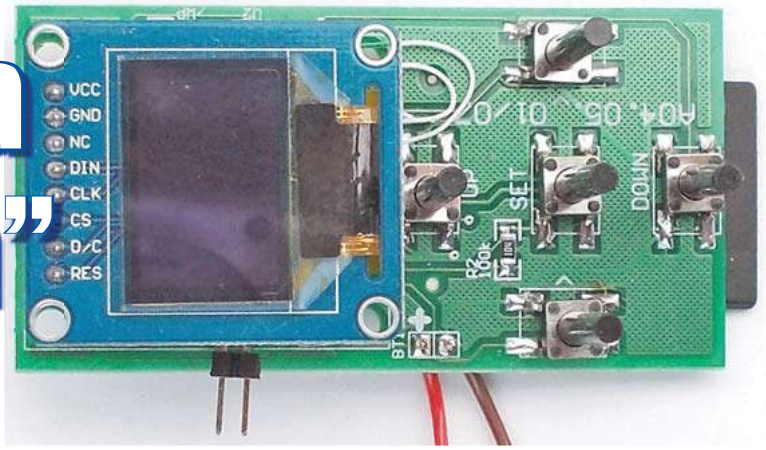
Elektroniczna „wielokostka”



Do czego służy kostka do gry, nie trzeba pisać. Kostka elektroniczna jest nowoczesną wersją tradycyjnej wykonanej z plastiku czy drewna. Tradycyjna mechaniczna kostka do gry jest prosta w budowie, tania, ale ma wady, m.in. potrafi potoczyć się w trudno dostępne miejsce. Kolejną jest hałas przez nią wytwarzany oraz fakt, że potrafi wpaść na planszę gry i przewrócić pionki. Wad tych pozbawiona jest kostka elektroniczna. W Internecie można znaleźć wiele konstrukcji, a wszystkie, jakie widziałem, wyświetlają wynik losowania na diodach LED ułożonych na wzór oczek na kostce. Prezentowana tutaj konstrukcja ma nowoczesny wyświetlacz OLED, co pozwala na wyświetlanie różnorodnych grafik. Ponadto może zastąpić cztery kostki o praktycznie dowolnej liczbie ścian.

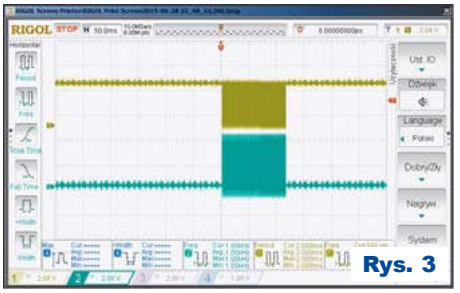
Grafiki przedstawiające poszczególne ścianki kostki, których może być nawet 250, zapisane są na karcie SD. Potrzebną grafikę łatwo stworzyć na komputerze, zapisać w popularnym standardzie BMP, po czym skopiować na kartę pamięci. Urządzenie pozwala na obsługę czterech kostek.

Opis układu

Schemat ideowy pokazany jest na **rysunku 1**. Układ zasilany jest z baterii. Sercem urządzenia jest mikrokontroler ARM STM32F103RBT8 taktowany częstotliwością 36MHz. Odczytuje on grafiki z karty SD umieszczonej w gnieździe J4. Do komunikacji wykorzystany jest interfejs SPI1, współdzielony z pamięcią DataFlash U2. Interfejs pracuje z częstotliwością 18MHz. Grafika jest wyświetlana na ***kolorowym wyświetlaczu OLED o rozdzielczości 96×64*** pikselе. Komunikację z wyświetlaczem realizuje interfejs SPI2, pracujący z częstotliwością 9MHz, wspomagany przez układ DMA. Do obsługi kostki wykorzystywanych jest pięć klawiszy. Jeden z nich, oznaczony „SET”, jest włączony inaczej niż pozostałe, ponieważ służy do wybudzania mikrokontrolera z uśpienia. J1 służy do zaprogramowania mikrokontrolera, J5 nie jest używany.

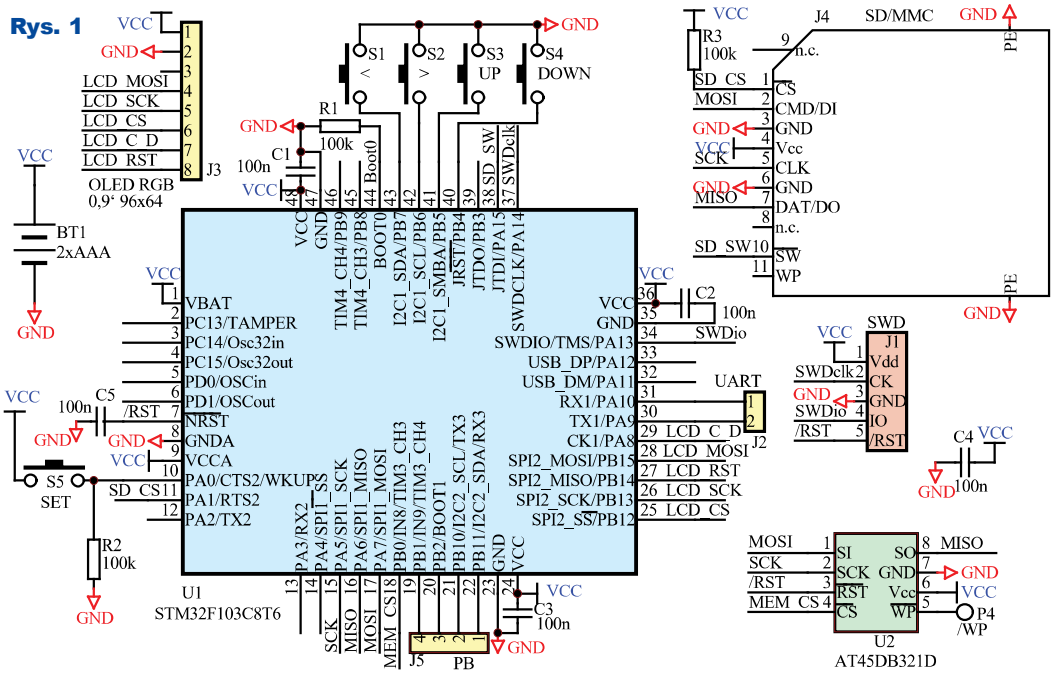
Program dla mikrokontrolera (dostępny w Elportalu) może wydawać się prosty, ale tak nie jest. Został napisany w KEIL µV 5, zajmuje ponad 32kB FLASH, ale sekcja kodu nie przekracza 26kB, więc można go skompilować darmową wersją programu. Zużycie RAM jest duże i wynosi 18kB, ale to głównie za sprawą dużego bufora na dane wyświetlacza. Przy pisaniu softu wspomagano się programem CubeMX. Program w dużej mierze korzysta z HAL, a tylko nieliczne fragmenty operują bezpośrednio na rejestrach mikrokontrolera. Do obsługi wyświetlacza wykorzystałam biblioteki Arduino. Niestety, nie używały one bufora w pamięci RAM, a nawet sprzętowego kasowania zawartości ekranu. W konsekwencji sama transmisja obrazu, bez czytania go z karty SD (znajdował się w FLASH), zajmowała około 120ms (**rysunek 2**). Może wydawać się, że to niedługo, ale niestety widoczne jest rysowanie kolejnych linii. Użycie bufora w RAM skróciło ten czas do około 46ms (**rysunek 3**). Czas ten można byłoby jeszcze trochę skrócić. W tym celu wystarczy zdefiniować okno wyświetlania (9 bajtów danych), zaczynające się w punkcie 0,0 a kończące 96, 64, po czym wysłać 12288 bajtów (96×64×2). Operacja ta zajęłaby ok. 11ms. Nie zdecydowałem się na to rozwiązanie, ponieważ w czasie transmisji

Rys. 2



Rys. 3

Rys. 1



CPU nie robiłby nic innego (poza przerwaniami), tylko wpisywał daną do rejestru DR:

```
SPI1->DR = (uint16_t)data;
```

po czym czekał na wysłanie danej:

```
while ( (SPI1->SR & SPI_FLAG_TXE) == RESET );
```

Trzeba by jeszcze pamiętać o tym, aby po zakończeniu transmisji, przed ustawieniem linii „/CS" wyświetlacza w stan nieaktywny, poczekać na faktyczne wysłanie danej:

```
while ( SPI1->SR & SPI_FLAG_BSY );
```

Aby procesor mógł pracować w czasie transmisji danych do wyświetlacza, można byłoby użyć mechanizmu przerwań. Nie jest to dobre rozwiązanie przy dużych prędkościach transmisji czy wolnych CPU. W tym urządzeniu CPU jest taktowany z częstotliwością 30MHz, a ciągłe wchodzenie i wychodzenie w przerwanie niepotrzebnie by go obciążało. Problem rozwiązuje DMA znane choćby z popularnego niegdyś Z-80 czy 8080. Używanie DMA nie jest trudne, zwłaszcza gdy korzysta się z HAL dostarczonego przez STMicroelectronics oraz CubeMX. Fragment programu za to odpowiedzialny jest prosty:

```
__SSD1331_CS_CLR();
HAL_SPI_Transmit_DMA( &hspi2, bufor, liczba_danych );
```

Po zakończeniu transmisji trzeba zdezaktywować sygnał „/CS", skąd jednak wiadomo, że DMA zakończyło transmisję? DMA może informować o wysłaniu połowy lub wszystkich danych (w nowszych mikrokontrolerach dodatkowo ¼ i ¾ danych), wywołując przerwanie, wystarczy więc w przerwaniu zdezaktywować linię „/CS", co wyróżniono w **listingu 1** kolorem.

Listing 1

```
void DMA1_Channel5_IRQHandler(void)
{
  /* USER CODE BEGIN DMA1_Channel5_IRQn 0 */

     DMA_HandleTypeDef *hdma = &hdma_spi1_tx;
     uint32_t flag_it = hdma->DmaBaseAddress->ISR;
  uint32_t source_it = hdma->Instance->CCR;
  /* Transfer Complete Interrupt management ***********************************/
  if (((flag_it & (DMA_FLAG_TC1 << hdma->ChannelIndex)) != RESET) && ((source_it & DMA_IT_TC) != RESET)){
     __SSD1331_CS_CLR();
     }


  /* USER CODE END DMA1_Channel5_IRQn 0 */
  HAL_DMA_IRQHandler(&hdma_spi2_tx);
  /* USER CODE BEGIN DMA1_Channel5_IRQn 1 */

  /* USER CODE END DMA1_Channel5_IRQn 1 */
}
```

Czy aby na pewno zadziała to poprawnie?

Niestety nie. DMA uznaje transmisję za zakończoną, gdy wyśle ostatnią daną do rejestru DR układu SPI, a jak już wiemy, w tym czasie dana jest jeszcze transmitowana. Aby nie obciąć kilku ostatnich bitów, należy poczekać na skasowanie bitu SPI_FLAG_BSY w rejestrze SR. Ze względu na to, że w przerwaniu nie powinno się stosować pętli jak na wcześniejszym listingu, zamiast dezaktywować linię „/CS" wyświetlacza, należy uruchomić timer, który po czasie wymaganym do wysłania danych z układu SPI wywoła przerwanie, które zdezaktywuje linię „/CS". Rozwiązanie takie pozwoliło na wysłanie danych do wyświetlacza w czasie około 12ms (**rysunek 4**). Co ważne, w czasie transmisji przez układ SPI, CPU może pracować. Jego wydajność nieznacznie spada, bo DMA „kradnie" cykl dostępu do magistrali, gdy CPU chce pobrać daną z tego samego obszaru co DMA, na przykład z pamięci RAM. Kolejnym problemem było wybudzanie mikrokontrolera ze stanu głębokiego uśpienia (ST-BY). Tak jak uśpienie jest dość proste:

```
  ssd1331_off();     // Uśpienie LCD
__HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU);
  PWR->CSR |= PWR_CSR_EWUP;
  PWR->CR |= PWR_CR_CWUF; // clear the WUF flag after 2 clock cycles
  PWR->CR |= PWR_CR_PDDS; // Enter Standby mode when the CPU enters deepsleep
  SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk; // low-power mode = stop mode
  SCB->SCR |= SCB_SCR_SLEEPONEXIT_Msk; // reenter low-power mode after ISR
  __WFI(); // enter low-power mode
```

tak wybudzenie sprawiło kłopoty. Wynikały one z tego, że nie był kasowany bit **PWR** w rejestrze CR funkcją **__HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU)**.

Gdy wybudzanie działało poprawnie, to nie działał debugger (następowało rozłączenie). Aby tak się nie działo, należy w rejestrze DBGMCU->CR ustawić odpowiednie bity:

```
  DBGMCU->CR |= DBGMCU_CR_DBG_SLEEP | DBGMCU_CR_DBG_STOP |
DBGMCU_CR_DBG_STANDBY;    //
```

Jeśli używany jest watchdog, trzeba dezaktywować go po zatrzymaniu programu debuggerem. Realizuje to funkcja:

```
  __HAL_DBGMCU_FREEZE_IWDG();
```

Ten sam efekt można osiągnąć, ustawiając bity rejestru

```
DBGMCU -> APB1FZ |= DBGMCU_APB1_FZ_DBG_IWDG_STOP;
```

Aby urządzenie nie traciło ustawień po uśpieniu (ostatnio używana kostka, ostatni stan generatora pseudolosu), skorzystano z rejestrów BATTERY BACKUP DOMAIN. W mikrokontrolerach STM32, w trybie ST-BY, zawartość pamięci RAM jest tracona. Wykorzystanie backup domain jest proste, wystarczy

```
  RCC->APB1ENR |= RCC_APB1ENR_PWREN | RCC_APB1ENR_BKPEN;
  PWR->CR = PWR_CR_DBP;
  BKP->CR = BKP_CR_TPAL;// | BKP_CR_TPE;
  BKP->CR &= ~BKP_CR_TPE;    // Tamper (PC13) wyłączony
```

i już jest możliwe korzystanie z rejestrów BKP->DR1...BKP->DR15, jak i ze zmiennych 16-bit (int16_t , uint16_t, short, unsigned short). Nowsze mikrokontrolery mają więcej rejestrów, które są 32-bitowe, ponadto często dodatkowo kilka kB RAM podtrzymywanych baterią.

Generator preudolosu też sprawił problemy. Użycie „systemowej" funkcji „rand" dało nierówny rozkład dla gier z ponad sześcioma możliwościami (ruletka i kostka 12 ścian). Niestety, gra „orzeł, reszka" dawała przewidywalny ciąg: orzeł, orzeł, reszka, reszka, itd. Rozpocząłem poszukiwania innego generatora pseudolosu, wybór padł na prosty 15-bitowy

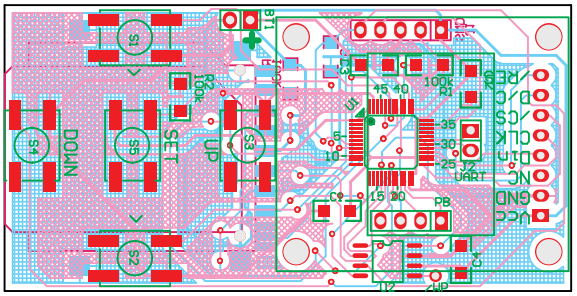
```
ziarnoRND = (a * ziarnoRND + c );
return ((( ziarnoRND >> 15) & 0xffff )
```

Rys. 4

Montaż i uruchomienie

Układ można zmontować na płytce drukowanej, której projekt pokazany jest na **rysunku 5**. Standardowo montujemy układ, zaczynając od elementów najmniejszych, a kończąc na największych. Fotografia wstępna oraz **fotografia 1** pokazują model. Układ nie wymaga uruchomienia.

Rys. 5



Z montowany prawidłowo ze sprawnych elementów powinien od razu pracować. Po włączeniu zasilania,

```
FRESULT fresult_read = f_read( &plik, bmpHeader, BMP_HEADER, &odczytanychBajtow );
 sprintf( str_buf, ANSI_PEN_BLUE"read header time="ANSI_PEN_YELLOW"%dms"CRLF""COL_TERM,
 (int)(TimSys - time) ); SendStringTerminal( str_buf );
 if ( bmpHeader[0] != ,B' || bmpHeader[1] != ,M' ) {
   SendStringTerminal( ANSI_PEN_RED"Plik nie jest BMP"CRLF""COL_TERM );
 }
 else if (  bmpHeader[14] != 40 ) {
   SendStringTerminal( ANSI_PEN_RED"Header <> 40"CRLF""COL_TERM );
   return BMP_HEADER;
 }
 else if (  bmpHeader[28] != 24 ) { //todo: akceptować 16-bit
   SendStringTerminal( ANSI_PEN_RED"Paleta <> 24-bit"CRLF""COL_TERM );
   return BMP_PALETA;
 }
 else if (  bmpHeader[30] != 0 ) {
   SendStringTerminal( ANSI_PEN_RED"Skompresowany"CRLF""COL_TERM );
   return BMP_KOMPRESJA;
 }
```

Listing 2

bez karty SD, wyświetlony zostanie komunikat z **rysunku 6.** Jeśli na karcie będą się znajdować wymagane pliki graficzne, ujrzymy ekran z **rysunku 7**. Przeskanowanie karty z 60 plikami trwa poniżej 5 sekund. W czasie skanowania widoczny jest ekran ze zmieniającymi się liczbami podobny do tego na **rysunku 8,** gdzie pierwsza liczba wskazuje numer kostki, druga numer grafiki. Pliki na karcie muszą być zapisane w formacie BMP. Wymiary obrazków powinny być równe 96×64 punkty, głębia koloru 24-bit, bez kompresji. Podczas skanowania analizowany jest nagłówek pliku. Obrazki dla prototypu tworzono programem Paint, grafiki pozyskane z Internetu poddano obróbce programem IrfanView. Na **listingu 2** widoczny jest fragment odpowiedzialny za wczytanie i analizę nagłówka pliku.

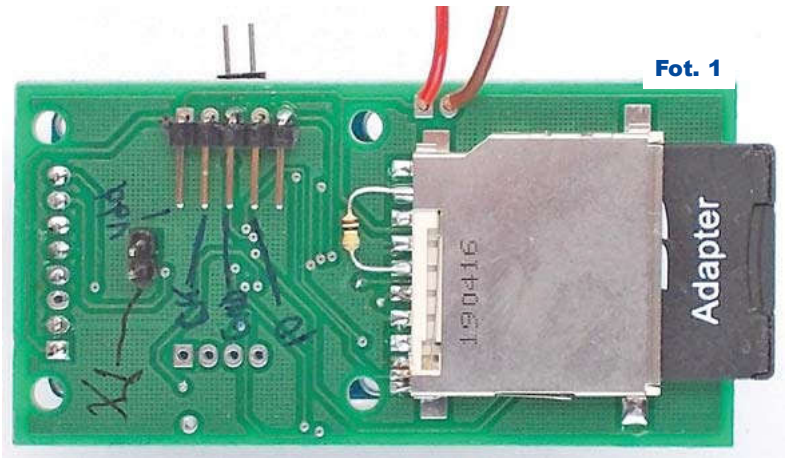
Komunikat o braku wymaganych plików może przyjąć nie tylko postać pokazaną na rysunku 6, ale także: „SD Error header”, „SD Error color” oraz „SD Error compression”.

Wymagania co do nazw plików są następujące: Nazwa pliku składa się z litery oraz liczby. Litera „a” oznacza pierwszą kostkę, „b” drugą, „c” trzecią, „d” czwartą. Skanowane będą tylko pliki zaczynające się na litery „a...d”. Kolejnym elementem nazwy jest numer obrazu, zaczynający się od cyfry „1”. Kolejne to „2”, 3” itd., do „9”, następnie „10”, „11” i tak do maksymalnie „250”. Nazwę pliku kończy „.bmp”. Można także zapisać na karcie plik z liczbą „01”. Obraz ten będzie wyświetlany w czasie losowania.

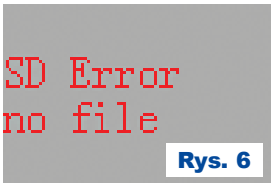
W materiałach dodatkowych można znaleźć przykładowe pliki graficzne. Dla tradycyjnej kostki pliki o nazwach *a1.bmp ... a6.bmp*. Dla gry „Orzeł, reszka”: *b01.bmp ... b2.bmp*. Dla kostki 12-ściennej: *c1.bmp ... c12.bmp*, dla ruletki: *d01.bmp ... d37.bmp* oraz inne, umieszczone w katalogach dysku. Grafiki mogą mieć inne wymiary; jeśli będą za duże, zostaną obcięte do 96×64, jeśli za małe, nie wypełnią całej powierzchni wyświetlacza. Realizuje to funkcja, pokazana na **listingu 3**.

W czasie skanowania karty SD na UART1 (złącze J2) wysyłane są informacje o znalezionych plikach, wykrytych błędach, jak pokazują **rysunki 9**, **10**. Nie należy przejmować się wszystkimi błędami, niektóre są rzeczą normalną, przykładowo nieodnalezienia obrazu numer 7 dla kostki o sześciu ścianach. Parametry transmisji 921600 8N1. Ze względu na wykorzystanie kodów ANSI zgodnych z VT100, do odczytu informacji z UART nie nadają się proste programy terminalowe, takie jak „Termite”, „Bray Terminal”. Należy użyć „Tera Term” czy „Putty”.

Na PCB przewidziano miejsce na pamięć DataFlash, ale w prototypie nie jest ona używana. Jej przeznaczeniem była możliwość skopiowania grafik z karty pamięci do DataFlash, skąd mogą one bardzo szybko zostać przekazane do wyświetlacza OLED. Czas odczytu grafik z karty SD jest stosunkowo długi, ale akceptowalny



Fot. 1



Rys. 6

Rys. 7

Rys. 8

```
uint32_t sizeX = bmpHeader[18] | bmpHeader[19] << 8 | bmpHeader[20] << 16  | bmpHeader[21] << 16;
 uint32_t sizeY = bmpHeader[22] | bmpHeader[23] << 8 | bmpHeader[24] << 16  | bmpHeader[25] << 16;
 uint32_t ofsRaw = bmpHeader[10] | bmpHeader[11] << 8 | bmpHeader[12] << 16  | bmpHeader[13] << 16;
 sprintf( str_buf, ANSI_PEN_CYAN"sizeX=%d sizeY=%d ofsRaw=%d"CRLF""COL_TERM, sizeX, sizeY, ofsRaw ); SendStringTerminal( str_buf );

 uint8_t static BmpLineBuf[SSD1331_WIDTH * 3];
 uint32_t h = sizeX; if (h > SSD1331_WIDTH ) h = SSD1331_WIDTH;
 uint32_t v = sizeY; if (v > SSD1331_HEIGHT ) v = SSD1331_HEIGHT;
 time = TimSys;
 for (u8 y = 0; y < v; y++) {
   f_lseek( &plik, ofsRaw + sizeX * 3 * y );   // Wskaźnik na kolejne linie pliku (szerokość * 24-bit * linia)
   //todo: jeśli liczba bajtów w linii nie jest podzielna przez 4 - dopełnić
   fresult_read = f_read( &plik, BmpLineBuf, h * 3, &odczytanychBajtow ); // Wczytanie jednej linii
   TRAP;
   for (u16 x = 0; x < h; x++) {
     u16 kolor = RGB(BmpLineBuf[2 + x * 3], BmpLineBuf[1 + x * 3], BmpLineBuf[0 + x * 3]);
     toBufSsd1331( x, SSD1331_HEIGHT - 1 - y, kolor ); // INFO -1-y bo BMP ma głupią organizacje ekranu
   }
 }
 sprintf( str_buf, ANSI_PEN_BLUE"read RAW time="ANSI_PEN_YELLOW"%dms"CRLF""COL_TERM, (int)(TimSys - time) ); SendStringTerminal( str_buf );
```

Listing 3

do wyświetlania statycznych obrazów. Jeśli miałyby być wyświetlane dodatkowe grafiki lub animacje (np. podczas losowania), to wykorzystanie DataFlash będzie konieczne.

W modelu pobór prądu w spoczynku, przy zasilaniu napięciem 3V, wynosił 23...100mA zależnie od wyświetlanej treści. Po przejściu w stan uśpienia (5 minut bezczynności) spadał do ok. 1mA. Autor sprawdził, co powoduje stosunkowo wysoki pobór prądu. Wyniki są następujące:

Mikrokontroler: 60µA
Wyświetlacz: 100µA
Karta SD: 840µA

Ze względu na stosunkowo duży pobór prądu konieczne jest zastosowanie wyłącznika zasilania. Może się wydawać, że 1mA to niedużo, ale to oznacza, że 2 akumulatory AA 600mAh wystarczą na 1200 godzin, a to jest tylko 50 dni.

Rys. 9

Rys. 10

Zakres napięć zasilających wynosi od 2,7 do 3,6V. Dolną granicę napięcia ogranicza karta SD i zależy ona od jej typu, wyświetlacz i mikrokontroler będą działać poprawnie przy 2,4V. Górną granicę wyznaczają wszystkie elementy półprzewodnikowe systemu. Rozwiązaniem byłaby przetwornica podwyższająco-obniżająca. Ze względu na pobór prądu nieprzekraczający 120mA przy 3,3V można użyć, prostego w aplikacji, układu MCP1252. Inna opcja to zrezygnowanie z karty SD na rzecz pamięci DataFlash. Karta potrzebna byłaby tylko na czas kopiowania danych z karty SD do pomięci i tylko na czas tej operacji należy zadbać o dobre zasilanie. Przy okazji zmniejszyłby się pobór prądu w uśpieniu, bo U2 w trybie Deep Power-down Current pobiera typowo 5uA, maksymalnie 10uA.

SaS
sas@elportal.pl

Wykaz elementów

R1,R2,R3 . . . . . 100k SMD 1206
C1,C2,C3,C4,C5 . . . . . 100nF SMD 1206
U1 . . . . . STM32F103C8T6
U2 . . . . . AT45DB321D. Nieużywana
S1...S5 . . . . . microswitch 5 x 7
BT1 . . . Koszyk na baterie AA lub AAA z wyłącznikiem
J1 . . . . . ZL201-05G
J2 . . . . . NS25-W2K. Tylko do debugowania
J3 . . . . . Wyświetlacz OLED RGB 0,95 96×64 KAMAMI ID: 561210
J4 . . . . . Gniazdo kart SD/MMC TME GSD090012SEU
J5 . . . . . Nieużywane

Ciąg dalszy ze strony 18

Obliczamy współczynnik kalibracyjny poprzez podzielenie wartości napięcia podanego przez napięcie zmierzone: 7.0V / 6,71 = 1,0432. Wartość tę wpisujemy do szkicu:

#define WSP_KALIBRACJI 1.0432

Po kalibracji należy wyłączyć komunikaty diagnostyczne poprzez następujące modyfikacje:

#define DEBUG_EN 0
– wyłączyć komunikaty diagnostyczne
#define KALIBRACJA_ADC_EN 0
– wyłączyć komunikaty kalibracyjne ADC

Kalibracja ADC jest koniczna tylko wtedy, gdy monitor baterii jest aktywowany (#define BAT_MON_EN 1). Po zakończonej konfiguracji należy wgrać do układu szkic z dobranymi zmiennymi konfiguracyjnymi. Od tej chwili układ jest gotowy do użytkowania. Aby układ działał prawidłowo, należy pamiętać o wyłączeniu komunikatów diagnostycznych przed ostatecznym programowaniem (#define DEBUG_EN 0). Ponadto akumulator zasilający (lub baterie) powinien dysponować wydajnością prądową o wartości około 300mA + max. prąd silnika.

Znaczenie diod sygnalizacyjnych opisano w **tabeli 1**.

Pomocą w realizacji, a także zachętą do budowy podobnego urządzenia będą fotografie zamieszczone w artykule.

Zbigniew Brzozowski
metalik@interia.pl

Tabela 1

Aktywna dioda	Komunikat
Migająca czerwona	Błąd inicjalizacji interfejsu sieciowego
Zielona i czerwona	Oczekiwanie na transmisję z nadajnika. Brak odbieranych ramek UDP
Zielona	Połączenie z nadajnikiem. Odbierane poprawne ramki z danymi
Czerwona	Przekroczenie progu rozładowania akumulatora (jeśli aktywowane)

Wykaz elementów

R1,R4 . . . . . 3,3kΩ SMD 1206
R2,R5,R11,R8 . . . . . 47kΩ SMD 1206
R3,R6 . . . . . 120Ω SMD 1206
R7 . . . . . 560kΩ lub 1MΩ SMD 1206
R9,R10 . . . . . 1kΩ SMD 1206
C1 . . . . . 470nF SMD 1206
C9,C8,C2,C5 . . . . . 100nF SMD 1206
C3,C4,C6,C7,C10 . . . . . .220-470uF/16V
Cx . 100nF – zamontować na wyprowadzeniach silnika
D1 . . . . . LED zielona SMD 1206
D2 . . . . . LED czerwona SMD 1206
D3,D4,D6,D5 . . . . . .1N5819
Q1,Q2 . . . . PMBT2907A, MMBT2907A SMD SOT-23
U1 . . . . . LM2940CT-5.0 – TO-220-3
U2 . . . . . ESP8266 D1 MINI
U3 . . . . . 74HCT125 DIP-14
U4 . . . . . L293 DIP-16
J1,J2,J3. . . . . Listwa gold-pin „łamana” raster 2,54 mm minimum 8 pinów
F1 . . . . . 3-5A, polimerowy
Socket gold-pin precyzyjny min. 16 pinów dla układu U2

Komplet podzespołów z płytką jest dostępny w Sklepie AVT jako zestaw AVT3259