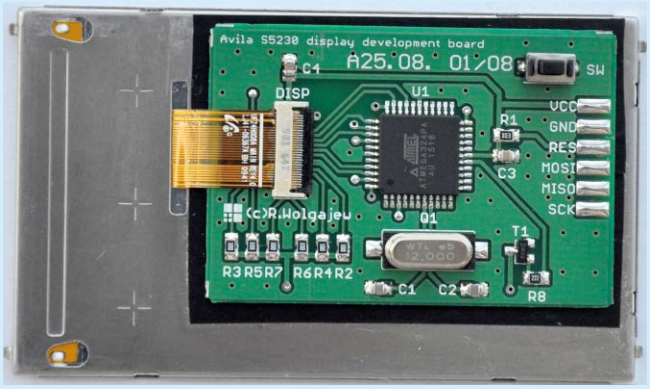
W ofercie AVT jest dostępna płytka ewaluacyjna umożliwiająca przyłączenie wyświetlacza opisywanego w artykule. Numer kitu AVT-5563.

Obsługa kolorowego wyświetlacza TFT z telefonu Samsung GT-S5230 Avila (2)

W poprzednim artykule opublikowanym w EP 12/2016 zaprezentowaliśmy podstawowe funkcje służące do inicjalizacji, wyświetlania punktów oraz rysowania podstawowych elementów – teraz zajmiemy się wyświetlaniem znaków oraz komunikatów.



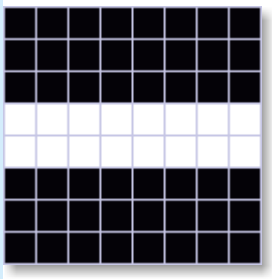
Przyszedł czas na obsługę ostatniego elementu interfejsów graficznych – czcionek ekranowych. Aby jednak umożliwić wygodną obsługę wielu czcionek ekranowych, konieczne było wprowadzenie nowego typu danych – jego definicję pokazano na **listingu 7**. Bazując na tak zdefiniowanej strukturze, wprowadzono funkcję, która korzystając ze zmiennej globalnej *static fontDescription CurrentFont* pozwala na ustawienie bieżącej czcionki ekranowej. Tę funkcję pokazano na **listingu 8**, natomiast na **listingu 9** pokazano funkcję umożliwiającą rysowanie znaków przy użyciu bieżącej czcionki ekranowej. Korzysta ona z argumentu *uint8_t Transparency*, od którego zależy, czy tło bieżącej czcionki ekranowej będzie określone wartością globalnej zmiennej *uint16_t Background* (w takim wypadku wartość argumentu *Transparency* jest równa *SOLID_TEXT*), czy też tło wyświetlanej czcionki będzie przeźroczyste (wartość argumentu *Transparency* równa *TRANSPARENT_TEXT*). Na bazie tej funkcji wprowadzono dwie nowe, umożliwiające wyświetlenie ciągu znaków zapisanego w pamięci RAM lub w pamięci programu (Flash) – pokazano je na **listingu 10**.

Do wygenerowania plików zawierających wzorce czcionek, oparte na czcionkach systemu Windows, polecam doskonały program PixeLab autorstwa Marcina Popławskiego. Szczegółowy opis programu zamieszczono w Elektronice Praktycznej 6/2015.

W tym miejscu przedstawię jeszcze jedną funkcję, której zadaniem jest wyświetlenie liczby typu *uint16_t* (tu z zakresu 0…9999) przy użyciu bieżącej czcionki ekranowej. Zadanie tego typu wydaje się być bolączką większości początkujących programistów, którzy w tym celu, zupełnie na siłę, starają się użyć standardowej funkcji *printf()*. Oczywiście można i tak, lecz płacimy za tę uniwersalność wysoką cenę wzrostem wielkości kodu wynikowego o dobre 1,5 kB. Co gorsze, wiele osób łączy wywołanie tejże funkcji z wyświetlaniem liczb zmiennoprzecinkowych, co jeszcze bardziej „demoluje" kod wynikowy.

Ze swojej, ponad 8-letniej praktyki, mogę powiedzieć jedno – poza nielicznymi wyjątkami dotyczącymi zwykle tematyki DSP, nie ma potrzeby używania liczb zmiennoprzecinkowych w aplikacjach wbudowanych, zwłaszcza tych wykorzystujących mikrokontrolery AVR. Po dogłębnej analizie okazuje się zwykle, że wyniki obliczeń jesteśmy w stanie przekazać w formacie całkowitym z odpowiednio przedstawionym przecinkiem. Właśnie do tego celu wykonano funkcję **TFTdrawInteger()**, którą pokazano na **listingu 11**. Nie jest to funkcja uniwersalna, bo taka wzorem *printf()* musiałaby być bardzo rozbudowana, jednak w większości wypadków zupełnie wystarczająca. Funkcja ta przyjmuje następujące argumenty wymagające dodatkowego komentarza:

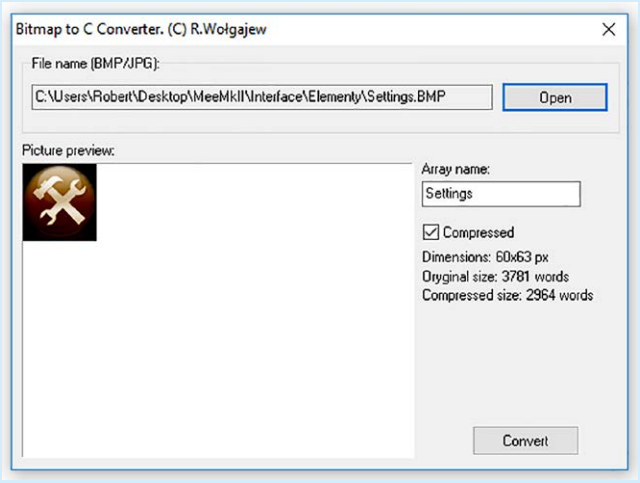
- *uint8_t Digits* – określa liczbę cyfr, przeznaczonych do wyświetlenia (1…4).
- *uint8_t Effects* – pozwala na specyfikację miejsca położenia przecinka (stałe DOT_POS3…DOT_POS1), przy czym samo wyświetlenie przecinka pozostaje zadaniem po stronie aplikacji użytkownika.



```
const uint16_t Uncompressed[] PROGMEM =
{
0x0808,  //Width (MSB)& Height (LSB)
0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0xFFFF,0xFFFF,0xFFFF,0xFFFF,0xFFFF,0xFFFF,0xFFFF,0xFFFF,
0xFFFF,0xFFFF,0xFFFF,0xFFFF,0xFFFF,0xFFFF,0xFFFF,0xFFFF,
0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000};

const uint16_t Compressed[] PROGMEM =
{
0x0808,  //Width (MSB)& Height (LSB)
0x0000,0x0000,0x0018,0xFFFF,0xFFFF,0x0010,0x0000,0x0000,
0x0018};
```

Rysunek 3. Wynik działania algorytmu kompresji obrazu



Rysunek 4 Wygląd aplikacji BMPConverter realizującej konwersję i kompresję pliku BMP do postaci tablicy języka C

Co więcej, zamieszczenie argumentu *Effects* pozwala na specyfikację niezbędnego odstępu pomiędzy znakami, które to mają być rozdzielone wspomnianym wcześniej przecinkiem (bity 5…0 tego argumentu). Poniżej przedstawiono przykłady użycia funkcji *TFTdrawInteger()* z rezultatem jej działania. Zakłada się, że przecinek zostanie wyświetlony na ekranie w wyniku działania aplikacji użytkownika. Sama funkcja zapewnia „wyłącznie" miejsce na wyświetlenie przecinka oraz zajmuje się „logiką" wyświetlania zera umieszczonego przed przecinkiem, jeśli występuje:

- **TFTdrawInteger(10, 100, 1234, 4, DOT_POS3|5);** – „1.234" (odstęp pomiędzy „1" a „2" powiększony o 5 pikseli).
- **TFTdrawInteger(10, 120, 234, 3, DOT_POS1|3);** – „23.4" (odstęp pomiędzy „3" a „4" powiększony o 3 piksele).
- **TFTdrawInteger(10, 120, 4, 2, DOT_POS1|4);** – „0.4" (odstęp pomiędzy „0" a „4" powiększony o 4 piksele).
- **TFTdrawInteger(10, 140, 4, 2, NO_DOTS);** – „ 4".

W tym miejscu pora na przysłowiową „wisienkę na torcie". Wcześniej, na listingu 6 pokazano funkcję umożliwiającą wyświetlenie na ekranie wyświetlacza TFT obrazka zapisanego w pamięci Flash mikrokontrolera. Jest to rozwiązanie nieskomplikowane, jednak obarczone pewną wadą. Jak

Tabela 2. Idea działania algorytmu kompresującego dane obrazków.

Liczba powtórzeń	Dane przed kompresją	Dane po kompresji
1	0xAAAA	0xAAAA
2	0xAAAA 0xAAAA	0xAAAA 0xAAAA 0x0002
3	0xAAAA 0xAAAA 0xAAAA	0xAAAA 0xAAAA 0x0003
4	0xAAAA 0xAAAA 0xAAAA 0xAAAA	0xAAAA 0xAAAA 0x0004
5	0xAAAA 0xAAAA 0xAAAA 0xAAAA 0xAAAA	0xAAAA 0xAAAA 0x0005
65535	65535 x 0xAAAA	0xAAAA 0xAAAA 0xFFFF

Listing 7. Definicja nowego typu danych odpowiedzialnego za przechowywanie parametrów bieżącej czcionki ekranowej.

```
typedef struct
{
    uint8_t Width;  //Current font widht (px)
    uint8_t Height;  //Current font height (px)
    uint8_t Interspace;  //Font interspace (px)
    uint8_t BytesPerChar;  //Bytes per char definition
    uint8_t FirstCharCode;  //First char ASCII code
    const uint8_t *Bitmap;  //Pointer to the font table
} fontDescription;
```

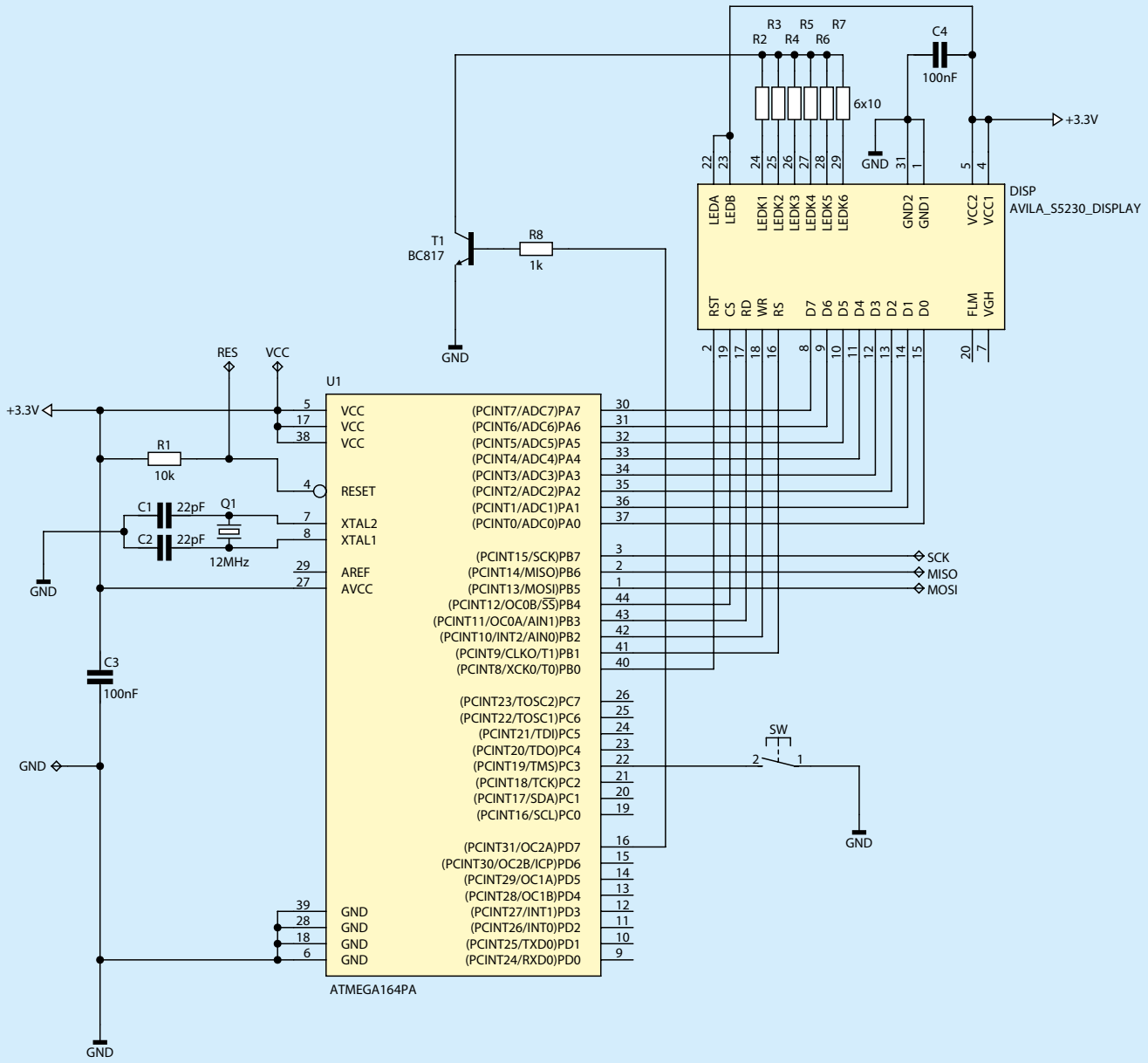
Listing 8. Funkcja odpowiedzialna za ustawienie bieżącej czcionki ekranowej.

```
void TFTsetFont(const fontDescription *Font)
{
    CurrentFont.Width = pgm_read_byte(&Font->Width);
    CurrentFont.Height = pgm_read_byte(&Font->Height);
    CurrentFont.Interspace = pgm_read_byte(&Font->Interspace);
    CurrentFont.BytesPerChar = pgm_read_byte(&Font->BytesPer-
Char);
    CurrentFont.FirstCharCode = pgm_read_byte(&Font->FirstChar-
Code);
    CurrentFont.Bitmap = (uint8_t*)pgm_read_word(&Font->Bitmap);
}
```

Listing 9. Funkcja odpowiedzialna za rysowanie znaków, przy użyciu bieżącej czcionki ekranowej

```
void TFTdrawChar(uint16_t X1, uint8_t Y1, char Character, const uint8_t Transparency)
{
    register uint8_t widthIndex, heightIndex, readByte, pixelsNr, i;
    const uint8_t *dataPointer;

    if(Transparency != TRANSPARENT_TEXT)
    {
        //We define display active area to simplify writing
        TFTsetActiveWindow(X1, Y1, X1+CurrentFont.Width-1, Y1+CurrentFont.Height-1);
        //We start memory writing
        writeCommand(CMD_MEMORY_WRITE);
    }
    //Now we calculate start address of the current character definition
    dataPointer = &CurrentFont.Bitmap[(CurrentFont.BytesPerChar*(Character-CurrentFont.FirstCharCode))];
    for(heightIndex = 0; heightIndex < CurrentFont.Height; heightIndex++)
    {
        for(widthIndex = 0; widthIndex < CurrentFont.Width; widthIndex += 8)
       {
            //We read character definition byte by byte
            readByte = pgm_read_byte(dataPointer++);
            //For fonts which width is not a multiple of 8 we need to calculate useful number of pixels to be sent
            pixelsNr = widthIndex+8 <= CurrentFont.Width ? 8 : CurrentFont.Width - widthIndex;
            for(i=0; i<pixelsNr; ++i)
            {
                if(Transparency == TRANSPARENT_TEXT)  //We check if the text background is transparent
                {
                    //We check the pixel presence
                    if(readByte & 0x80)
                    {
                        //We define display active area for one active pixel to simplify writing
                        TFTsetActiveWindow(X1+widthIndex+i, Y1+heightIndex, X1+widthIndex+i, Y1+heightIndex);
                        writeCommand(CMD_MEMORY_WRITE);
                        writeData(Colour >> 8); writeData(Colour & 0xFF);
                    }
                }
                else
                {
                    //Pixel color depends on the pixel presence
                    if(readByte & 0x80) {writeData(Colour >> 8); writeData(Colour & 0xFF);}
                    else {writeData(Background >> 8); writeData(Background & 0xFF);}
                }
                readByte<<=1;
            }
        }
    }
}
```



Rysunek 5. Schemat ideowy płytki ewaluacyjnej wyświetlacza GT-S5230

łatwo się domyślić, tak przygotowane obrazki zajmują sporo cennej pamięci programu, przez co jest niemożliwe korzystanie z wielu tego rodzaju elementów w procesorach o ograniczonej wielkości pamięci programu, jak dla przykładu, mikrokontrolery AVR. Jak temu zaradzić? Odpowiedź wydaje się dość prosta – należy zastosować metodę kompresji powtarzających się danych, co pozwoli

Listing 10 Funkcje umożliwiające wyświetlenie ciągu znaków z pamięci RAM, jak i pamięci programu (Flash)

```
void TFTdrawString(uint16_t X1, uint8_t Y1, char *String, const uint8_t Transparency)
{
    while(*String)
    {
        TFTdrawChar(X1, Y1, *String++, Transparency);
        X1 += CurrentFont.Width + CurrentFont.Interspace;
    }
}

void TFTdrawString_P(uint16_t X1, uint8_t Y1, const char *String, const uint8_t Transparency)
{
    register char Character;
    while((Character = pgm_read_byte(String++)))
    {
        TFTdrawChar(X1, Y1, Character, Transparency);
        X1 += CurrentFont.Width + CurrentFont.Interspace;
    }
}
```

Listing 11. Funkcja pozwalająca na wyświetlenie liczby typu Integer (zakres 0...999)

```
void TFTdrawInteger(uint16_t X1, uint8_t Y1, uint16_t Integer, uint8_t Digits, uint8_t Effects)
{
    register uint8_t Digit1, Digit2;
    register uint8_t dotPos = Effects & DOT_MASK;  //Bits: 7..6
    register uint8_t Interspace = Effects & INTERSPACE_MASK;  //Bits: 5..0
    register uint8_t fontStep = CurrentFont.Width + CurrentFont.Interspace;
    if((Digit1 = Integer/1000)) Integer -= Digit1*1000;
    if(Digits == 4)
    {
        if(Digit1 || (dotPos == DOT_POS3)) Digit1 += '0'; else Digit1 = ' ';
        TFTdrawChar(X1, Y1, Digit1, SOLID_TEXT);
        X1 += fontStep;
        if(dotPos == DOT_POS3) X1 += Interspace;
    }
    if((Digit2 = Integer/100)) Integer -= Digit2*100;
    if(Digits > 2)
    {
        if(Digit2 || (Digit1 != ' ') && Digits == 4) || (dotPos >= DOT_POS2)) Digit2 += '0'; else Digit2 = ' ';
        TFTdrawChar(X1, Y1, Digit2, SOLID_TEXT);
        X1 += fontStep;
        if(dotPos == DOT_POS2) X1 += Interspace;
    }
    Digit1 = Integer/10;
    if(Digits >1)
    {
        if(Digit1 || (Digit2 != ' ') && Digits > 2) || (dotPos >= DOT_POS1)) Digit1 += '0'; else Digit1 = ' ';
        TFTdrawChar(X1, Y1, Digit1, SOLID_TEXT);
        X1 += fontStep;
        if(dotPos == DOT_POS1) X1 += Interspace;
    }
    TFTdrawChar(X1, Y1, '0'+Integer%10, SOLID_TEXT);
}
```

Listing 12. Funkcja odpowiedzialna za wyświetlanie skompresowanych obrazków na ekranie wyświetlacza TFT void TFTdrawCompressedPicture(uint16_t X1, uint8_t Y1, const uint16_t *Picture)

```
{
    register uint16_t pixelsToSend, pixelA, pixelB;
    register uint8_t Width, Height;
    //We read the first word that holds the picture width and height (MSB and LSB)
    pixelA = pgm_read_word(Picture++);
    //We calculate the picture width and height
    Width = pixelA >> 8; Height = pixelA & 0xFF;
    //We calculate how many pixels we need to send
    pixelsToSend = Width * Height;
    //We define display active area to simplify writing
    TFTsetActiveWindow(X1, Y1, X1+Width-1, Y1+Height-1);
    //We start memory writing
    writeCommand(CMD_MEMORY_WRITE);
    while(pixelsToSend)
    {
        //We read pixel n and n+1
        pixelA = pgm_read_word(Picture++);
        pixelB = pgm_read_word(Picture);
        //If the pixel n is different than the pixel n+1 or if it is the last pixel we send it to the TFT
        if(pixelA != pixelB || pixelsToSend == 1)
        {
            writeData(pixelA >> 8); writeData(pixelA & 0xFF);
            pixelsToSend--;
        }
        else
        {
            //Otherwise we read how many the same pixels we need to send (third word)
            pixelB = pgm_read_word(++Picture);
            pixelsToSend -= pixelB;
            Picture++;
            //We sent them to the TFT
            while(pixelB--) {writeData(pixelA >> 8); writeData(pixelA & 0xFF);}
        }
    }
}
```

na ograniczenie rozmiaru tablicy przechowującej treść obrazu. Ideę działania zastosowanego algorytmu kompresji najlepiej jest prześledzić analizując przykładowe ciągi słów 16-bitowych reprezentujących kolory kolejnych pikseli obrazu (założono powtarzanie się pikseli o kolorze 0xAAAA), które to przedstawiono w **tabeli 2**.

Jak widać, idea działania wspomnianego algorytmu jest niezwykle prosta, a pomimo tego, dla obrazków o dużych obszarach jednolitych kolorów, osiągany stopień kompresji jest całkiem spory, co pozytywnie wpływa na użycie pamięci Flash mikrokontrolera. Wymownym dowodem powyższej tezy jest **rysunek 3**, na którym przedstawiono obrazek i wynikową tablicę elementów 16-bitowych, reprezentującą jego treść w wypadku stosowania jak i niestosowania wspomnianego algorytmu kompresji obrazu.

Do „pełni szczęścia” brakuje nam dedykowanej aplikacji, za której pomocą dokonamy konwersji pliku BMP do postaci tablicy języka C (o wspomnianej wcześniej organizacji danych) i która to dodatkowo umożliwi nam skompresowanie obrazka wedle powyższego algorytmu. Na szczęście napisanie takiej aplikacji nie jest rzeczą zbyt skomplikowaną. Specjalnie na potrzeby tego projektu napisałem aplikację, która realizuje wymaganą funkcjonalność – pokazano ją na **rysunku 4**. Myślę, że przedstawione oprogramowanie jest na tyle proste i czytelne, iż nie wymaga dodatkowego słowa komentarza. Jedyne, czego potrzebujemy to funkcja, która umożliwi nam wyświetlenie na ekranie wyświetlacza TFT tak skompresowanego obrazka – pokazano ją na **listingu 12**.

Płytka ewaluacyjna

To tyle, jeśli chodzi o obsługę naszego, niezmiernie ciekawego wyświetlacza TFT. Na **rysunku 5** pokazano schemat ideowy płytki ewaluacyjnej, która umożliwia przetestowanie możliwości wyświetlacza GT-S5230, zanim zastosujemy go w docelowej aplikacji. Przypomnijmy, że płytka jest dostępna w ofercie AVT (numer kitu AVT-5563). Zastosowany typ mikrokontrolera nie jest „krytyczny” i wynika wyłącznie z faktu dysponowania odpowiednią liczbą wyprowadzeń I/O i dużą częstotliwością taktowania przy napięciu zasilania równym 3,3 V. Jako dodatkową funkcjonalność wprowadzono możliwość sterowania jasnością podświetlenia wyświetlacza TFT (tranzystor T1 i wyprowadzenie OC2A mikrokontrolera, na którym sprzętowo możemy generować przebieg PWM) oraz dodatkowy przycisk SW do użycia w aplikacji użytkownika.

Kończąc mam nadzieję, że ten krótki artykuł spowoduje, iż konstruktorzy coraz częściej będą sięgali po wyświetlacz TFT, zamiast stosowania prostych wyświetlaczy alfanumerycznych o znacznie ograniczonej funkcjonalności. Przecież jego cena nie pozostawia złudzeń odnośnie do tego, jakiego rodzaju element powinniśmy wybrać.

Robert Wołgajew, EP

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: **ftp://ep.com.pl**, user: **33948**, pass: **5gcckdmq**