STM32 – sterowanie wyświetlaczami LED w technice charlieplexing

Charlieplexing (crossplexing) – to sposób sterowania matryc LED przy użyciu minimalnej możliwej liczby połączeń pomiędzy układem sterującym i matrycą. Sama nazwa tego sposobu sterowania techniki jest anegdotą – pochodzi ona od imienia jej domniemanego wynalazcy, Charlie Allena, który w 1995 roku zaproponował jej użycie w produktach firmy Maxim. W rzeczywistości ten sposób sterowania matrycy LED został opatentowany przez AEG-Telefunken już w 1982 roku. Nadaje się on do sterowania wyświetlaczami 7-segmentowymi, matrycami lub dowolnymi zespołami diod LED małej mocy.

O ile przy typowym multipleksowaniu n wyjść układu umożliwia sterowanie nie więcej niż $(n/2)^2$ diod, to przy nharlieplexingu możliwe jest sterowanie n×(n–1) diod. Dysponując trzema wyjściami można sterować 6 diodami, 4 – 12 diodami, a 9 – np. 8-cyfrowym wyświetlaczem 7-segmentowym z kropkami. Oczywiście, ten sposób sterowania oprócz zalet ma też pewne wady. Główną z nich jest konieczność trójstanowego sterowania linii połączonych z diodami LED (a nie dwustanowego, jak przy zwykłym multipleksowaniu). Ponadto poprawne sterowanie matrycą wymaga zapewnienia odpowiednich własności elektrycznych wyjść sterujących, co przy użyciu w tym celu wyjść mikrokontrolera nie zawsze jest możliwe. Problemem dla początkujących programistów może być również nietrywialny sposób wyznaczania stanów wyjść niezbędnych do uaktywnienia wybranej diody matrycy, co jest potrzebne przy zamianie obrazu do wyświetlenia na dane sterujące wyświetlaczem.

Sterowanie elektryczne wyświetlacza charlieplexing

W celu zrozumienia zasady charlieplexingu warto zacząć od podstaw, czyli sposobu sterowania zwykłym wyświetlaczem multipleksowanym. W wyświetlaczu takim mamy dwa rodzaje wyjść sterujących – wyjścia wspólnych elektrod (wyboru cyfr lub wierszy matrycy) oraz wyjścia sterowania segmentów lub kolumn. Wyjścia wspólnych elektrod są wyjściami typu napięciowego – w idealnym przypadku powinny to być klucze o zerowej rezystancji; zwykle w tym charakterze używa się tranzystorów MOS. Wyjścia kolumn/segmentów powinny być sterowanymi źródłami prądowymi – w tym przypadku używamy albo prawdziwych źródeł prądowych dostępnych w specjalizowanych układach scalonych do sterowania diod LED, albo zwykłych wyjść układów logicznych lub wzmacniaczy tranzystorowych z rezystorami szeregowymi, stanowiących uproszczone i dalekie od doskonałości źródła prądowe. W odniesieniu do sterowanych w ten sposób matrycami używamy określeń typów konfiguracji „wspólna anoda" lub „wspólna katoda", mając ma myśli elektrody diod LED sterowane z kluczy napięciowych. W wyświetlaczu ze wspólną anodą anody diod są sterowane przez klucze napięciowe, a katody – przez źródła prądowe.

Przy wyświetlaniu typu charlieplexing każde wyjście układu sterującego służy zarówno do sterowania wierszem, jak i kolumną matrycy. W przypadku pełnej matrycy każde z n wyjść steruje jednym wierszem i n–1 kolumnami matrycy, przy czym wyjście sterujące wierszem nie może równocześnie sterować żadną kolumną w tym wierszu.

Podstawowy schemat najprostszego wyświetlacza typu charlieplexing, z sześcioma diodami LED, przedstawiono na **rysunku 1**. Wyświetlacz taki jest multipleksowany trójfazowo, a w każdej fazie mogą być uaktywnione dwie diody LED.

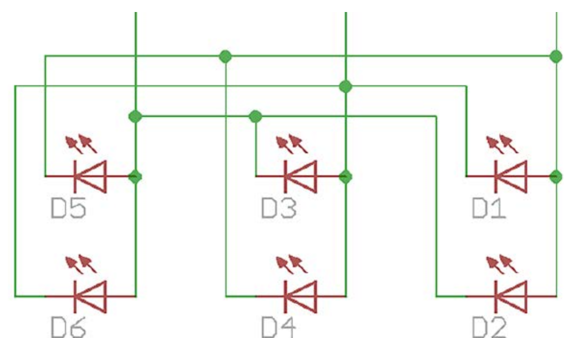
W idealnym wypadku wyjście, które w danej fazie wyświetlania steruje wierszem, powinno pracować jako klucz napięciowy, a wyjścia sterujące kolumnami aktywnymi w danym wierszu – jako źródła prądowe. Wyjście sterujące kolumną nieaktywną w danej fazie jest całkowicie wyłączone – znajduje się ono w stanie wysokiej impedancji. W taki sposób są skonstruowane specjalizowane układy sterowników matryc działające w technice charlieplexing, np. z serii Maxim MAX695x lub Austria Microsystems AS11xx.

Celem tego artykułu jest przedstawienie sposobu ekonomicznego sterowania wieloma diodami LED bez użycia specjalizowanych układów – bezpośrednio z wyjść mikrokontrolera. Rozwiązanie takie jest szczególnie atrakcyjne, gdy np. musimy wysterować z mikrokontrolera kilka diod pełniących funkcję wskaźników, umieszczonych na oddzielnej płytce drukowanej przymocowanej do obudowy urządzenia, minimalizując przy tym liczbę połączeń pomiędzy płytką mikrokontrolera i płytką z diodami.

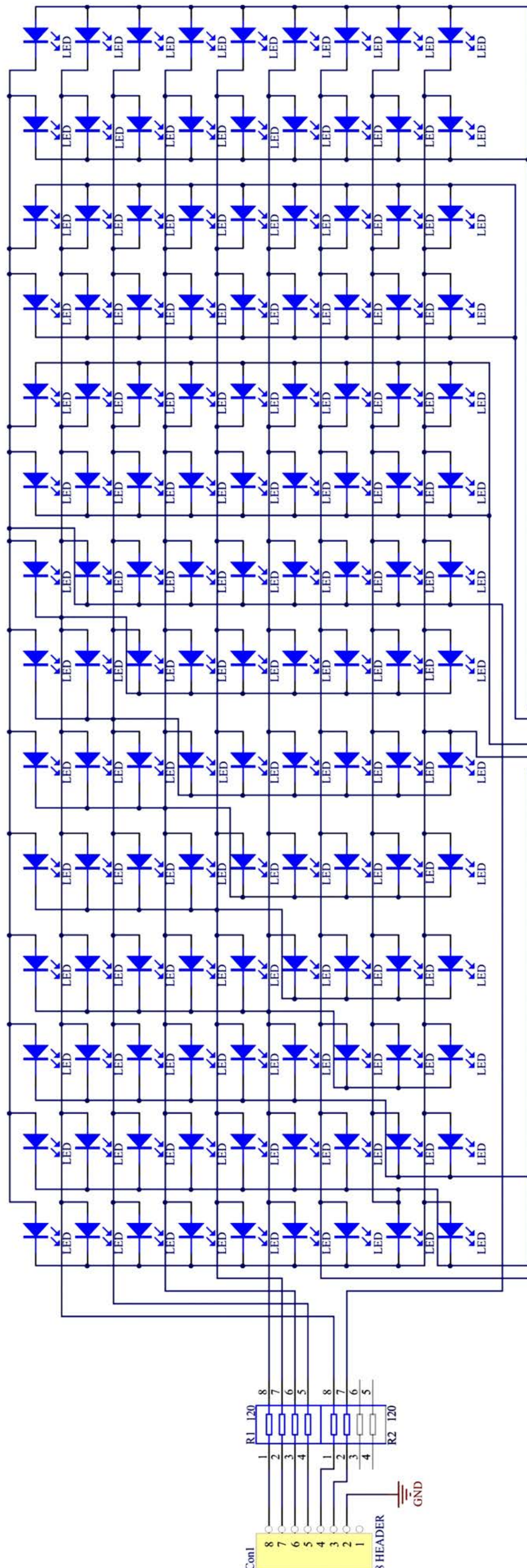
Ponieważ mikrokontrolery nie są zwykle wyposażone w wyjścia mogące pełnić równocześnie funkcję źródeł napięciowych i prądowych, warto przyjrzeć się sposobom elektrycznego sterowania matrycami LED umożliwiającym uzyskanie własności wyjść zbliżonych do idealnych, opisanych powyżej.

Typowo przy sterowaniu matrycy w konwencji charlieplexing używamy rezystorów szeregowych pomiędzy wyjściami mikrokontrolera i matrycą, ograniczających natężenie prądu płynącego przez diody matrycy. Odpowiada to zastosowaniu źródeł prądowych zarówno do wysterowania wierszy, jak i kolumn, co nie jest właściwe, gdyż jasność świecenia diod zależy wówczas od liczby diod zaświecanych równocześnie w wybranym wierszu. Pomimo niedoskonałości tego sposobu sterowania matrycą jest on często stosowany – w taki sposób jest skonstruowany np. popularny moduł LoL (Lots of Lights) w formacie zgodnym z Arduino (**rysunek 2**).

Nieco lepszym i równie tanim rozwiązaniem jest połączenie sterowania wierszami bezpośrednio do wyjść mikrokontrolera, a kolumn



Rysunek 1. Sterowanie 6 diodami LED z 3 wyjść



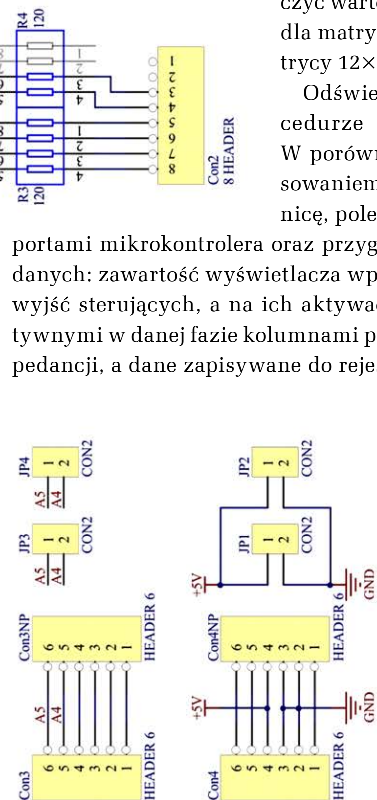
– przez rezystory. W ten sposób nie podwyższając kosztu elementów, uzyskujemy wysterowanie napięciowe wierszy i wysterowanie prądowe kolumn, dzięki czemu jasność diody nie zależy od liczby diod aktywnych w danym wierszu matrycy. W tym przypadku jednak rezystory muszą być umieszczone przy matrycy diod, a nie przy mikrokontrolerze, co nie zawsze jest możliwe. Rozwiązanie takie zastosowano np. w module KA-NUCLEO-MULTISENSOR do sterowania czterema diodami LED RGB.

Zasady programowego sterowania wyświetlaczem

Podobnie jak w przypadku klasycznego multipleksowania, wyświetlacz typu charlieplexing musi być odświeżany ze stałą częstotliwością, gwarantującą wrażenie ciągłości wyświetlanego obrazu (brak migotania). Jeżeli dopuszczamy możliwość ruchu obserwatora względem wyświetlacza, częstotliwość odświeżania nie powinna być niższa od 400 Hz. Górna wartość częstotliwości odświeżania jest uwarunkowana parametrami czasowymi elementów przełączających oraz – przy programowym sterowaniu wyświetlaniem – zajętością czasu procesora.

Maksymalne rozmiary matrycy wynikają z wydajności prądowej wyjść. Maksymalne natężenie prądu płynącego przez wyjście sterujące wierszem jest sumą natężeń prądów wyjść sterujących kolumnami. Stopnie wyjściowe spotykane w większości mikrokontrolerów są specyfikowane na maksymalne natężenie 20 mA. Oznacza to, że maksymalne natężenie prądu pojedynczej diody w matrycy sterowanej przez n wyjść nie może przekroczyć wartości 20 mA / (n – 1), czyli 7 mA dla matrycy 4×3 lub ok. 1,8 mA dla matrycy 12×11 (takiej jak w module LoL).

Odświeżanie następuje zawsze w procedurze obsługi przerwania timera. W porównaniu ze zwykłym multipleksowaniem mamy tu jednak istotną różnicę, polegającą na sposobie sterowania portami mikrokontrolera oraz przygotowaniu używanych do tego danych: zawartość wyświetlacza wpływa nie na poziomy logiczne wyjść sterujących, a na ich aktywację – wyjścia sterujące nieaktywnymi w danej fazie kolumnami pozostają w stanie wysokiej impedancji, a dane zapisywane do rejestrów wyjściowych nie zależą od zawartości wyświetlacza i są zawsze takie same dla danego wiersza matrycy. W przypadku konfiguracji ze wspólną anodą wyjście sterujące wierszem jest na poziomie wysokim, wyjścia sterujące kolumnami – na poziomie niskim, przy czym wyjścia sterujące kolumnami aktywnymi są włączone, a nieaktywnymi – wyłączone.



Podczas przełączania wierszy matrycy w obsłudze przerwania timera należy kolejno:

- Wyłączyć sterowanie liniami portów, ustawiając je w stan wysokiej impedancji (np. poprzez skonfigurowanie ich jako wejść).
- Ustawić w rejestrze wyjściowym nowy stan linii, niezależny od wyświetlanego obrazu i wynikający jedynie z fazy wyświetlania (wybranego wiersza).
- Włączać sterowanie wyjścia wyboru wiersza oraz wyjść sterowania a tymi kolumnami, które w danym wierszu mają być aktywne (zaświecone).

Rysunek 2. Schemat modułu KAMduino LoL Shield zaczerpnięty z dokumentacji tego modułu. Rezystory szeregowe ograniczają natężenie prądu zarówno przy sterowaniu wierszami, jak i kolumnami

Wygodnie jest, gdy wszystkie linie sterujące wyświetlaczem są wyprowadzone z jednego portu GPIO mikrokontrolera – ułatwia to tworzenie programu sterującego wyświetlaczem.

Charlieplexing w STM32

W mikrokontrolerach rodziny STM32 do sterowania trybem pracy linii portu służy rejestr MODER, w którym każdej linii portu odpowiadają dwa kolejne bity. W celu wyłączenia sterowania liniami i ustawienia stanu wysokiej impedancji należy zaprogramować linię jako wejście cyfrowe lub analogowe, co odpowiada kombinacjom 00 lub 11. W celu włączenia sterowania cyfrowego linii należy ustawić ją jako wyjście GPIO, zapisując na odpowiednie pozycje rejestru MODER kombinację 01.

Wyświetlany obraz jest przygotowywany w pamięci w postaci, w której każdy bit odpowiada stanowi pojedynczej diody. Dla ułatwienia tworzenia obrazu wygodnie jest zorganizować dane w postaci odpowiadającej geometrii wyświetlacza. Postać ta w przypadku charlieplexingu nie ma prostego odwzorowania w wartości danych potrzebnych do sterowania wyświetlaczem, dlatego przy każdej zmianie wyświetlanego obrazu należy przygotować nowe dane do sterowania wyświetlaczem, przechowywane w oddzielnym wektorze, którego każdy element odpowiada fizycznemu wierszowi matrycy diod wyświetlacza. Warto zwrócić uwagę, że rozmieszczenie diod nie musi mieć prostego związku ze sposobem ich połączenia, więc organizacja danych sterujących wyświetlaczem jest zupełnie inna od organizacji danych reprezentujących obraz, a zaprojektowanie postaci tych danych i fragmentów kodu przekształcającego obraz w dane sterujące nie jest zadaniem trywialnym.

Przykłady programów dla STM32

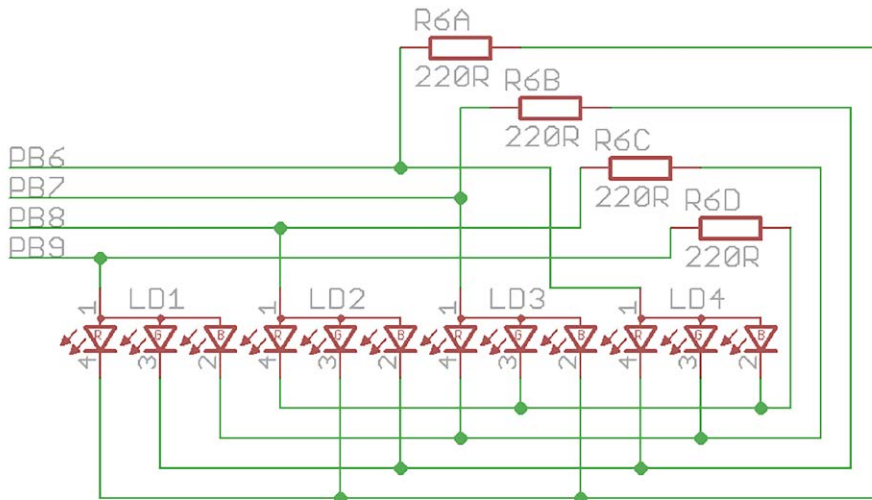
Dwa przedstawione poniżej przykłady pokazują sterowanie czterema diodami RGB (czyli dwunastoma diodami) przy użyciu czterech linii oraz sterowanie matrycą 126 diod przy użyciu 12 linii. Oba przykładowe programy zostały napisane dla mikrokontrolera STM32L476, umieszczonego na płytce Nucleo-64.

Projekty z przykładami są zawarte w archiwum ep_l476_cpx.zip.

Konfigurowanie taktowania mikrokontrolera

Mikrokontroler STM32L476 jest w obu przykładowych projektach taktowany częstotliwością 80 MHz, uzyskaną przez powielenie częstotliwości 4 MHz pochodzącej z wewnętrznego generatora MSI. W celu uzyskania takiego taktowania należy kolejno:

- skonfigurować parametry głównej pętli PLL dla uzyskania częstotliwości 80 MHz z częstotliwości wejściowej 4 MHz, pochodzącej z MSI,
- skonfigurować parametry dostępu do pamięci Flash odpowiednie dla docelowej częstotliwości pracy
- poczekać na synchronizację PLL i włączyć taktowanie mikrokontrolera z PLL.



Rysunek 3. Matryca 4 diod RGB z rezystorami ograniczającymi tylko natężenie prądu katod, umieszczona na płytce KA-NUCLEO-MULTISENSOR

Listing 1. Plik cpx.c – Sterowanie czterech diod RGB

```
/*
    KA-NUCLEO-MULTISENSOR LED mpx demo
    gbm 12'2017
*/

#include "stm32l4yy.h"
#include "ka_nuc_multis.h"
#define SYSCLK_FREQ 80000000u
#define CPX_FREQ 1600u
#define CPXROWS 4

int main(void)
{
    RCC->AHB2ENR |= RCC_AHB2ENR_GPIOBEN;
    // Clock setup: PLL - 80 MHz
    RCC->PLLCFGR = RCC_PLLCFGR_PLLREN | RCC_PLLCFGR_PLLNV(40)
          | RCC_PLLCFGR_PLLMV(1) | RCC_PLLCFGR_PLLSRC_MSI;
    RCC->CR |= RCC_CR_PLLON;
    // set Flash speed
    FLASH->ACR |= FLASH_ACR_PRFTEN | FLASH_ACR_LATENCY_4WS;
    while (!(RCC->CR & RCC_CR_PLLRDY));
    RCC->CFGR |= RCC_CFGR_SW_PLL;
    SysTick_Config(SYSCLK_FREQ / CPX_FREQ);
    SCB->SCR = SCB_SCR_SLEEPONEXIT_Msk; // sleep while not in handler
    __WFI();        // go to sleep
}

// CPX MODER OR masks for 12 LEDs
static const uint32_t cpxmask[] =
{
    BF2(CPX_LED3_BIT, GPIO_MODER_OUT) | BF2(CPX_LED0_BIT, GPIO_MODER_OUT),// right R
    BF2(CPX_LED3_BIT, GPIO_MODER_OUT) | BF2(CPX_LED1_BIT, GPIO_MODER_OUT),
    BF2(CPX_LED3_BIT, GPIO_MODER_OUT) | BF2(CPX_LED2_BIT, GPIO_MODER_OUT),
    BF2(CPX_LED2_BIT, GPIO_MODER_OUT) | BF2(CPX_LED3_BIT, GPIO_MODER_OUT),
    BF2(CPX_LED2_BIT, GPIO_MODER_OUT) | BF2(CPX_LED0_BIT, GPIO_MODER_OUT),
    BF2(CPX_LED2_BIT, GPIO_MODER_OUT) | BF2(CPX_LED1_BIT, GPIO_MODER_OUT),
    BF2(CPX_LED1_BIT, GPIO_MODER_OUT) | BF2(CPX_LED2_BIT, GPIO_MODER_OUT),
    BF2(CPX_LED1_BIT, GPIO_MODER_OUT) | BF2(CPX_LED3_BIT, GPIO_MODER_OUT),
    BF2(CPX_LED1_BIT, GPIO_MODER_OUT) | BF2(CPX_LED0_BIT, GPIO_MODER_OUT),
    BF2(CPX_LED0_BIT, GPIO_MODER_OUT) | BF2(CPX_LED1_BIT, GPIO_MODER_OUT),
    BF2(CPX_LED0_BIT, GPIO_MODER_OUT) | BF2(CPX_LED2_BIT, GPIO_MODER_OUT),
    BF2(CPX_LED0_BIT, GPIO_MODER_OUT) | BF2(CPX_LED3_BIT, GPIO_MODER_OUT) // left B
};

static uint32_t cpxctrl[CPXROWS];       // cpx control data
// m: bgrbgrbgrbgr, msb - left diode, lsb - right diode
static void cpx_encode(uint32_t m)
{
    cpxctrl[0] = 0;
    cpxctrl[1] = 0;
    cpxctrl[2] = 0;
    cpxctrl[3] = 0;
    for (uint32_t i = 0; i < 12; i ++)
    {
        if (m & 1) cpxctrl[i / 3] |= cpxmask[i];
        m >>= 1;
    }
}

void SysTick_Handler(void)
{
    static uint8_t phase;
    CPX_PORT->MODER &= CPX_MODER_OFF;
    CPX_PORT->BSRR = CPX_BSRR_OFF | CPX_LED3_MSK >> phase;
    CPX_PORT->MODER |= cpxctrl[phase];
    if (++phase == CPXROWS) phase = 0;
    static uint16_t tdiv;
    if (++tdiv == CPX_FREQ)
    {
        tdiv = 0;
        // prepare image
        static uint8_t seq;
        if (++seq == CPXROWS) seq = 0;
        cpx_encode(042104210 >> (seq * 3));
    }
}
```

Listing 3. Obsługa wyświetlacza multipleksowanego przy użyciu DMA

```
/*
    STM32L4 KAMduino LoL shield demo
    gbm, 11'2016
*/
#include "stm32l4yy.h"
#include "stm32nucleo64.h"
#include "stm32nucleo64_ard.h"
#define SYSCLK_FREQ   80000000u
#define SYSTICK_FREQ  4000u
#define IMGROWS    9
#define IMGCOLS    14
#define CPXPHASES 12
#define PATTERN 0x03030303  // bar pattern to animate
#define GPIOA_MODER_CPXOFF  GPIOA_MODER_SWDA
#define GPIOB_MODER_CPXOFF  0xffffffff
#define GPIOC_MODER_CPXOFF  0xffffffff
#define GPIOA_BRR_CPXOFF    AR_D_GPIOA_MASK
#define GPIOB_BRR_CPXOFF    AR_D_GPIOB_MASK
#define GPIOC_BRR_CPXOFF    AR_D_GPIOC_MASK

enum gpio_ {PA, PB, PC}; // gpio port indices
GPIO_TypeDef * const portmap[] = {GPIOA, GPIOB, GPIOC};

// CPX pin indices (NOT pin numbers)
enum dind_ {D2, D3, D4, D5, D6, D7, D8, D9, D10, D11, D12, D13};
// pin map - gpio ports and bits for Arduino conn D2..D13 pins
const struct {
    uint8_t port:4, bit:4;
} ard_pinmap[] = {
    {PA, AR_D2_BIT},
    {PB, AR_D3_BIT},
    {PB, AR_D4_BIT},
    {PB, AR_D5_BIT},
    {PB, AR_D6_BIT},
    {PA, AR_D7_BIT},
    {PA, AR_D8_BIT},
    {PC, AR_D9_BIT},
    {PB, AR_D10_BIT},
    {PA, AR_D11_BIT},
    {PA, AR_D12_BIT},
    {PA, AR_D13_BIT}
};

// anode, cathode pin for each LOL LED
static const struct {
    uint8_t a:4, c:4;
} lolmap[IMGROWS][IMGCOLS] = {
    {{D13, D5}, {D13, D6}, {D13, D7}, {D13, D8}, {D13, D9}, {D13, D10}, {D13, D11},
    {D13, D12}, {D13, D4}, {D4, D13}, {D13, D3}, {D3, D13}, {D13, D2}, {D2, D13}},
    {{D12, D5}, {D12, D6}, {D12, D7}, {D12, D8}, {D12, D9}, {D12, D10}, {D12, D11},
    {D12, D13}, {D12, D4}, {D4, D12}, {D12, D3}, {D3, D12}, {D12, D2}, {D2, D12}},
    {{D11, D5}, {D11, D6}, {D11, D7}, {D11, D8}, {D11, D9}, {D11, D10}, {D11, D12},
    {D11, D13}, {D11, D4}, {D4, D11}, {D11, D3}, {D3, D11}, {D11, D2}, {D2, D11}},
    {{D10, D5}, {D10, D6}, {D10, D7}, {D10, D8}, {D10, D9}, {D10, D11}, {D10, D12},
    {D10, D13}, {D10, D4}, {D4, D10}, {D10, D3}, {D3, D10}, {D10, D2}, {D2, D10}},
    {{D9, D5}, {D9, D6}, {D9, D7}, {D9, D8}, {D9, D10}, {D9, D11}, {D9, D12},
    {D9, D13}, {D9, D4}, {D4, D9}, {D9, D3}, {D3, D9}, {D9, D2}, {D2, D9}},
    {{D8, D5}, {D8, D6}, {D8, D7}, {D8, D9}, {D8, D10}, {D8, D11}, {D8, D12},
    {D8, D13}, {D8, D4}, {D4, D8}, {D8, D3}, {D3, D8}, {D8, D2}, {D2, D8}},
    {{D7, D5}, {D7, D6}, {D7, D8}, {D7, D9}, {D7, D10}, {D7, D11}, {D7, D12},
    {D7, D13}, {D7, D4}, {D4, D7}, {D7, D3}, {D3, D7}, {D7, D2}, {D2, D7}},
    {{D6, D5}, {D6, D7}, {D6, D8}, {D6, D9}, {D6, D10}, {D6, D11}, {D6, D12},
    {D6, D13}, {D6, D4}, {D4, D6}, {D6, D3}, {D3, D6}, {D6, D2}, {D2, D6}},
    {{D5, D6}, {D5, D7}, {D5, D8}, {D5, D9}, {D5, D10}, {D5, D11}, {D5, D12},
    {D5, D13}, {D5, D4}, {D4, D5}, {D5, D3}, {D3, D5}, {D5, D2}, {D2, D5}}
};

int main(void)
{
    // PLL - 40 MHz
    RCC->PLLCFGR = RCC_PLLCFGR_PLLREN | RCC_PLLCFGR_PLLNV(40) | RCC_PLLCFGR_PLLMV(1) | RCC_PLLCFGR_PLLSRC_MSI;
    RCC->CR |= RCC_CR_PLLON;  //
    RCC->AHB2ENR = RCC_AHB2ENR_GPIOAEN | RCC_AHB2ENR_GPIOBEN | RCC_AHB2ENR_GPIOCEN;
    FLASH->ACR |= FLASH_ACR_PRFTEN | FLASH_ACR_LATENCY_4WS;
    GPIOA->PUPDR = GPIOA_PUPDR_SWD;      // leave only SWD pulls
    GPIOB->PUPDR = 0;          // turn off default pulls
    while (!(RCC->CR & RCC_CR_PLLRDY));
    RCC->CFGR |= RCC_CFGR_SW_PLL;
    while ((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_PLL);
    SysTick_Config(SYSCLK_FREQ / SYSTICK_FREQ);
    SCB->SCR = SCB_SCR_SLEEPONEXIT_Msk;
    __WFI();  // sleep
}

void SysTick_Handler(void)
{
    static uint16_t lol_img[IMGROWS];   // image to display - 9 rows x 14 cols
    static _Bool lol_upd;      // update request flag
    static uint8_t cpxphase;  // current cpx phase

    static uint32_t moder_val[CPXPHASES][3];
    GPIOA->MODER = GPIOA_MODER_CPXOFF;  // all LEDs off
    GPIOB->MODER = GPIOB_MODER_CPXOFF;  // all LEDs off
    GPIOC->MODER = GPIOC_MODER_CPXOFF;  // all LEDs off
    GPIOA->BRR = GPIOA_BRR_CPXOFF;
    GPIOB->BRR = GPIOB_BRR_CPXOFF;
    GPIOC->BRR = GPIOC_BRR_CPXOFF;
    GPIOA->MODER = moder_val[cpxphase][PA];
    GPIOB->MODER = moder_val[cpxphase][PB];
    GPIOC->MODER = moder_val[cpxphase][PC];
    portmap[ard_pinmap[cpxphase].port]->BSRR = 1u << ard_pinmap[cpxphase].bit;
    if (++ cpxphase == CPXPHASES)
    {
        cpxphase = 0;
        if (lol_upd)
        {
            lol_upd = 0;
            // convert bitmap to MODER content
            // anode activation
            for (uint32_t ph = 0; ph < CPXPHASES; ph ++)
            {
```

Listing 3. cd.
```
                moder_val[ph][PA] = GPIOA_MODER_CPXOFF;
                moder_val[ph][PB] = GPIOB_MODER_CPXOFF;
                moder_val[ph][PC] = GPIOC_MODER_CPXOFF;
                moder_val[ph][ard_pinmap[ph].port] &= BF2A(ard_pinmap[ph].bit, GPIO_MODER_OUT);
            }
            // cathode activation
            for (uint32_t r = 0; r < IMGROWS; r ++)
                for (uint32_t c = 0; c < IMGCOLS; c ++)
                    if (lol_img[r] >> c & 1)
                    {
                        uint32_t cathode = lolmap[r][c].c;
                        moder_val[lolmap[r][c].a][ard_pinmap[cathode].port] &= BF2A(ard_pinmap[cathode].bit, GPIO_MODER_OUT);
                    }
        }
    }
    static uint16_t tdiv;
    if (++ tdiv == SYSTICK_FREQ / 10)
    {
        tdiv = 0;
        // animate arrow pattern
        static uint8_t s;
        lol_img[8] = lol_img[0] = PATTERN >> s;
        lol_img[7] = lol_img[1] = PATTERN >> (s + 1);
        lol_img[6] = lol_img[2] = PATTERN >> (s + 2);
        lol_img[5] = lol_img[3] = PATTERN >> (s + 3);
        lol_img[4] = PATTERN >> (s + 4);
        if (++s == 8) s = 0;
        lol_upd = 1;
    }
}
```

Wymienione powyżej czynności są wykonywane w obu programach na początku funkcji main(). Ponieważ maksymalne dozwolone częstotliwości pracy wszystkich szyn wewnętrznych mikrokontrolera L476 wynoszą 80 MHz, nie ma potrzeby konfigurowania dzielników częstotliwości poszczególnych szyn – wszystkie peryferiale, w tym timery, są taktowane częstotliwością 80 MHz.

Przykład 1 – Sterowanie czterech diod RGB przy użyciu czterech wyjść mikrokontrolera

Pierwszy przykład demonstruje sterowanie diod RGB umieszczonych na płytce KA-NUCLEO-MULTISENSOR umieszczonej w złączach Morpho, zawartych na płytce Nucleo-64. Schemat połączenia diod przedstawia **rysunek 3**. Diody są sterowane z czterech linii portu GPIOB. Program został uruchomiony na płytce L476RG-Nucleo, zawierającej mikrokontroler STM32L476RGT.

Program, przedstawiony na **listingu 1**, animuje sekwencję, w której kolejno zaświecane są na poszczególnych pozycjach trzy kolory podstawowe. Plik nagłówkowy ka_nuc_multis.h zawiera definicje zasobów płytki, w tym linii portów, do których jest podłączony wyświetlacz. Ponadto w skład projektu wchodzą trzy własne pliki zawierające definicje przydatne przy programowaniu mikrokontrolerów rodziny STM32. Stanowią one uzupełnienie pliku definicji zasobów mikrokontrolera dostarczonego przez producenta.

Po każdej modyfikacji obrazu następuje przygotowanie danych do sterowania wyświetlaczem. Mamy to do czynienia z multipleksowaniem 4-fazowym, więc dane sterujące składają się z 4 słów 32-bitowych, zawierających wartości wpisywane do rejestru MODER portu GPIOB w poszczególnych fazach wyświetlania.

Dane do wyświetlania są przygotowywane w postaci 12 bitów 16-bitowego argumentu, przekazywanego do funkcji cpx_encode(). Każde kolejne trzy bity odpowiadają stanowi trzech składowych jednej diody RGB. Funkcja cpx_encode() najpierw zeruje elementy wektora sterującego wyświetlaniem, a następnie wpisuje do nich przy użyciu operacji sumy logicznej maski odpwiadające aktywacji diod, które mają być zaświecone.

Do przekodowania obrazu na postać odpowiadającą zawartości rejestru MODER służy 12-elementowy wektor, którego każdy element zawiera maskę bitową służącą do aktywowania sterowania wiersza i kolumny sterujących daną diodą. Każde kolejne trzy elementy wektora odpowiadają jednej diodzie RGB o wspólnej anodzie i mogą służyć do uzyskania maski dla jednej fazy wyświetlania, sterującej świeceniem wszystkich składowych danej diody RGB.

Do odświeżania wyświetlacza i animacji jego zawartości użyto przerwania timera SysTick, zgłaszanego z częstotliwością 1600 Hz. W przerwaniu następuje kolejno:

- deaktywacja wyjść sterujących wyświetlaniem,
- ustawienie stanów logicznych wyjść dla kolejnej fazy wyświetlania,
- włączenie wyjść sterujących diodami, które mają być zaświecone.

Co jedną sekundę wyświetlany obraz jest zmieniany, a nowy wzorzec zaświeconych diod jest zamieniany na słowa sterujące dla rejestru MODER poprzez wywołanie funkcji cpx_encode().

Przykład 2 – Sterowanie matrycy 126 diod przez 12 wyjść mikrokontrolera

Drugi przykład demonstruje sterowanie matrycy 126 diod umieszczonej na płytce KAMduino LoL, zgodnej elektycznie z popularnym modułem Arduino LoL. Matryca zawiera 9 wierszy po 14 diod, jednak jej organizacja elektryczna jest zupełnie inna – diody są połączone w 12 wierszy zawierających po 11 lub 9 diod. Organizacja sterowania nie ma więc w tym przypadku prostego odwzorowania w geometrii wyświetlacza, co komplikuje projekt oprogramowania.

Matryca jest sterowana z linii D2..13 złącza Arduino. Dodatkowym problemem jest fakt, że linie te są na płytce Nucleo-64 dołączone do trzech portów GPIO: PA, PB i PC. Projekt struktur danych służących do sterowania wyświetlaczem wymaga starannego przemyślenia.

Wygenerowany obraz jest przechowywany w 9-elementowym wektorze słów 16 bitowych lol_img[], w którym każde słowo reprezentuje jeden wiersz wyświetlacza. Ponieważ mamy do czynienia z multipleksowaniem 12-fazowym, dane sterujące wyświetlaczem mają postać tablicy dwuwymiarowej moder_val, złożonej z dwunastu 3-elementowych wektorów słów 32-bitowych, przesyłanych w kolejnych fazach do rejestrów MODER portów GPIOA, GPIOB i GPIOC. Zamiana obrazu bitowego na wartości sterujące rejestrami MODER wymaga zdefiniowania kilku dodatkowych struktur danych i wyliczeń enum ułatwiających dostęp do nich.

Wyliczenie gpio_ służy do intuicyjnego indeksowania portów GPIO. Wektor portmap[] zawiera adresy portów GPIO. Wyliczenie dind_ definiuje symbole dla linii D2..13 złącza Arduino. Wektor ard_pinmap zawiera struktury złożone z indeksów i numerów linii portów GPIO, odpowiadających liniom D2…D13 Arduino. Dwuwymiarowa tablica lolmap zawiera struktury określające numery linii Arduino sterujących anodą i katodą każdej z diod matrycy. Opisuje ona odwzorowanie pikseli matrycy obrazu w położenie diody w matrycy sterowania.

Fragment kodu odpowiedzialny za przekodowanie obrazu pobiera z tablicy lolmap numery linii Arduino, a następnie używa ich do zaindeksowania wektora ard_pinmap w celu wyliczenia masek bitowych dla rejestrów MODER i umieszczenia ich we właściwych elementach tablicy moder_val.

Program, przedstawiony na **listingu 2**, animuje obraz przesuwających się na wyświetlaczu strzałek. Podobnie. jak w poprzednim przykładzie, do odświeżania wyświetlacza i animacji jego zawartości użyto przerwania timera SysTick,. Tym razem jednak z powodu znacznie większej liczby faz multilpleksowaniam jest ono zgłaszane z częstotliwością 4000 Hz. W przerwaniu następuje kolejno:

- deaktywacja wyjść portów PA, PB i PC sterujących wyświetlaniem,
- wyzerowanie wyjść,
- włączenie wyjść sterujących wierszem i kolumnami, które mają być zaświecone,
- ustawienie wyjścia sterującego wierszem.

Obraz do wyświetlenia jest zmieniany co 1 sekundę, a nowy wzorzec zaświeconych diod jest przetwarzany do postaci tablicy słów sterujących dla rejestrów MODER po zakończeniu najbliższego pełnego cyklu odświeżania. W ten sposób unikamy artefaktów, które mogłyby powstawać, gdyby zmiana obrazu nie była zsynchronizowana z cyklem odświeżania wyświetlacza.

Grzegorz Mazur