

**Listing 4. Program demonstracyjny**

```
/*
STM32F0 tutorial
non-blocking LCD service with simultaneous voltage/temperature readout
and UART reception
gbm, 12'2013
*/

#include „f030exp1.h"
#include „lcd.h"
#define SYSCLK_FREQ  (HSI_VALUE)
#define SYSTICK_FREQ  1600
#define BAUD_RATE  115200
#define DEG_SIGN  239  // LCD degree sign code
#define AVG_SHIFT 4

void SystemInit(void)
{
    FLASH->ACR = FLASH_ACR_PRFTBE;  // enable prefetch
}

static const struct init_entry_ init_table[] =
{
     // enable peripherals
    {&RCC->APB2ENR, RCC_APB2ENR_USART1EN | RCC_APB2ENR_ADC1EN},
    {&RCC->AHBENR, RCC_AHBENR_GPIOAEN | RCC_AHBENR_SRAMEN},
    // port setup
    {&GPIOA->AFR[1], BF4(10, 1) | BF4(9, 1)}, // USART pins 10 - RX, 9 - TX
    {&GPIOA->MODER, GPIOA_MODER_SWD
        | BF2(0, GPIO_MODER_OUT) | BF2(1, GPIO_MODER_OUT)  // LCD data
        | BF2(2, GPIO_MODER_OUT) | BF2(3, GPIO_MODER_OUT)
        | BF2(4, GPIO_MODER_OUT) | BF2(6, GPIO_MODER_OUT)  // LCD control
        | BF2(10, GPIO_MODER_AF) | BF2(9, GPIO_MODER_AF)},  // UART pins as AF
    // ADC setup
    {&ADC1->CHSELR, ADC_CHSELR_CHSEL17 | ADC_CHSELR_CHSEL16},  // VREF, TS
    {&ADC1->SMPR, ADC_SMPR_239_5},  // ADC
    {&ADC->CCR, ADC_CCR_TSEN | ADC_CCR_VREFEN},  // ADC
    {&ADC1->CFGR1, ADC_CFGR1_WAIT | ADC_CFGR1_CONT},  // ADC
    // ADC cal
    {&ADC1->CR, ADC_CR_ADCAL},  // ADC
    // USART1 setup
    {(__IO32p)&USART1->BRR, (SYSCLK_FREQ + BAUD_RATE / 2) / BAUD_RATE},
    {&USART1->CR3, USART_CR3_DMAT},  // enable Tx DMA
    {&USART1->CR1, USART_CR1_RXNEIE | USART_CR1_TE | USART_CR1_RE | USART_CR1_UE},  // enable
    //SysTick setup
    {&SCB->SHP[1], 0x80c00000},  // PendSV lower priority, Systick higher
    {&SysTick->LOAD, SYSCLK_FREQ / SYSTICK_FREQ - 1},
    {&SysTick->VAL, 0},
    {&SysTick->CTRL, SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_TICKINT_Msk
        | SysTick_CTRL_ENABLE_Msk},
    // interrupts and sleep
    {&NVIC->ISER[0], 1 << USART1_IRQn},    // enable interrupts
    {&SCB->SCR, SCB_SCR_SLEEPONEXIT_Msk},  // sleep while not in handler
    {0, 0}
};

int main(void)
{
    lcd.screen[0][6] = DEG_SIGN;
    lcd.screen[0][7] = ‚C';
    lcd.screen[0][14] = ‚m';
    lcd.screen[0][15] = ‚V';
    writeregs(init_table);
    __WFI();  // go to sleep
/*}
```

**Listing 4. c.d.**

```
void SysTick_Handler(void)
{
    static uint8_t tdiv = 0;
    static uint8_t sdiv = 0;
    static uint32_t adc_avg[2];
    lcdhandler();
    if ((++ tdiv & 15) == 0)
    {
        int i;
        static enum {ADCH_TS, ADCH_VREF} adch = ADCH_TS;
        if (ADC1->ISR & ADC_ISR_EOC)
        {
            uint32_t val = ADC1->DR;  // 0 before first conversion
            if (adc_avg[adch] == 0)
            {
                // initial measure - set
                adc_avg[adch] = val << AVG_SHIFT;
             }
             else
             {
                 // low-pass filters
                 adc_avg[adch] = adc_avg[adch] + val - (adc_avg[adch] >> AVG_SHIFT);
             }
             if (++ adch > ADCH_VREF)
             adch = ADCH_TS;
        }
        else if (ADC1->ISR & ADC_ISR_ADRDY)
        {
            // ready for conversion
            ADC1->CR = ADC_CR_ADSTART | ADC_CR_ADEN;        // start cont. conversion
        }
        else if ((ADC1->CR & (ADC_CR_ADCAL | ADC_CR_ADEN)) == 0)
        {
            // calibrated but not enabled yet - enable
            ADC1->CR = ADC_CR_ADEN;
        }
        if (++ sdiv == 100)
        {
            uint32_t Vdd_mV;
            int32_t Temperature_x10, tsign;
            sdiv = 0;
            Vdd_mV = VREFINT_CAL * 3300 / (adc_avg[ADCH_VREF] >> AVG_SHIFT);
            // show results
            for (i = 3; i >= 0; i --)
            {
                lcd.screen[0][9 + i] = Vdd_mV % 10 + ,0';
                Vdd_mV /= 10;
            }
            Temperature_x10 = (T_CAL2 - T_CAL1) * 10
            * (((int32_t)adc_avg[ADCH_TS] >> AVG_SHIFT) * VREFINT_CAL - TS_CAL1 * (int32_t)(adc_avg[ADCH_
VREF] >> AVG_SHIFT))
            / (((int32_t)TS_CAL2 - (int32_t)TS_CAL1) * (int32_t)(adc_avg[ADCH_VREF] >> AVG_SHIFT))
                + T_CAL1 * 10;
            tsign = ,+';
            if (Temperature_x10 < 0)
            {
                tsign = ,-';
                Temperature_x10 = -Temperature_x10;
            }
            lcd.screen[0][5] = Temperature_x10 % 10 + ,0';
            lcd.screen[0][4] = ,.';
            Temperature_x10 /= 10;
            i = 3;
            do {
                lcd.screen[0][i --] = Temperature_x10 % 10 + ,0';
                Temperature_x10 /= 10;
            } while (Temperature_x10);
            lcd.screen[0][i --] = tsign;
            while (i >= 0)
            lcd.screen[0][i --] = , ,;
            lcd.req.upd[0] = 1;
        }
    }
}

void USART1_IRQHandler(void)
{
    static uint8_t idx = 0;
    uint32_t c;
    if (USART1->ISR & USART_ISR_RXNE)  // data received
    {
        c = USART1->RDR;
        if (c >= , ,)
        {
            USART1->TDR = c;
            if (idx == 16) for (idx = 0; idx < 15; idx ++)
            {
                lcd.screen[1][idx] = lcd.screen[1][idx + 1];
                lcd.screen[1][idx ++] = c;
                lcd.req.upd[1] = 1;
            }
        }
        else if (c == ,\r')
        {
            USART1->TDR = c;
            USART1->CR1 |= USART_CR1_TXEIE;  // will send ,\n'
            idx = 0;
        }
        else if (c == 0x1b) lcd.req.init = 1;
    }
    if (USART1->ISR & USART1->CR1 & USART_ISR_TXE)
    {
        USART1->CR1 &= ~USART_CR1_TXEIE;
        USART1->TDR = ,\n';
    }
}
```

nej operacji na wyświetlaczu, w tym zmiany zawartości całego wyświetlacza, nie powinien przekroczyć 30 ms. Z danych układu HD44780 wynika, że maksymalny czas wykonania prostych poleceń, w tym zapisu znaku oraz ustawienia pozycji wprowadzania, może wynosić do ok. 120 µs. W przypadku wyświetlacza o organizacji 2×16 znaków odświeżenie wymaga zapisu 32 znaków i dwukrotnego ustawienia pozycji wprowadzania – łącznie 34 operacji o najkrótszym gwarantowanym łącznym czasie wykonania nieco powyżej 4 ms. Okres przerwań timera może więc być wielokrotnie dłuższy od wymaganego przez sterownik wyświetlacza czasu 120 µs, wystarczy, że całkowity czas odświeżenia wyświetlacza zmieści się w założonym górnym limicie. Zbyt krótki okres timera powoduje niepotrzebne zwiększenie obciążenia procesora i może skutkować wydłużeniem czasu odpowiedzi na inne przerwania o niższym lub tym samym priorytecie. Przy przedstawionych założeniach zakres wartości okresu timera odpowiedni do obsługi wyświetlacza 2×16 znaków zawiera się więc w granicach od 120 µs do ok. 880 µs, a dla wyświetlacza 4×20 znaków – od 120 µs do 350 µs, przy czym w miarę możliwości powinien być wybrany okres zbliżony do górnej granicy zakresu, aby nie obciążać niepotrzebnie procesora.

Kolejnym zagadnieniem projektowym jest wybór trybu pracy timera. Ponieważ obsługa wyświetlacza nie jest krytyczna czasowo, w mikrokontrolerach z wielopoziomowym systemem przerwań może ona mieć niski priorytet. Przerwania o takim samym lub wyższych priorytetach wywłaszczania będą powodowały opóźnienia w obsłudze przerwania wyświetlacza. Jeśli czas obsługi tych przerwań będzie długi (np. rzędu setek mikrosekund), może on niekiedy wpłynąć na przesunięcie czasu operacji na wyświetlaczu i spowodować niezachowanie minimalnego odstępu czasu pomiędzy dwoma kolejnymi poleceniami. W takim przypadku timer używany do obsługi wyświetlacza musi pracować w trybie jednokrotnym, z ponownym uruchomieniem na końcu procedury obsługi przerwania timera. Jeżeli okoliczności opisane powyżej nie zachodzą, można użyć przerwania timera zgłaszanego periodycznie ze stałą częstotliwością, co upraszcza nieco projekt oprogramowania – jeden timer może być wtedy zastosowany zarówno do obsługi wyświetlacza, jak i np. odliczania czasu w urządzeniu.

Prezentowany przykład korzysta z przerwania timera zgłaszanego ze stałą częstotliwością 1600 Hz.

## Procedura obsługi przerwania timera

Wszystkie akcje związane z zapisem poleceń i danych do sterownika wyświetlacza są wykonywane przez procedurę lcdhandler(), zrealizowaną w konwencji automatu i wywoływaną z procedury obsługi przerwania timera.

Przedstawiony przykład (**listing 2**) zawiera obsługę trzech akcji: inicjowania wyświetlacza oraz aktualizacji pierwszego i drugiego wiersza. Zastosowano w nim 4-bitowy interfejs wyświetlacza,

Ponieważ okres, z jakim jest zgłaszane przerwanie timera, odpowiada wykonaniu przez sterownik LCD jednej prostej operacji, przy każdym przerwaniu timera może nastąpić zapis do sterownika jednego bajtu, mający na celu np. ustawienie pozycji wprowadzania lub zapis znaku. Okres ten jest jednak zbyt mały dla niektórych poleceń wykonywanych podczas inicjowania sterownika – po zapisie tych poleceń do sterownika niezbędne jest odliczenie większej liczby okresów timera. Dotyczy to poleceń czyszczenia wyświetlacza i powrotu do pozycji początkowej, które mają gwarantowany czas wykonania poniżej 5 ms. Z kolei w sekwencji inicjującej wg. dokumentacji sterownika występują zwiększone odstępy czasowe pomiędzy początkowymi poleceniami ustawiającymi tryb pracy interfejsu. Ponadto w przypadku zastosowania interfejsu 4-bitowego sekwencja inicjująca zawiera na wstępie pojedyncze zapisy 4-bitowe, podczas gdy wszystkie późniejsze akcje są wykonywane w postaci par takich zapisów.

W celu uproszczenia automatu obsługi wyświetlacza przyjęto, że wszystkie odstępy czasowe pomiędzy początkowymi poleceniami w sekwencji inicjującej będą równe, a ich wartość będzie taka sama, jak wartość opóźnień dla poleceń czyszczenia i ustawienia pozycji początkowej – 5 ms. W porównaniu z zastosowaniem wzorcowych minimalnych odstępów czasu zalecanych przez producenta wydłuża to czas inicjowania o kilka milisekund, co nie wpływa na jego poprawność ani nie powoduje opóźnienia widocznego dla użytkownika. Czas opóźnienia wolniejszych poleceń jest odliczany przy użyciu timera programowego dekrementowanego w procedurze obsługi wyświetlacza.

Procedura została zrealizowana w postaci automatu o czterech stanach:

- stan gotowości/bezczynności – brak akcji do wykonania;
- stan inicjowania – polecenia 4-bitowe przesyłane co ok. 5 ms;
- stan zapisu poleceń – polecenia 8-bitowe z odstępem jednego okresu timera, a dla poleceń czyszczenia i przywracania pozycji początkowej – ok. 5 ms;
- stan zapisu danych – kolejne bajty danych (zawartość wyświetlacza lub wzorce znaków definiowanych przez użytkownika) przesyłane z odstępem jednego okresu timera.

Wykrycie ustawienia znacznika żądania akcji w stanie bezczynności po zakończeniu wykonywania polecenia powoduje przejście do stanu inicjowania lub poleceń. W każdym ze stanów oprogramowanie posługuje się dwiema zmiennymi – wskaźnikiem wysyłanych poleceń lub danych oraz licznikiem wysyłanych bajtów. Podczas inicjowania po przesłaniu pierwszych czterech poleceń (w 8-bitowym trybie pracy interfejsu) następuje przejście ze stanu inicjowania do stanu zapisu poleceń, w którym kolejne polecenia są przesyłane już z użyciem interfejsu 4-bitowego. Jeżeli z poleceniem są związane dane, po zakończeniu przesyłania bajtów poleceń następuje ustawienie wskaźnika i licznika danych, zmiana stanu linii RS wyświetlacza i przejście do stanu zapisu danych danych. Po zakończeniu przesyłania danych lub w razie braku potrzeby ich przesyłania automat przechodzi do stanu gotowości.

Przy starcie programu znacznik żądania inicjowania ma wartość 1 – dzięki temu wyświetlacz jest automatycznie inicjowany.

Polecenie inicjowania ma priorytet wyższy niż polecenia aktualizacji. W przypadku, gdy zachodzi obawa, że wskutek częstego ustawiania znacznika żądania aktualizacji pierwszego wiersza nie nastąpi aktualizacja

drugiego lub kolejnych wierszy, można zastosować rotację priorytetów poleceń aktualizacji poszczególnych wierszy wyświetlacza, tak, jak zostało to zrealizowane w przykładzie.

## Prymitywy sterowania HD44780

Procedury zawarte w pliku lcd.c korzystają z makrodefinicji podstawowych akcji obsługi sterownika wyświetlacza, zawartych w pliku f030exp1.h (**listing 3**).

Wyrażenia LCD_E_SET, LCD_E_CLR, LCD_RS_SET i LCD_RS_CLR służą do zmiany stanu linii RS i E. W mikrokontrolerach STM32F0 są one zrealizowane przez zapisy do rejestrów BRR i BSRR. Wyrażenie LCD_DATA_OUT służy do ustawienia stanu linii danych interfejsu wyświetlacza. Można to zrobić poprzez pojedynczy zapisu rejestru BSRR, powodujący selektywne ustawienie stanów linii danych wyświetlacza. Korzystamy tu ze szczególnej cechy rejestru BSRR – przy równoczesnym żądaniu zerowania i ustawiania linii portu wyższy priorytet ma operacja ustawienia stanu 1.

Zmiana stanu wyjść poprzez zapis rejestrów BRR lub BSRR umożliwia uniknięcie konfliktów powodujących błędne ustawienie stanu wyjść podczas operacji na portach wykonywanych przez kod o innych priorytetach, w tym przez procedury obsługi przerwań o wyższych priorytetach.

Wyrażenia E_L_DELAY i E_H_DELAY służą do wprowadzenia opóźnień przed zmianą stanu linii E. Uzyskuje się to przez wykonanie przez procesor odpowiedniej liczby instrukcji pustych , reprezentowanych przez pseudofunkcję __NOP(). Zgodnie z danymi układu HD44780 odstęp pomiędzy zmianami stanu linii E przy zasilaniu sterownika napięciem 3,3 V powinien wynosić nie mniej niż 500 ns, co przy częstotliwości μC równej 8 MHz odpowiada czterem prostym instrukcjom procesora. Od czasu tego należy odjąć czas wykonania instrukcji zapisu danych i zmiany stanu linii E. Ponieważ przygotowanie danych i ich zapis wymaga wykonania przynajmniej czterech instrukcji, przy niskich częstotliwościach procesora wyrażenie E_L_DELAY może być puste.

## Program przykładowy

Program demonstrujący nieblokującą obsługę wyświetlacza (**listing 4**) powstał przez modyfikację poprzedniego przykładu – pomiaru napięcia zasilania i temperatury mikrokontrolera. W górnym wierszu wyświetlacza są wyświetlane wyniki pomiarów wykonywanych w przerwaniu timera, a w dolnym wierszu jest widoczny tekst wpisywany z terminala przez port szeregowy, co następuje w procedurze obsługi przerwania portu szeregowego.

W programie użyto timera SysTick, zaprogramowanego na częstotliwość 1600 Hz. Przy każdym wywołaniu procedura obsługi przerwania timera wywołuje procedurę obsługi wyświetlacza, a przy co 16-tym wywołaniu wykonywany jest kod odpowiedzialny za pomiary.

Port szeregowy jest obsługiwany przy użyciu przerwań. Znaki wprowadzane z terminala są wyświetlane w dolnym wierszu wyświetlacza. Jeśli długość wiersza tekstu przekroczy 16 znaków, wpisywanie kolejnych znaków powoduje przesuwanie tekstu na wyświetlaczu. Wprowadzenie kodu CR powoduje, że kolejne znaki będą wpisywane od pierwszej kolumny. Wpisywane znaki są odsyłane zwrotnie i wyświetlane na terminalu. Wprowadzenie kodu Esc powoduje zainicjowanie sterownika wyświetlacza.

Podobnie jak w poprzednich przykładach, w oprogramowaniu nie występuje pętla zdarzeń. Po zainicjowaniu procesor jest usypiany – działają wyłącznie procedury obsługi przerwań.

## Podsumowanie

W porównaniu z klasyczną obsługą sterownika HD44780 z aktywnym oczekiwaniem, korzystającą z testowania stanu lub opóźnień czasowych, prezentowane rozwiązanie ma kilka istotnych zalet. Są to:

- Brak konieczności sterowania linią R/–W sterownika wyświetlacza.
- Znaczna (kilkunastokrotna) redukcja liczby instrukcji wykonywanych przez procesor w związku z obsługą wyświetlacza, skutkująca oszczędnością energii zużywanej przez urządzenie.
- Pełny asynchronizm, umożliwiający niezależną modyfikację zawartości części wyświetlacza przez różne fragmenty oprogramowania (w tym przez procedury obsługi przerwań) bez konieczności synchronizacji dostępu do wyświetlacza, oczekiwania i jakichkolwiek opóźnień czasowych.

Zaprezentowane rozwiązanie stanowi dowód na nieprawdziwość pojawiających się często w poradnikach opinii, jakoby obsługa wyświetlacza LCD nie mogła zachodzić w przerwaniu i wymagała obecności pętli zdarzeń.

**Grzegorz Mazur**