32 bity jak najprościej (7)

STM32F0 – nieblokująca obsługa interfejsu 1-Wire

Opracowany przez firmę Dallas Semiconductors (obecnie Maxim Integrated), popularny interfejs 1-Wire umożliwia dołączenie do mikrokontrolera wielu układów peryferyjnych przy użyciu pojedynczej linii sygnałowej i linii masy. Układy o niewielkim poborze mocy mogą być zasilane z linii danych interfejsu, dzięki czemu można dołączyć do mikrokontrolera np. sieć czujników korzystając tylko z dwóch przewodów.

Opis interfejsu jest zawarty m.in. w dokumencie [DS18B20]. Interfejs 1-Wire nie jest obsługiwany sprzętowo przez typowe mikrokontrolery. Producent układów publikuje noty aplikacyjne zawierające opis programowej implementacji protokołu 1-Wire [AN126] oraz realizację protokołu przy użyciu interfejsu UART [APP214].

Na najniższym poziomie – transmisji bitów – specyfikacja interfejsu definiuje trzy cykle: inicjowania (RESET), zapisu i odczytu. Każdy cykl jest poprzedzony nieaktywnym (wysokim) poziomem linii danych stanowiącym zakończenie poprzedniego cyklu i gwarantującym zgromadzenie w układach zasilanych z linii danych energii potrzebnej w czasie poziomu aktywnego (niskiego). Cykl bitowy rozpoczyna się wyzerowaniem linii danych przez układ nadrzędny. Po czasie określonym w protokole transmisji następuje wyłączenie sterowania linii danych przez układ nadrzędny. Układ podrzędny może przetrzymać linię na poziomie niskim przez dodatkowy czas – w ten sposób nadaje on dane do układu nadrzędnego. Każdy cykl interfejsu jest charakteryzowany przez trzy podstawowe parametry czasowe:

- Czas impulsu startu – wyzerowania linii danych przez układ nadrzędny.
- Czas okna danych, w którym poziom linii danych niesie informacje o wartości bitu danych.
- Czas nieaktywny cyklu, po którym można rozpocząć następny cykl.

Cykl inicjowania charakteryzuje się dużo dłuższymi czasami charakterystycznymi od cykli transmisji danych. Uaktywnienie linii danych przez układ podrzędny podczas okna danych w cyklu inicjowania służy do zgłoszenia układowi nadrzędnemu obecności układu podrzędnego.

Typowa realizacja programowa układu nadrzędnego interfejsu 1-Wire polega na odczekiwaniu odcinków czasu protokołu i odpowiednich zmianach i próbkowaniu stanu linii danych. Podczas wymiany danych z układem podrzędnym mikrokontroler nadrzędny musi zapewnić odpowiednią dokładność odmierzania odcinków, co oznacza, że nie może on w tym czasie wykonywać żadnych innych czynności, w tym obsługiwać przerwań. Może to powodować gubienie zdarzeń lub opóźnienia w realizowanym algorytmie sterowania.

Program przykładowy

Prezentowany przykład ilustruje pozbawioną tych wad, nieblokującą obsługę interfejsu 1-Wire. Dzięki temu transmisje danych nie powodują blokowania procesora na czas ich wykonywania i mogą być one inicjowane z procedur obsługi przerwań.

Przedstawiona na **listingu 1** implementacja 1-Wire bazuje na przerwaniach timera, a do jej realizacji jest potrzebny timer, który może równocześnie odmierzać trzy odcinki czasu, generując przerwania po każdym z nich. Cechę taką ma większość timerów mikrokontrolerów STM32F – te, które są wyposażone w rejestr końca cyklu ARR i przynajmniej dwa rejestry porównania CCRx, jak np. TIM1, TIM3 lub TIM15.

Przykładowy program zapewnia obsługę popularnego cyfrowego czujnika temperatury typu DS18B20. Program cyklicznie odczytuje identyfikator układu zawarty w jego pamięci stałej, inicjuje pomiar temperatury i odbiera wynik pomiaru. Dane odczytane z układu DS18B20 są wyświetlane na wyświetlaczu ciekłokrystalicznym. Czujnik temperatury może pracować z oddzielnym zasilaniem lub może być zasilany przez linię danych.

Program został napisany dla mikrokontrolera STM32F030, umieszczonego na opisanej wcześniej płytce eksperymentalnej STM32F030exp1. Do obsługi wyświetlacza LCD wykorzystano moduł opisany w jednym z poprzednich odcinków serii nt. programowania STM32F0, zapewniający nieblokujące inicjowanie wyświetlacza i aktualizację jego zawartości. Program można łatwo zaadaptować dla dowolnego innego mikrokontrolera serii STM32F i innej płytki, modyfikując inicjowanie portów i peryferiali i zmieniając definicje zasobów związanych z obsługą interfejsu 1-Wire, umieszczone w pliku nagłówkowym definiującym zasoby sprzętowe.

Inicjowanie mikrokontrolera

W celu łatwej adaptacji kodu obsługi 1-Wire dla dowolnego timera, w pliku nagłówkowym definiującym zasoby sprzętowe płytki zdefiniowano trzy symbole, odpowiadające nazwie struktury timera, numerowi jego przerwania i nazwie procedury obsługi przerwania oraz symbole definiujące port i linię używane przez interfejs 1-Wire.

Ponieważ obsługa interfejsu 1-Wire wymaga szybkiej reakcji na przerwania, z czasami odpowiedzi na poziomie 1..2 µs, mikrokontroler musi pracować z odpowiednio dużą częstotliwością. W przykładowym programie użyto wewnętrznego generatora RC i modułu PLL, zaprogramowanego na częstotliwość 48 MHz. W programowaniu PLL (**listing 2**) biorą udział dwie procedury: *SystemInit()* i *main()* – technika ta została wyjaśniona w jednym

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 32086, pass: sqz8sawb

z poprzednich odcinków serii. Procedura *SystemInit* i tablica inicjowania peryferiali zostały umieszczone w plik ow-ts-init.c. Do programowania timera została również zdefiniowana stała PRE_1us – wartość preskalera zegara timera odpowiadająca okresowi 1 μs.

Inicjowanie mikrokontrolera obejmuje:

- Włączenie timera i innych peryferiali w rejestrach sterowania zegarami bloku RCC.
- Zaprogramowanie portów GPIO do obsługi LCD i 1-Wire.

Listing 1. Plik definicji zasobów F030exp1.h

```
/*
    F030exp1 board defs
    gbm, 05'14
*/

#include "stm32f0yy.h"
#include "stm32futil.h"
//===================================
#define DCC_IN_PORT                 GPIOA
#define DCC_IN_BIT          5
#define SRV1_PWM   TIM14->CCR1      // PA4
#define SRV2_PWM   TIM3->CCR1       // PA6
//===================================
// OneWire
#define OWTIM      TIM3
#define OWTIM_IRQnTIM3_IRQn
#define OWTIM_IRQHandler   TIM3_IRQHandler
#define PRE_1us    (SYSCLK_FREQ / 1000000)
#define OW_PORT    GPIOB
#define OW_PIN     1
//===================================
#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
#ifdef PLL_48
    #define PLLMUL12
    // 48 MHz SYSCLK
    #define E_L_DELAY   __NOP();__NOP();__NOP();__NOP();__NOP();__NOP();__NOP();__NOP()
    #define E_H_DELAY   __NOP();__NOP();__NOP();__NOP();__NOP();__NOP();__NOP();__NOP();__NOP();__NOP();__NOP()
#else
    // 8 MHz SYSCLK
    #define E_L_DELAY
    #define E_H_DELAY __NOP();__NOP();__NOP()
#endif
```

Listing 2. Inicjowanie mikrokontrolera (plik ow-ts-init.c)

```
/*
    STM32F0 tutorial - OneWire
    Non-blocking DS18B20 ROM & temperature readout
    Initialization
    gbm, 05'2014
*/

#include "f030exp1.h"
#include "lcd.h"
//===========================
#define SYSTICK_FREQ       1600
//===========================
void SystemInit(void)
{
    FLASH->ACR = FLASH_ACR_PRFTBE;  // enable prefetch
    RCC->CFGR = RCC_CFGR_PLLMULV(PLLMUL);  // set PLL multiplier to 4 for 16 MHz clock
    RCC->CR |= RCC_CR_PLLON;       // turn PLL on
}
//===========================
const struct init_entry_ init_table[] =
{
    {&RCC->CFGR, RCC_CFGR_PLLMULV(PLLMUL) | RCC_CFGR_SW_PLL},        // switch to PLL clock
    // enable peripherals
    {&RCC->APB1ENR, RCC_APB1ENR_TIM3EN},
    {&RCC->AHBENR, RCC_AHBENR_GPIOAEN | RCC_AHBENR_GPIOBEN | RCC_AHBENR_RSTVAL},
    // port setup
    {&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
    {&OW_PORT->BSRR, 1 << OW_PIN},
    {&GPIOB->MODER, BF2(OW_PIN, GPIO_MODER_OUT)},  // UART pins as AF
    // OneWire timer setup
    {(__IO32p)&OWTIM->PSC, PRE_1us - 1},
    {(__IO32p)&OWTIM->DIER, TIM_DIER_CC2IE | TIM_DIER_CC1IE | TIM_DIER_UIE},  // enable update interrupt
    //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 << OWTIM_IRQn},  // enable interrupts
    {&SCB->SCR, SCB_SCR_SLEEPONEXIT_Msk},  // sleep while not in handler
    {0, 0}
};
```

Listing 3. Przykładowy program do obsługi 1-Wire z wykorzystaniem przerwań

```
/*
 * owcore.c
 * STM32F OneWire interface using any timer w/ 2 CCRs
 * gbm 05'14, inspired by James Harwood's code for
LPC1xxx
 */

#include "board.h"
#include "onewire.h"

volatile _Bool ow_busy, ow_ok;

// GPIO helper macros
#define OW_LOW     (OW_PORT->BRR = (1 << OW_PIN))
#define OW_HIGH    (OW_PORT->BSRR = (1 << OW_PIN))
#define OW_OD      (OW_PORT->OTYPER |= (1 << OW_PIN)) //
open drain
#define OW_PP      (OW_PORT->OTYPER &= ~(1 << OW_PIN))
// strong drive

#define OW_STARTTIM        (TIM_CR1_OPM | TIM_CR1_CEN)

enum OWstate_ {
    OWS_IDLE, OWS_RESET,
    OWS_WRITE, OWS_READ,
    OWS_NOTPRESENT
};

static enum OWstate_ ow_state, ow_next_state;
static const uint8_t *wr_ptr;
static uint8_t *rd_ptr;
static uint8_t wr_len, rd_len;

// OneWire timing in us
#define T_A   6
#define T_B   64
#define T_C   60
#define T_D   10
#define T_E   9
#define T_F   55
#define T_G   0
#define T_H   480
#define T_I   70
#define T_J   410
#define T_BIT 70

static void ow_start(void)
{
    OWTIM->SR = ~(TIM_SR_CC2IF | TIM_SR_CC1IF | TIM_SR_
UIF);  // clear interrupt flags
    OW_LOW; // drive low
    OWTIM->CR1 = OW_STARTTIM; // start timer
}

static void init_reset(void)
{
    // fire reset pulse
    OW_OD;
    ow_state = OWS_RESET;
    OWTIM->CCR1 = T_H; // reset drive duration
    OWTIM->CCR2 = T_H + T_I; // presence pulse detection
    OWTIM->ARR = T_H + T_I + T_J - 1; // reset finished
    ow_start();
}

static void init_write(void)
{
    OWTIM->CCR1 = T_A; // end of short write pulse
    OWTIM->CCR2 = T_C; // end of long write pulse
    OWTIM->ARR = T_BIT - 1; // end of write cycle
    ow_start();
}

static void init_read(void)
{
    OWTIM->CCR2 = T_A + T_E; // sample time
    ow_start();
}

// OW timer interrupt
void OWTIM_IRQHandler(void)
{
    static uint8_t curr_byte;
    static uint8_t bit_pos;
    uint32_t ir = OWTIM->SR & (TIM_SR_CC2IF | TIM_SR_
CC1IF | TIM_SR_UIF);
    uint32_t ow_sense = (OW_PORT->IDR >> OW_PIN) & 1;
    OWTIM->SR = ~ir;  // clear interrupts
    switch (ow_state) {
        case OWS_RESET:
            switch (ir)
            {
                case TIM_SR_CC1IF:  // release
                    OW_HIGH;
                    break;
                case TIM_SR_CC2IF:  // sense
                    if (ow_sense) ow_next_state = OWS_
NOTPRESENT;
                    break;
                case TIM_SR_UIF:  // timer stopped
```

- Częściowe zaprogramowanie timera dla 1-Wire.
- Konfigurację priorytetów wywłaszczania przerwań.
- Ustawienie timera SysTick na zgłaszanie przerwań z częstotliwością 1600 Hz, potrzebną do obsługi LCD.

Początkowe zainicjowanie timera sprowadza się do ustawienia preskalera tak, aby przebieg wejściowy miał okres 1 μs oraz włączenia źródeł przerwań – na końcu okresu i przy osiągnięciu wartości rejestrów CCR1 i CCR2. Rejestr CCR1 będzie zawierał czas uaktywnienia linii danych przez mikrokontroler na początku cyklu. Rejestr CCR2 przy zapisie bitu służy do odmierzenia dodatkowego czasu aktywności linii dla bitu o wartości x, a w cyklach odczytu i inicjowania – do określenia momentu próbkowania stanu linii danych. Rejestr ARR zawiera czas trwania całego cyklu transmisji bitu.

Poprawna obsługa 1-Wire przy użyciu przerwań wymaga zagwarantowania, że nie wystąpią opóźnienia obsługi przerwania timera wynikające z obsługi innych wyjątków. Dlatego niezbędne jest obniżenie priorytetów wszystkich innych używanych w systemie przerwań z pozostawieniem przerwania timera używanego do obsługi 1-Wire jako jedynego o najwyższym priorytecie, w celu umożliwienia wywłaszczania innych przerwań przez przerwanie timera. W programie demonstracyjnym jedynym innym przerwaniem jest przerwanie timera SysTick, którego priorytet ustala się w rejestrze SCB->SHP[1]. Zostaje on obniżony z domyślnej wartości 0 do wartości 0x80 (w rdzeniach Cortex-M0 o priorytecie wyjątku decyduje wartość dwóch najbardziej znaczących bitów 8-bitowego pola priorytetu, a mniejsza wartość odpowiada wyższemu priorytetowi wywłaszczania).

Sterowanie linią danych 1-Wire

Głównym modułem programu zawierającym niskopoziomową obsługę interfejsu 1-Wire, jest plik *owcore.c*, który zawiera procedurę obsługi przerwania timera sterującego transmisją i potrzebne procedury pomocnicze.

Linia danych 1-Wire jest linią typu „otwarty dren". Jest ona zewnętrznie podciągnięta do dodatniego bieguna zasilania rezystorem o wartości 3,3…4,7 kΩ. W przypadku zasilania układu DS18B20 z linii danych jest jednak wymagane dodatkowe, aktywne wysterowanie linii w stan wysoki na czas pomiaru temperatury. Wysterowanie takie nie przeszkadza w poprawnej pracy układów z interfejsem 1-Wire w czasie, gdy nie są transmitowane dane. Przyjęto, że linia danych będzie ustawiana w tryb z „otwartym drenem" przy rozpoczynaniu transmisji danych, a po zakończeniu zapisu, po którym nie następuje odczyt (czyli np. po wydaniu polecenia pomiaru temperatury), będzie ona przełączana w tryb „push-pull" z aktywnym stanem wysokim. Dla ułatwienia zapisu operacji na linii danych w module owcore.c zdefiniowano następujące makra:

- OW_LOW – wyzerowanie linii.
- OW_HIGH – ustawienie linii.
- OW_OD – przełączenie linii w tryb „otwarty dren".
- OW_PP – przełączenie linii w tryb „push-pull".

Podczas początkowego inicjowania mikrokontrolera linia zostaje ustawiona do pracy w trybie „push-pull" na poziomie wysokim.

Obsługa przerwań timera 1-Wire

Timer sterujący transmisją na szynie 1-Wire jest włączany tylko na czas transmisji. Jest on zaprogramowany w tryb

jednokrotny, a po każdym uruchomieniu odmierza trzy odcinki czasu, których długości zależą od realizowanej fazy protokołu 1-Wire. Po każdym okresie transmisji bitu, jeżeli transmisja ma być kontynuowana, timer zostaje ponownie uruchomiony.

Obsługa przerwań została zrealizowana w konwencji automatu (**listing 3**). Początkowym stanem przy rozpoczęciu transmisji jest stan OWS_RESET, w którym jest generowany impuls inicjowania układów, a następnie sprawdzana ich odpowiedź w celu stwierdzenia, czy na szynie znajduje się jakiś układ. Po zakończeniu cyklu inicjowania następuje:

- Jeżeli nie wykryto odpowiedzi – zakończenie transakcji z błędem.
- Jeżeli wykryto odpowiedź – przejście do stanu zapisu danych – OWS_WRITE.

W stanie OWS_WRITE następuje transmisja bloku bajtów o długości określonej przez wartość zmiennej *wr_len*, rozpoczynającego się od adresu ustawionego w zmiennej *wr_ptr*. W stanie tym przerwania porównań z rejestrami CCRx timeera służą do zdeaktywowania linii danych, po czasie CCR1 przy transmisji zera i po czasie CCR2 przy transmisji jedynki. Okres timera odmierza okresy cykli transmisji bitów.

Po zakończeniu nadawania, jeżeli licznik odbioru rd_len ma wartość różną od zera, następuje przejście do stanu odbioru OWS_READ. W przeciwnym razie transmisja zostaje zakończona (stan OWS_IDLE).

W stanie OWS_READ dane są odbierane z szyny 1-Wire. Rejestr CCR1 timera określa czas impulsu rozpoczynającego cykl transmisji bitu,a rejestr CCR2 – moment próbkowania odbieranego bitu. Odbierane dane są zapisywane kolejno do bufora, którego adres przechowuje zmienna rd_ptr. Po odebraniu bloku następuje zakończenie transmisji i przejście do stanu OWS_IDLE. Opisany automat może zostać łatwo rozbudowany o kolejne stany, potrzebne do identyfikacji wielu układów dołączonych do szyny.

Interfejs pomiędzy niskopoziomową obsługą 1-Wire i innymi składnikami oprogramowania zapewniają publiczne zmienne typu _Bool:

- Zmienna *ow_busy* sygnalizująca trwanie transmisji.
- Zmienna *ow_ok* sygnalizująca poprawność wykonania zakończonej operacji.

Do inicjowania transakcji służy procedura owstart_wr_rd(), do której przekazywane są cztery argumenty – adresy i długości buforów danych, które mają być nadane i odebrane przez interfejs 1-Wire. Jeżeli transakcja składa się tylko z fazy zapisu – długość danych odbieranych ma wartość 0. Procedura inicjująca transakcję ustawia zmienną *ow_busy*, a zmienną *ow_ok* – zeruje. Zmienna *ow_busy* jest zerowana przez procedurę obsługi przerwania timera 1-Wire przy zakończeniu transakcji. Jednocześnie zmienna *ow_ok* ma nadawany poziom niski lub wysoki, stosownie do statusu zakończenia transakcji.

W module *owcore.c* zdefiniowano ponadto kilka procedur pomocniczych, które grupują akcje związane z inicjowaniem transakcji i zmianami faz protokołu. Procedury te ustawiają stan linii danych 1-Wire, programują odcinki czasu w rejestrach timera i uruchamiają timer.

Procedury użytkowe 1-Wire

Moduł onewire.c zawiera zestaw procedur realizujących funkcjonalność interfejsu 1-Wire potrzebną w danej aplikacji

Listing 3. c.d.

```
                    bit_pos = 0;
                    curr_byte = *wr_ptr ++;
                    ow_state = ow_next_state;
                    switch (ow_state) {
                        case OWS_WRITE:
                            init_write();
                            break;
                        default:
                            ow_busy = 0;
                            return;
                    }
                    break;
                }
                break; // reset
            case OWS_WRITE:
                switch (ir) {
                    case TIM_SR_CC1IF: // release if 1
                       if (((curr_byte >> bit_pos) & 1) == 0)
                        break;
                    case TIM_SR_CC2IF: // release
                        OW_HIGH;
                        break;
                    case TIM_SR_UIF: // end of bit write
                        if (++ bit_pos == 8)
                        {
                            // end of byte
                            bit_pos = 0;
                            if (-- wr_len == 0)
                            {
                                // end of write
                                if (rd_len)
                                {
                                    init_read();
                                    ow_state = OWS_READ;
                                }
                            else
                            {
                                // finish
                                ow_state = OWS_IDLE;
                                ow_ok = 1;
                                ow_busy = 0;
                                OW_PP;  // strong drive
                            }
                            return;
                        }
                        else curr_byte = *wr_ptr ++;
                        //next byte
                    }
                    OW_LOW;
                    OWTIM->CR1 = OW_STARTTIM; // start timer
                    break;
                }
                break; // write
            case OWS_READ:
                switch (ir) {
                    case TIM_SR_CC1IF: // release
                        OW_HIGH;
                        break;
                    case TIM_SR_CC2IF: // sample
                        curr_byte >>= 1;
                        if (ow_sense) curr_byte |= 0x80;
                        break;
                    case TIM_SR_UIF:
                    // end of bit read cycle
                        if (++ bit_pos == 8)
                        {
                            // end of byte
                            bit_pos = 0;
                            *rd_ptr ++ = curr_byte;
                            if (-- rd_len == 0)
                            {
                                // end of data
                                ow_state = OWS_IDLE;
                                ow_ok = 1;
                                ow_busy = 0;
                                return;
                            }
                        }
                        OW_LOW;  // next bit
                        OWTIM->CR1 = OW_STARTTIM; // start
timer
                        break;
                }
            break; // read
                default:
                break;
    }
}

// Start OW transaction - write optionally folowed by
read
void owstart_wr_rd(const uint8_t *wrbuf, const uint32_t
wrlen, uint8_t *rdbuf, uint32_t rdlen)
{
    wr_ptr = wrbuf;
    wr_len = wrlen;
    rd_ptr = rdbuf;
    rd_len = rdlen;
    ow_next_state = OWS_WRITE;
    ow_busy = 1;
    ow_ok = 0;
    init_reset();
}
```

Listing 4. Procedury do obsługi 1-Wire

```
/*
 * onewire.c
 * STM32F OneWire interface utility functions
 * gbm 05'14
 */

#include "onewire.h"

void ow_convert1(void)
{
    static const uint8_t wr_buf[] = {CMD_SKIP, CMD_CONV};
    owstart_wr_rd(wr_buf, 2, 0, 0);
}

void ow_readrom(uint8_t *buf)
{
    static const uint8_t wr_buf[] = {CMD_ROM};
    owstart_wr_rd(wr_buf, 1, buf, 8);
}

void ow_readsp1(uint8_t *buf)
{
    static const uint8_t wr_buf[] = {CMD_SKIP, CMD_READ};
    owstart_wr_rd(wr_buf, 2, buf, 9);
}
```

Listing 5. Obsługa czujnika DS18B20

```
/*
    STM32F0 tutorial - OneWire
    Non-blocking DS18B20 ROM & temperature readout
    Main program file
    gbm, 05'2014
*/

#include "board.h"
#include "lcd.h"
#include "onewire.h"
#include "crc8.h"
#include <string.h>
//===========================================
#define DEG_SIGN  239  // LCD degree sign code
//===========================================
extern const struct init_entry_ init_table[];
//===========================================
int main(void)
{
    while (!(RCC->CR & RCC_CR_PLLRDY));  // wait for PLL lock
    writeregs(init_table);
    __WFI();  // go to sleep
}
//===========================================
 static inline uint8_t hexdigit(uint32_t v)
{
    return "0123456789abcdef"[v & 0xf];
}
//===========================================
// 1600 Hz interrupt
void SysTick_Handler(void)
{
    static uint8_t tdiv = 0;
    lcdhandler();
    if ((++ tdiv & 15) == 0)
    {
        // 10 ms
        if (!ow_busy)
        {
            static enum {S_START, S_ROM, S_CONVERT, S_READ} state = S_START;
            static uint8_t owbuf[9];
            static uint8_t conv_timer;
            int i;
            switch (state)
            {
                case S_START:
                    ow_readrom(owbuf);
                    state = S_ROM;
                    break;
                case S_ROM:
                    if (ow_ok && crc8(owbuf, 8) == 0)
                    {
                        for (i = 0; i < 8; i ++)
                        {
                            lcd.screen[0][i * 2] = hexdigit(owbuf[i] >> 4);
                            lcd.screen[0][i * 2 + 1] = hexdigit(owbuf[i]);
                        }
                        ow_convert1();  // start temperature conversion
                        conv_timer = 76;
                        state = S_CONVERT;
                    }
                    else
                    {
                        memcpy(lcd.screen[0], "No OW device     ", 16);
                        state = S_START;
                    }
                    lcd.req.upd[0] = 1;
                    break;
                case S_CONVERT:
                    if (-- conv_timer == 0)
                    {
                        ow_readsp1(owbuf);
                        state = S_READ;
```

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 32086, pass: sqz8sawb

Listing 5. c.d.

```
                    }
                    break;
                case S_READ:
                    if (ow_ok && crc8(owbuf, 9) == 0)
                    {
                        uint8_t tsign = '+';
                        int16_t t = owbuf[0] | (owbuf[1] << 8);
                        lcd.screen[1][14] = DEG_SIGN;
                        lcd.screen[1][15] = 'C';
                        if (t < 0)
                        {
                            tsign = '-';
                            t = -t;
                        }
                        lcd.screen[1][13] = (t & 0xf) * 10 / 16 + '0';
                        lcd.screen[1][12] = '.';
                        t >>= 4;
                        i = 11;
                        do {
                            lcd.screen[1][i --] = t % 10 + '0';
                            t /= 10;
                        } while (t);
                        lcd.screen[1][i --] = tsign;
                        while (i >= 8) lcd.screen[1][i --] = ' ';
                    }
                    else memcpy(lcd.screen[1] + 8, " Error! "' 8);
                state = S_START;
                lcd.req.upd[1] = 1;
            } // switch
        }
    }
}
```

– w naszym przypadku są to trzy procedury potrzebne do obsługi pojedynczego czujnika temperatury (**listing 4**):

- *ow_readrom()* inicjuje odczyt zawartości pamięci stałej układu.
- *ow_convert1()* inicjuje pomiar temperatury.
- *ow_readsp1()* inicjuje odczyt rejestrów zmiennych układu DS18B20, w tym wartości temperatury.

Procedury te są nieblokujące i służą jedynie do zainicjowania transakcji. Zakończenie transakcji jest sygnalizowane przez wyzerowanie zmiennej ow_busy.

Przerwanie SysTick

Przerwanie timera systemowego służy do realizacji głównej funkcjonalności programu. Jest ono zgłaszane z częstotliwością 1600 Hz. W każdym przerwaniu następuje wywołanie obsługi wyświetlacza LCD, a przy co szesnastym (z częstotliwością 100 Hz) jest sprawdzany stan zmiennej *ow_busy;* gdy nie trwa transakcja 1-Wire – następuje uruchomienie automatu sterującego obsługą czujnika temperatury.

Współpraca z układem DS18B20

Obsługa czujnika temperatury została również zrealizowana w konwencji automatu (**listing 5**), uruchamianego po zakończeniu transakcji 1-Wire i działającego w czterech stanach:

- Stan S_START inicjuje próbę odczytu identyfikatora układu, po czym następuje przejście do stanu S_ROM.
- W stanie S_ROM, jeżeli nie wykryto układu, następuje powrót do stanu S_START. Jeżeli dane zostały odczytane bez błędu – są one wyświetlane na wyświetlaczu, inicjowana jest transmisja polecenia pomiaru temperatury, uruchamiany jest programowy timer odmierzający czas konwersji i następuje przejście do stanu S_CONVERT.
- W stanie S_CONVERT po wyzerowaniu timera oczekiwania na zakończenie pomiaru jest inicjowany odczyt wyniku pomiaru i następuje przejście do stanu S_READ.
- W stanie S_READ, o ile nastąpił poprawny odbiór danych, zmierzona temperatura jest wyświetlana w dolnej linii wyświetlacza, po czym następuje przejście do stanu S_START.

Do weryfikacji poprawności danych odczytanych z czujnika służy procedura obliczająca kod korekcyjny CRC8. Jego wartość powinna wynosić 0.

Temperatura jest odczytywana z czujnika w kodzie U2, Wartość temperatury jest zapisana w stopniach Celsjusza w postaci stałopozycyjnej, z czterema bitami części ułamkowej. W celu wyświetlenia temperatury najpierw jest ona sprowadzana do postaci znak-moduł, następnie binarna część ułamkowa jesz zamieniana na reprezentującą ją jedną cyfrę dziesiętną, po czym zachodzi standardowa konwersja części całkowitej do postaci ciągu cyfr dziesiętnych. Znaki reprezentujące temperaturę są kolejno wpisywane do pamięci zawartości wyświetlacza.

Oprogramowanie umożliwia np. wygodne testowanie układów DS18B20 – w czasie pracy urządzenia można wyjąć układ z podstawki i włożyć inny, po czym można odczytać na wyświetlaczu jego identyfikator i aktualną wartość temperatury. Odczyt i wyświetlenie identyfikatora układu następuje praktycznie natychmiast po umieszczeniu czujnika w podstawce, a kolejne pomiary temperatury następują z okresem ok. 0,8 sekundy, wynikającym z parametrów czasowych czujnika.

Grzegorz Mazur
gbm@ii.pw.edu.pl

Bibliografia:

1. *[DS18B20] DS18B20 Programmable Resolution 1-Wire Digital Thermometer, 2008, Maxim Integrated Products, Inc.*
2. *[AN126] APPLICATION NOTE 126, 1-Wire Communication Through Software, May 30, 2002, Maxim Integrated Products, Inc.*
3. *[AN162] APPLICATION NOTE 162, Interfacing the DS18X20/DS1822 1-Wire® Temperature Sensor in a Microcontroller Environment, Mar 08, 2002, Maxim Integrated Products, Inc.*
4. *[APP214] TUTORIAL 214, Using a UART to Implement a 1-Wire Bus Master, Sep 10, 2002, Maxim Integrated Products, Inc.*