Światła choinkowe z układami WS2812B sterowane z STM32NUCLEO

Od kilku lat dostępne są na rynku trójkolorowe diody LED ze zintegrowanymi sterownikami, zgodnymi logicznie z układem WS2811 chińskiego producenta WorldSemi. Elementy te są wytwarzane przez kilku producentów pod różnymi nazwami i w różnych obudowach. Najbardziej rozpowszechnione są układy typu WS2812B, zawierające diodę RGB i sterownik w 4-końcówkowej obudowie do montażu powierzchniowego typu 5050. Układy PL9823 firmy BaiCheng występują również w typowych obudowach LED do montażu przewlekanego, o średnicy 5 lub 8 mm. Podobne do WS2812 są również układy APA104 produkowane przez Shenzhen Led Color Opto Electronic. Od układów WorldSemi różnią się one logicznie kolejnością wartości składowych w strumieniu danych (BRG dla WS2812, RGB dla P9823 i APA104).

Wymienione elementy są dostępne w postaci wielu gotowych produktów, w tym w łańcuchach świateł połączonych przewodami, sztywnych listwach i matrycach oraz giętkich taśmach. Taśmy mają długość od 1 do 5 m i zawierają od 30 do 144 elementów na metr bieżący. Są one wyposażone w zunifikowane złącza, umożliwiające łączenie ich w dłuższe łańcuchy. Zazwyczaj przy zakupie taśmy otrzymujemy również przewód ze złączem, służący do podłączenia zasilania i sterowania do taśmy.

W opisywanym projekcie świateł choinkowych zastosowano taśmę zwierającą 144 układy na metr. Taśmy o takiej gęstości upakowania elementów umożliwiają uzyskanie szczególnie spektakularnych animacji, dając wrażenie płynności ruchu animowanych obiektów. Oprogramowanie może zostać skonfigurowane do sterowania dowolną liczbą układów; próby przeprowadzono z taśmami o długości od 1, 2, 3 i 4 metrów (odpowiednio 144, 288, 432 i 576 układów).

Zasilanie taśm z WS2812

Projektując oświetlenie efektowe z układami WS2812 należy zwrócić uwagę na moc pobieraną przez układy i wynikające z niej natężenie prądu zasilania oraz sposób doprowadzenia zasilania.

Taśma ze 144 elementami przy wysterowaniu wszystkich diod na pełną jasność pobiera prąd o natężeniu do 10 A. Niezbędna moc zasilacza i sposób doprowadzenia napięcia zasilającego do taśm zależą więc istotnie od z maksymalnej używanej jasności (wartości wypełnienia) diod oraz maksymalnej liczby równocześnie zaświeconych diod. Praktyka wskazuje, że w przypadku użycia taśm w niewielkim i niezbyt jasno oświetlonym pomieszczeniu maksymalne wypełnienie dla diod nie powinno przekraczać 1/8. Ponieważ efekty świetlne zrealizowane w sterowniku powodują równoczesne świecenie nie więcej niż ¼ wszystkich diod, maksymalne natężenie prądu zasilania można szacować na ok. 0.3 A na 144 elementy. Oznacza to, że sterownik z jedną lub nawet dwiema taśmami może być zasilany podczas prób z interfejsu USB komputera używanego do jego programowania, a sterowane w ten sposób taśmy mogą być zasilane przy użyciu tylko złącz trójstykowych, służących również do transmisji danych.

W przypadku użycia pełnej jasności diod może okazać się konieczne oddzielne doprowadzenie zasilania do każdej taśmy z dwóch stron przy użyciu dodatkowych przewodów, wyprowadzonych standardowo z każdej taśmy.

Jeżeli nie korzystamy z dodatkowych doprowadzeń zasilania, niezbędne jest ich usunięcie lub zaizolowanie w celu wyeliminowania możliwości wystąpienia przypadkowego zwarcia zasilania.

Sterowanie układów WS2812

Opis realizacji transmisji danych do układów WS2812 znajduje się w artykule [EP_WS2812], dostępnym w archiwum EP (numer 3/2014). W projekcie tym dane były transmitowane przy użyciu interfejsu SPI, z kodowaniem danych i obsługą SPI z wykorzystaniem przerwań. Rozwiązanie takie ma jedną istotną zaletę: minimalizuje ono zajętość pamięci RAM mikrokontrolera. Jego wadami są konieczność pracy z 12-bitowymi ramkami SPI oraz znaczne użycie mocy obliczeniowej procesora związane z obsługą przerwań zgłaszanych z częstotliwością około 200 kHz. Warto podkreślić, że pomimo tak dużej częstotliwości przerwań rozwiązanie takie zużywa znacznie mniejszą część czasu procesora niż całkowicie programowe formowanie strumienia danych dla WS2812, z którego korzysta popularne oprogramowanie dla płytek rodziny Arduino.

Praca z 12-bitowymi ramkami nie jest możliwa w mikrokontrolerach serii STM32F4, w których interfejs SPI używa wyłącznie ramek 8- albo 16-bitowych. Z drugiej strony pojemność pamięci RAM dostępna w serii STM32F4, jak również w większych modelach serii STM32F0, umożliwia przechowywanie całego zakodowanego strumienia danych dla łańcucha WS2812, zajmującego trzykrotnie więcej miejsca niż jego postać wyjściowa, niezakodowana, a dostępny w mikrokontrolerach moduł bezpośredniego dostępu do pamięci (DMA) umożliwia jego transmisję przez interfejs SPI bez udziału procesora. Rozwiązanie zastosowane w prezentowanym projekcie korzysta więc z transmisji przez SPI wektora przygotowanych wcześniej przez oprogramowanie ramek 16-bitowych przy użyciu DMA. Dzięki temu niemal cała dostępna moc obliczeniowa procesora może być użyta do animacji obiektów i ich rasteryzacji.

W pamięci RAM mikrokontrolera są przechowywane następujące struktury danych:

- Dane animowanych obiektów, zależne od algorytmów animacji, o całkowitym rozmiarze do kilku kB.
- Wartości składowych RGB dla poszczególnych diod – 3 bajty na element, ok. 1700 B przy 576 szt. WS2812.
- Dane zakodowane do transmisji przez SPI – 9 B na element, ok, 5 kB przy 576 szt. WS2812.

Listing 1. Plik board.h
```
#if defined(STM32F091) || defined(STM32F051)
#include „stm32f0yy.h”
#ifdef PLLMUL
#define SYSCLK_FREQ  (HSI_VALUE * PLLMUL / 2)
#else
#define SYSCLK_FREQ  (HSI_VALUE)
#endif
#elif defined(STM32F401xC) || defined(STM32F411xE)
#include „stm32f4yy.h”
#endif
#include „stm32util.h”
#include „stm32nucleo64.h”
// timing
#define SYSTICK_FREQ 100  // 100 Hz -> 10 ms
// WS2812B strip defs
#define NPIXELS 288
#define LED_MAX 30  // maximum LED duty
```

Jak wynika z powyższych szacunków, w sterowniku powinien zostać zastosowany model mikrokontrolera wyposażony w min. 16 kB pamięci RAM. Warunki te spełniają m.in. STM32F072, STM32F091 oraz wszystkie modele serii STM32F4.

Sterownik taśm WS2812 – sprzęt

Do realizacji dwóch wersji sterownika użyto płytek STM32 NUCLEO z mikrokontrolerami STM32F091 i STM32F411. Oprogramowanie może zostać łatwo przekonfigurowane dla dowolnego mikrokontrolera serii STM32F0 lub STM32F4 umieszczonego na płytce serii NUCLEO, DISCOVERY lub własnej użytkownika; można je również niezbyt dużym nakładem pracy zaadaptować dla mikrokontrolerów z innych serii rodziny STM32F.

Sterownik jest połączony z taśmą trzema przewodami:, dołączonymi do złącza MORPHO CN10 z prawej strony płytki NUCLEO. Są to linie:

- masy GND (CN10-9),
- zasilania U5V (CN10-8),
- linii danych MOSI/PA7 (CN10-15).

Uwaga: przy takim połączeniu płytka NUCLEO wraz z taśmą LED może być zasilana albo wyłącznie z interfejsu USB komputera albo wyłącznie z zewnętrznego zasilacza, służącego do zasilania taśm. W przypadku użycia zasilacza zewnętrznego, interfejs USB płytki NUCLEO nie może być połączony z komputerem

Struktura oprogramowania sterownika

Całe oprogramowanie składa się z dwóch głównych modułów zapisanych w oddzielnych plikach źródłowych. Pierwszy moduł jest odpowiedzialny za konfigurację peryferiali mikrokontrolera i obsługę transmisji danych do łańcucha układów WS2812. Moduł ten jest specyficzny dla danej serii mikrokontrolerów. W przedstawionym projekcie użyto dwóch realizacji tego modułu – jednej dla serii STM32F0, drugiej dla STM32F4. Drugi moduł zawiera oprogramowanie sterujące animacją. Jest ono niezależne od typu użytego mikrokontrolera. Moduł animacji nie jest silnie związany z modułem transmisji danych i może zostać łatwo wymieniony na inny, z innymi algorytmami animacji obiektów.

Ponadto w skład projektu wchodzi trzeci plik źródłowy, zawierający procedury wywoływane z modułu animacji i zapewniający zapamiętywanie ostatnio wybranego trybu pracy w pamięci Flash mikrokontrolera przy zaniku zasilania. Podobnie jak w przypadku kodu odpowiedzialnego za transmisję danych, również obsługa pamięci Flash jest specyficzna dla poszczególnych serii mikrokontrolerów.

Pliki

Oprogramowanie zostało przygotowane przy użyciu darmowej wersji środowiska Keil MDK-ARM 5.17. Wszystkie pliki źródłowe projektu są zawarte w pliku archiwum ws2812ctree.zip, dostępnym na serwerze EP. Archiwum składa się z następujących folderów:

- Keil – zawiera foldery z plikami projektów dla obu użytych mikrokontrolerów,
- b_nucleo64 – zawiera pliki definicji zasobów płytki NUCLEO64 oraz plik board.h z parametrami konfiguracji sterownika efektów świetlnych,
- Common – zawiera pomocnicze pliki definicji zasobów mikrokontrolerów STM32F
- WS2812 – zawiera dwie wersje modułu z funkcją main() i obsługą transmisji: WS2812dma-f0.c i WS2812dma-f4.c,
- led-anim – zawiera procedury animacji efektów świetlnych i dwie wersje obsługi pamięci Flash.

W celu skompilowania projektów w środowisku Keil niezbędne jest zainstalowanie przy użyciu Pack Installer trzech pakietów: ARM::CMSIS, Keil::STM32F0_DFP i Keil::STM32F4_DFP.

Plik *board.h* jest włączany do wszystkich plików źródłowych projektu. Zawiera on dyrektywy *#include* włączające pliki specyficzne dla użytego w projekcie mikrokontrolera, definicje stałych związanych z taktowaniem wersji z STM32F0 oraz podstawowe parametry sterownika – częstotliwość generowani ramek animacji, liczbę pikseli w łańcuchu i maksymalne wypełnienie dla diod.

Inicjowanie mikrokontrolera i obsługi transmisji

Przy zastosowaniu interfejsu SPI do transmisji danych do układów WS2812, każdy przesyłany oktet musi zostać zakodowany w postaci 24 bitów, a częstotliwość transmisji bitów powinna wynosić około 2.4 MHz [EP_WS2812]. Ponieważ w mikrokontrolerach serii STM32F częstotliwość transmisji SPI jest uzyskiwana przez podział częstotliwości szyny APB mikrokontrolera przez potęgę dwójki z zakresu 2..256, łatwą do uzyskania częstotliwością transmisji jest 2.5 MHz, a w celu jej uzyskania mikrokontroler powinien być taktowany częstotliwością będącą wielokrotnością 20 MHz.

W wersji oprogramowania dla mikrokontrolera STM32F091 główny plik źródłowy nosi nazwę *WS2812dma-f0.c*. Ponieważ domyślnie włączany do projektu plik *system_stm32f0xx.c*, pochodzący z pakietu Keil:STM32F0_DFP zawiera funkcję SystemInit() programującą generator zegar w nieodpowiedni dla naszego zastosowania sposób, należało zmienić nazwę tej funkcji w pliku *system_stm32f0xx.c* i zdefiniować własną funkcję *SystemInit()* w pliku *WS2812dma-f0.c*.

W realizacji z STM32F091 użyto częstotliwości rdzenia mikrokontrolera i szyny APB równej 40 MHz, uzyskiwanej z generatora PLL taktowanego wewnętrznym generatorem HSI o częstotliwości ok. 8 MHz.

W wersji oprogramowania dla mikrokontrolera STM32F411 główny plik źródłowy nosi nazwę *WS2812dma-f4.c*. Źródłem głównego przebiegu zegarowego jest syntezer PLL taktowany wewnętrznym generatorem HSI. Główny zegar mikrokontrolera ma częstotliwość 80 MHz, a zegary szyn APB – 40 MHz.

Kolejnym krokiem po zaprogramowaniu generatora zegara jest zainicjowanie modułów peryferyjnych. Do transmisji danych jest używany moduł SPI1. Aktywne są jedynie dwa wyjścia MOSI tego modułu, wyprowadzone na linie PA7 i PB5 – łańcuch układów WS2812 może być podłączony do jednego z dwóch dostępnych wyjść.

Po zainicjowaniu portów i interfejsu SPI1 jest wywoływana procedura inicjowania modułu animacji *app_init()*. Następnie jest programowany układ wykrywania spadku napięcia zasilania – włączane jest generowanie przerwań przy spadku napięcia poniżej 2.8 V. Pseudofunkcja *SysTick_Config* ustawia timer Systick na zgłaszanie przerwań z częstotliwością 100 Hz, która jest częstotliwością generowania kolejnych faz animacji. Wywołanie to obniża jednocześnie priorytet przerwania SysTick, co jest istotne dla zapewnienia szybkiej reakcji na zanik zasilania. Ostatnią czynnością jest włączenie w procesorze funkcji uśpienia przy wyjściu z obsługi wyjątku i uśpienie procesora w oczekiwaniu na przerwanie.

Wszystkie użyte w projekcie układy peryferyjne są w obu wersjach projektu inicjowane w podobny sposób. Różnice pomiędzy zawartościami rejetrów sterujących w obu wersjach wynikają z drobnych różnic w budowie peryferiali pomiędzy STM32F0 i STM32F4.

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 60086, pass: sjh7zycq

Listing 2. Plik WS2812dma-f0.c

```
/*
    STM32F0
    WS2812B control with SPI & DMA, PLL in use, SysTick-driven animation
    gbm, 11'2015
*/
#define PLLMUL 10 // 40 MHz clock
#include „board.h"
#include „WS2812.h"
//=============================================================================
void SystemInit(void)
{
    RCC->CFGR = RCC_CFGR_PLLMULV(PLLMUL);
    RCC->CR |= RCC_CR_PLLON;  // turn PLL on
}
//=============================================================================
static const struct init_entry_ init_table[] =
{
    {&FLASH->ACR, FLASH_ACR_PRFTBE | 1},  // enable prefetch, 1 wait state
    {&RCC->CFGR, RCC_CFGR_PLLMULV(PLLMUL) | RCC_CFGR_SW_PLL},  // switch to PLL clock
    // enable peripherals
    {&RCC->APB1ENR, RCC_APB1ENR_PWREN},
    {&RCC->APB2ENR, RCC_APB2ENR_SPI1EN},
    {&RCC->AHBENR, RCC_AHBENR_RSTVAL | RCC_AHBENR_GPIOAEN | RCC_AHBENR_GPIOBEN
        | RCC_AHBENR_GPIOCEN | RCC_AHBENR_DMAEN},
    // port setup - SPI1 MOSI at PA7 and PB5; SPI1 is fun 0 (default)
    {&GPIOA->OSPEEDR, BF2(7, GPIO_OSPEEDR_MED)},  // MOSI - medium speed
    {&GPIOA->MODER, GPIOA_MODER_SWD       | BF2(7, GPIO_MODER_AF)},  // MOSI as AF
    {&GPIOB->OSPEEDR, BF2(5, GPIO_OSPEEDR_MED)},  // MOSI - medium speed
    {&GPIOB->MODER, BF2(5, GPIO_MODER_AF)},  // MOSI as AF
    // SPI setup
    {(__IO32p)&SPI1->CR2, SPI_CR2_DSIZE(16) | SPI_CR2_TXDMAEN},
    {(__IO32p)&SPI1->CR1, SPI_CR1_SSM | SPI_CR1_SSI | SPI_CR1_SPE | SPI_CR1_BRDIV16 | SPI_CR1_MSTR},  //
enable
    //SysTick setup
    {&SCB->SHP[1], 0x80c00000},  // PendSV lowest 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},
    // Flash unlock
    {&FLASH->KEYR, FLASH_FKEY1},
    {&FLASH->KEYR, FLASH_FKEY2},
    // PVD setup
    {&PWR->CR, PWR_CR_PLSV(5) | PWR_CR_PVDE},  // 2.58 V, enable
    // interrupts and sleep
    {&EXTI->RTSR, EXTI_MR_PVD},  // PVD
    {&EXTI->IMR, EXTI_MR_PVD},  // PVD
    {&NVIC->ISER[0], 1 << PVD_IRQn},  // enable interrupts
    {0, 0}  // 3108 vs 3252
};
//=============================================================================
extern void app_init(void);
void WEAK app_init(void) { }

int main(void)
{
  while (!(RCC->CR & RCC_CR_PLLRDY));      // wait for PLL lock
  writeregs(init_table);
  app_init();
  SCB->SCR = SCB_SCR_SLEEPONEXIT_Msk;      // sleep while not in handler
  __WFI();       // go to sleep
}
//=============================================================================
// WS2812 reset pulse width
// (50 us -> 120 bits @2.4 MHz, 125 bits @2.5 MHz)
#define WS_RST_FRM8 // no. of 16-bit frames for RESET pulse
//=============================================================================
#define NBYTES (NPIXELS * 3)
// no. of 16-bit frames for LED string control (9 frames per pixel pair)
#define NFRAMES (WS_RST_FRM + (NPIXELS + 1) / 2 * 9)

void WS2812_start(struct wspix_ *ptr)
{
    static uint16_t enc_wsdata[NFRAMES];
    static const uint16_t encode[] = {
        04444, 04446, 04464, 04466, 04644, 04646, 04664, 04666,
        06444, 06446, 06464, 06466, 06644, 06646, 06664, 06666
    };  // bit-to triple encoding table - 4->12 bits
    if (DMA1->ISR & DMA_ISR_TCIF3 || DMA1_Channel3->CCR == 0)
    {
        uint8_t *wsptr = (uint8_t *)ptr;
        uint16_t *ep = enc_wsdata + WS_RST_FRM;     // skip reset frames
        for (uint32_t i = 0; i < NBYTES; i +=2)
        {
            uint32_t ev;
            // encode 2 bytes as 3 16-bit words
            ev = encode[*wsptr >> 4] << 12;
            ev |= encode[*wsptr ++ & 0xf];  // 24 bits in ev
            *ep ++ = ev >> 8;      // store 16 bits
            ev <<= 12;    // 8 bits left
            ev |= encode[*wsptr >> 4];  // 20 b total
            *ep ++ = ev >> 4;      // 4 bits left
            *ep ++ = ev << 12 | encode[*wsptr ++ & 0xf];  // 16 bits
        }
        // setup WS2812 data transfer
        DMA1->IFCR = DMA_IFCR_CGIF3;
        DMA1_Channel3->CCR = 0;  // disable
        DMA1_Channel3->CPAR = (uint32_t)&SPI1->DR;
        DMA1_Channel3->CMAR = (uint32_t)enc_wsdata;
        DMA1_Channel3->CNDTR = NFRAMES;
        // medium priority, increment memory adress, mem->periph, enable
```

Listing 2. c.d.

```
        DMA1_Channel3->CCR = DMA_CCR_PL_0 | DMA_CCR_MSIZE16 | DMA_CCR_PSIZE16
            | DMA_CCR_MINC | DMA_CCR_DIR | DMA_CCR_EN;
        // clear buffer
        DMA1_Channel1->CCR = 0;  // disable
        DMA1_Channel1->CPAR = (uint32_t)enc_wsdata;  // constant zero
        DMA1_Channel1->CMAR = (uint32_t)ptr;  //
        DMA1_Channel1->CNDTR = (NBYTES + 3) >> 2;
        // increment memory adress, mem->periph, enable
        DMA1_Channel1->CCR = DMA_CCR_MEM2MEM | DMA_CCR_MSIZE32 | DMA_CCR_PSIZE32
            | DMA_CCR_MINC | DMA_CCR_EN;
    }
}
```

Listing 3. Plik WS2812dma-f4.c

```
/*
    STM32F4
    WS2812B control with SPI & DMA, PLL in use, SysTick-driven animation
    gbm, 11'2015
*/
#include „board.h”
#include „WS2812.h”
//============================================================================
#define SYSCLK_FREQ        80000000u
extern void app_init(void);
void WEAK app_init(void) {}

int main(void)
{
    // PLL - 80 MHz
    RCC->PLLCFGR = (RCC->PLLCFGR & RCC_PLLCFGR_RSVD)
        | RCC_PLLCFGR_PLLQV(8) | RCC_PLLCFGR_PLLPV(4)
        | RCC_PLLCFGR_PLLNV(160) | RCC_PLLCFGR_PLLMV(8);
    RCC->CR |= RCC_CR_PLLON;  //
    RCC->CFGR = RCC_CFGR_PPRE2_DIV2 | RCC_CFGR_PPRE1_DIV2;  // APB2, APB1 prescaler = 2
    // set Flash speed
    FLASH->ACR = FLASH_ACR_PRFTEN | FLASH_ACR_LATENCY_2WS;  // 2ws 64..90
    while (!(RCC->CR & RCC_CR_PLLRDY));
    RCC->CFGR |= RCC_CFGR_SW_PLL;
    while ((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_PLL);
    // enable peripherals
    RCC->AHB1ENR = RCC_AHB1ENR_GPIOAEN | RCC_AHB1ENR_GPIOBEN
        | RCC_AHB1ENR_GPIOCEN | RCC_AHB1ENR_DMA2EN;
    RCC->APB1ENR = RCC_APB1ENR_PWREN;
    RCC->APB2ENR = RCC_APB2ENR_SPI1EN;
    // port setup - SPI1 MOSI at PA7 and PB5
    GPIOA->AFR[0] = BF4(7, 5);  // SPI1: AF5
    GPIOB->AFR[0] = BF4(5, 5);  // SPI1: AF5
    GPIOA->OSPEEDR = GPIOA_OSPEEDR_SWD | BF2(7, GPIO_OSPEEDR_MED);  // MOSI - medium speed
    GPIOA->MODER = GPIOA_MODER_SWD | BF2(7, GPIO_MODER_AF);  // MOSI as AF
    GPIOB->OSPEEDR = BF2(5, GPIO_OSPEEDR_MED);  // MOSI - medium speed
    GPIOB->MODER = BF2(5, GPIO_MODER_AF);  // MOSI as AF
    // SPI setup
    SPI1->CR2 = SPI_CR2_TXDMAEN;
    SPI1->CR1 = SPI_CR1_DFF | SPI_CR1_SSM | SPI_CR1_SSI | SPI_CR1_SPE | SPI_CR1_BRDIV16 | SPI_CR1_MSTR;
    // Flash unlock
    FLASH->KEYR = FLASH_FKEY1;
    FLASH->KEYR = FLASH_FKEY2;
    app_init();
    PWR->CR = PWR_CR_PLSV(6) | PWR_CR_PVDE;  // PVD: 2.8 V, enable
    while (PWR->CSR & PWR_CSR_PVDO);
    // interrupts and sleep
    EXTI->RTSR = EXTI_MR_PVD;  // PVD
    EXTI->IMR = EXTI_MR_PVD;  // PVD
    NVIC_EnableIRQ(PVD_IRQn);
    SysTick_Config(SYSCLK_FREQ/100);
    SCB->SCR = SCB_SCR_SLEEPONEXIT_Msk;
    __WFI();  // sleep
}
//============================================================================
// WS2812 reset pulse width
// (50 us -> 120 bits @2.4 MHz, 125 bits @2.5 MHz)
#define WS_RST_FRM8 // no. of 16-bit frames for RESET pulse
//============================================================================
#define NBYTES     (NPIXELS * 3)
// no. of 16-bit frames for LED string control (9 frames per pixel pair)
#define NFRAMES    (WS_RST_FRM + (NPIXELS + 1) / 2 * 9)

void WS2812_start(struct wspix_ *ptr)
{
    static uint16_t enc_wsdata[NFRAMES];
    static const uint16_t encode[] = {
        04444, 04446, 04464, 04466, 04644, 04646, 04664, 04666,
        06444, 06446, 06464, 06466, 06644, 06646, 06664, 06666
    };  // bit-to triple encoding table - 4->12 bits
    if (DMA2->LISR & DMA_LISR_TCIF3 || DMA2_Stream3->CR == 0)
    {
        uint8_t *wsptr = (uint8_t *)ptr;
        uint16_t *ep = enc_wsdata + WS_RST_FRM;     // skip reset frames
        uint32_t ev;
        for (uint32_t i = 0; i < NBYTES; i ++)
        {
            // encode each 2 bytes as 3 16-bit words
            ev = encode[*wsptr >> 4] << 12;
            ev |= encode[*wsptr ++ & 0xf];  // 24 bits in ev
            *ep ++ = ev >> 8;
            ev <<= 8;     // 8 bits left, left-align at 16-bit boundary
            if (++ i < NBYTES)  // not the last byte
            {
                ev <<= 4;  // make place for 12 bits
                ev |= encode[*wsptr >> 4];  // 20 b total
                *ep ++ = ev >> 4;
```

Listing 2. c.d.

```
                ev <<= 12;  // 4 bits left
                ev |= encode[*wsptr ++ & 0xf];  // 16 bits
            }
            *ep ++ = ev;
        }
        // setup WS2812 data transfer - DMA2 Stream3 ch3
        DMA2_Stream3->CR = 0;  // disable
        DMA2->LIFCR = DMA_LIFCR_CALLF3 | DMA_LIFCR_CALLF2;  // clear stream4 flags
        DMA2_Stream3->PAR = (uint32_t)&SPI1->DR;
        DMA2_Stream3->M0AR = (uint32_t)enc_wsdata;
        DMA2_Stream3->NDTR = NFRAMES;
        // medium_priority, increment memory adress, mem->periph, enable
        DMA2_Stream3->CR = DMA_SxCR_CHSELV(3) | DMA_SxCR_PL_0 | DMA_SxCR_MSIZE16 | DMA_SxCR_PSIZE16
            | DMA_SxCR_MINC | DMA_SxCR_DIR_M2P | DMA_SxCR_EN;
        // clear buffer using Stream2
        DMA2_Stream2->CR = 0;      // disable
        DMA2_Stream2->PAR = (uint32_t)enc_wsdata;  // constant zero
        DMA2_Stream2->M0AR = (uint32_t)ptr;  // buf to clear
        DMA2_Stream2->NDTR = (NBYTES + 3) >> 2;
        // increment memory adress, mem->mem, enable
        DMA2_Stream2->CR = DMA_SxCR_MSIZE32 | DMA_SxCR_PSIZE32
            | DMA_SxCR_MINC | DMA_SxCR_DIR_M2M | DMA_SxCR_EN;
    }
}
```

Obsługa transmisji danych

Do inicjowania transmisji danych do WS2812 służy funkcja WS2812, umieszczona w głównym pliku źródłowym wraz z procedurą inicjującą. Obie wersje (dla F0 i F4) są zbudowane analogicznie, a jedyna różnica pomiędzy nimi polega na odmiennym programowaniu modułu DMA, którego implementacje w obu seriach mikrokontrolerów wykazują istotne różnice.

Funkcja WS2812_start zawiera instrukcję warunkową, pomijającą całą transmisję danych w przypadku, gdy poprzednia transmisja nie została zakończona. Dzięki temu w można np. zachować stałą częstotliwość generowania ramek przez moduł animacji, niezależnie od liczby sterowanych elementów – przy większej liczbie elementów nie będą jednak wyświetlane wszystkie wygenerowane ramki.

Działanie funkcji rozpoczyna się od zakodowania danych do transmisji. Zakodowane dane zostają umieszczone w buforze enc_wsdata[]. Pierwsze ramki danych w buforze pozostają niewykorzystane i mają stałą wartość 0 – ich transmisja służy do wygenerowani symbolu RESET dla układów WS2812. Następnie zostają zainicjowane dwie transmisje DMA – jedna z nich przesyła dane z bufora *enc_wsdata[]* do interfejsu SPI, a druga służy do wyzerowania bufora zawierającego niezakodowane kolory poszczególnych pikseli. Zerowanie bufora przez DMA jest szybsze niż zerowanie programowe, więc ograniczamy w ten sposób zajętość procesora, zwiększając czas dostępny dla procedury sterowania animacją. Warto zauważyć, że ze względu na użycie do zerowania bufora kanału DMA o priorytecie arbitrażu wyższym niż kanału używanego do transmisji danych, zachodzi potrzeba programowego podwyższenia priorytetu kanału, z którego korzysta SPI podczas programowania rejestru konfiguracji kanału. W przeciwnym przypadku transmisja SPI mogłaby zostać opóźniona.

Zawartość obu wersji pliku z inicjowaniem mikrokontrolera i obsługą transmisji zamieszczono poniżej

Moduł animacji

Moduł animacji jest niezależny od typu mikrokontrolera i wspólny dla obu wersji projektu. Jest on zapisany w pliku ctree1.c. Na początku pliku są zdefiniowane stałe – parametry animacji. Dwie główne procedury zawarte w tym pliku – to procedura app_init(), wywoływana jednokrotnie podczas inicjowania sterownika oraz procedura obsługi przerwania timera systemowego SysTick.

Moduł animacji umożliwia wyświetlanie czterech rodzajów obiektów należących do dwóch typów – nieruchomych, pojedynczych pikseli o modulowanej jasności oraz zmieniających położenie grup pikseli. Do animacji tych dwóch typów obiektów służą oddzielne fragmenty programu i struktury danych. Animacja może działać w pięciu trybach. W czterech z nich wyświetlane są obiekty jednego z dostępnych typów, w piątym (domyślnym po zaprogramowaniu sterownika) – wyświetlane są równocześnie obiekty wszystkich czterech typów. Tryb animacji jest przełączany przy użyciu przycisku użytkownika (w kolorze niebieskim) na płytce NUCLEO.

Procedura obsługi przerwania przy zaniku zasilania *PVD_IRQHandler()* zapamiętuje ostatnio wybrany tryb pracy w pamięci Flash. Procedura *app_init()* odczytuje z pamięci Flash zapamiętany podczas wyłączania tryb pracy sterownika. Dzięki temu przy kolejnym włączeniu sterownik zachowuje ostatnio ustawiony tryb pracy.

Procedury animacji

Kod odpowiedzialny za animację obiektów jest umieszczony w procedurze obsługi timera SysTick, wywoływanej z częstotliwością określoną w pliku board.h. parametry animacji zdefiniowane na początku pliku ctree1.c zostały obliczone dla częstotliwości generowania ramek animacji równej 100 Hz. Generowanie kolejnych ramek animacji składa się z trzech faz:

- generowania nowych obiektów,
- animowania obiektów,
- rasteryzacji obrazu obiektów.

Warunkiem utworzenia nowego obiektu jest wolne miejsce w strukturze danych opisującej obiekty. Maksymalna liczby obiektów każdego typu jest stała, a wygenerowanie nowego jest możliwe po zakończeniu życia jednego z dotychczas aktywnych obiektów. Ponadto pomiędzy utworzeniem każdego obiektu każdego typu musi upłynąć określony czas.

Zarówno odstęp czasowy generowania obiektów jak i wartości wszelkich parametrów obiektów są wyznaczane przy użyciu generatora liczb pseudolosowych, zrealizowanego w postaci rejestru przesuwającego LFSR.

Losowanie koloru obiektu

Pierwszy krokiem przy tworzeniu nowego obiektu jest losowanie jego koloru. Aglgorytm losowania koloru został opracowany w taki sposób, by zapewnić w miarę kontrastowe kolory kolejnych obiektów. Suma jasności składowych koloru obiektu jest stała. Co drugi kolor obiektu pochodzi z kolejnych odcinków trójkąta barw rgb (jedna ze składowych ma wartość 0).

Tworzenie obiektów

Utworzenie obiektu w postaci nieruchomego piksela polega na wylosowaniu jego położenia. W celu uniknięcia nakładania i sklejania obiektów punktowych obiektów losowanie powtarzane jest do uzyskania pozycji nie zajmowanej przez żaden inny obiekt punktowy ani nie sąsiadującej z nią bezpośrednio.

Poza punktami oprogramowanie może tworzyć dwa typy obiektów ruchomych – piksele porszające się w górę taśmy oraz „fajerwerki" wystrzeliwane w górę taśmy, a następnie „spadające" w dół. Poruszające się piksele

Listing 3. Parametry animacji zdefiniowane w pliku ctree1.c

```
#define NDOTS (NPIXELS / 10) // max. no of acive dots
#define DOT_GEN_DELAY   50   // min. dot generation spacing
#define DOT_TWUP_SPEED  4    // time to full intensity
#define DOT_TWDN_SPEED  2    // off time
#define DOT_STEADY_TIME 150  // max. on time

#define NWORMS (NPIXELS / 12) // max no. Of moving objects
#define CLIMB_SPEED  4        // microunits (1/256 pixel) per time unit
#define WORM_GEN_MIN 300      // min. worm generation delay
#define WORM_GEN_SPAN ((NPIXELS * 256 / CLIMB_SPEED / NWORMS - WORM_GEN_MIN) * 2)  // max. random gen. delay
#define FSCALE    2048
// fireworks flight time - 0 to top, depends on no. of pixels.
// MAXFTIME^2 / FSCALE = (NPIXELS - 1)
#if     NPIXELS == 60
#define MAXFTIME 348
#elif   NPIXELS == 144
#define MAXFTIME 543
#elif   NPIXELS == 288
#define MAXFTIME 767
#elif   NPIXELS == 432
#define MAXFTIME 940
#elif   NPIXELS == 576
#define MAXFTIME 1087
#endif
#define MINFTIME (MAXFTIME - 131)
#define FTDELTA 33  // diff 131 = 4 * 33 - 1
```

z chwilą dotarcia do końca taśmy zamieniają się w „spadające" fajerwerki. Szybkość ruchu pikseli jest stała. Szybkość początkowa fajerwerków jest zmienna w pewnym przedziale, dzięki czemu fajerwerki docierają w pobliże końca taśmy, ale na różne wysokości.

Użycie dobrej jakości generatora pseudolosowego do wyznaczania wartości parametrów obiektów powoduje, że w całej animacji nie daje się zauważyć powtarzalności ani regularności.

Animowanie obiektów nieruchomych

Animowanie nieruchomych pikseli polega na modulacji ich jasności. Dostępne są dwa rodzaje modulacji: piksele „statyczne" zaświecają się płynnie, świecą ze stałą jasnością przez losowy, niezbyt długi czas, a następnie płynnie gasną. Piksele „dynamiczne" mają zmienną jasność, przypominającą migotanie świecy. Ich czas życia jest znacznie dłuższy od czasu świecenia pikseli statycznych.

Animowanie obiektów ruchomych

Animowanie obiektów ruchomych polega na zmianie ich pozycji, a w przypadku „fajerwerków" - również na symulacji efektu powolnego wygaszania smugi światła za poruszającym się obiektem. Dzięki temu fajerwerki pozostawiają za sobą ślad o długości zależnej od bieżącej prędkości wznoszenia lub opadania. Smuga za fajerwerkami jest uzyskiwana przez przechowywanie kolorów ciągu pikseli i płynne wygaszanie ich ze stałą szybkością.

Pozycja czoła każdego obiektu ruchomego jest określana z dokładnością 1/256 piksela. Ułamkowa część pozycji jest używana jako parametr do antyaliasingu kolorów pikseli.

Parametry animacji

Poniżej zamieszczono fragment pliku ctree1.c z definicjami stałych sterujących animacją. Przy stałych są umieszczone komentarze opisujące ich znaczenie.

Wszystkie parametry czasowe animacji są wyrażone w jednostkach 1/100 sekundy. Stała MAXFTIME określa maksymalny czas wznoszenia dla fajerwerków, obliczony w taki sposób, by docierały one do końca taśmy. Czas ten zależy od liczby pikseli w sterowanych taśmach. Stała MINFTIME określa pośrednio minimalną wysokość, do której mogą docierać fajerwerki, a stała FTDELTA – różnicę czasu wznoszenia dla kolejnych fajerwerków.

Rasteryzacja obiektów

Ostatnią fazą procedury animacji jest rasteryzacja obiektów. W przypadku obiektów ruchomych odbywa się ona z uwzględnieniem pozycji ułamkowych (obiekt położony „pomiędzy" pikselami), co daje wrażenie większej płynności ruchu, zwłaszcza przy małych prędkościach.

Po rasteryzacji jest wywoływana procedura WS2812_start() enkodująca dane dla WS2812 i inicjująca ich transmisję do łańcucha układów.

Grzegorz Mazur

Bibliografia

1. *[EP_WS2812] G. Mazur, 32 bity jak najprościej (3) STM32F0 – płytka eksperymentalna*
2. *z mikrokontrolerem STM32F030F4, Elektronika Praktyczna 03/2014, s.90..96*
3. *[WS2812B] WorldSemi, WS2812B Intelligent control LED integrated light source, 2014, www.world-semi.com*
4. *[UM1724] UM1724 User manual STM32 Nucleo boards, Rev 7, ST Microelectronics 2015*
5. *[RM0383] RM0383 Reference manual, STM32F411xC/E advanced ARM®-based 32-bit MCUs, Rev 1, ST Microelectronics, July 2014*
6. *[F411DS] STM32F411xC STM32F411xE Datasheet, Rev 4, ST Microelectronics, February 2015*
7. *[RM0091] RM0091 Reference manual, STM32F0x1/STM32F0x2/STM32F0x8 advanced ARM®-based 32-bit MCUs, Rev 8, ST Microelectronics, July 2015*
8. *[F091DS] STM32F091xB STM32F091xC Datasheet, Rev 2, ST Microelectronics, February 2015*

Niezbędnik każdego elektronika

ELEKTRONIKA PRAKTYCZNA

www.ep.com.pl

Międzynarodowy magazyn elektroników konstruktorów