W ofercie AVT*
AVT6035

Najważniejsze parametry:
- napięcie zasilania: 4...5 V,
- prąd obciążenia (średni/maksymalny): 30 mA/110 mA (emisja dźwięku),
- dopuszczalny prąd styków przekaźnika: 10 A @ 30 VDC, 0,3 A @ 125 VAC.

Dodatkowe materiały do pobrania ze strony www.ulubionykiosk.pl/media

AVT6009	multiLock (EP 11/2023)
----	Simple Access System 2, część 2 (EP 6/2022)
----	Simple Access System 2, część 1 (EP 5/2022)
----	NFC Lock (EP 4/2022)
AVT5186	Bezstykowy zamek RFID (-)
AVT969	Bezstykowy zamek RFID (-)
AVT3129	Zamek elektroniczny/immobilizer (-)
AVT886	System bezstykowej kontroli dostępu (EP 10/2000)

* **Uwaga!** Elektroniczne zestawy do samodzielnego montażu. Wymagana umiejętność lutowania! Podstawową wersją zestawu jest wersja **[B]** nazywana potocznie KIT-em (z ang. zestaw). Zestaw w wersji **[B]** zawiera elementy elektroniczne (w tym **[UK]** – jeśli występuje w projekcie), które należy samodzielnie wlutować w dołączoną płytkę drukowaną (PCB). Wykaz elementów znajduje się w dokumentacji, która jest podlinkowana w opisie kitu. Mając na uwadze różne potrzeby naszych klientów, oferujemy dodatkowe wersje:
- **wersja [C]** – zmontowany, uruchomiony i przetestowany zestaw **[B]** (elementy wlutowane w płytkę PCB),
- **wersja [A]** – płytka drukowana bez elementów i dokumentacji.

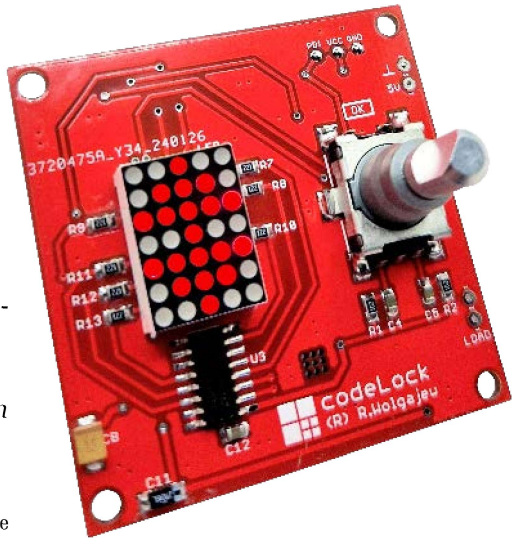
Kity, w których występuje układ scalony wymagający zaprogramowania, mają następujące dodatkowe wersje:
- **wersja [A+]** – płytka drukowana **[A]** + zaprogramowany układ **[UK]** i dokumentacja,
- **wersja [UK]** – zaprogramowany układ.

Nie każdy zestaw AVT występuje we wszystkich wersjach! Każda wersja ma załączony ten sam plik PDF! Podczas składania zamówienia upewnij się, którą wersję zamawiasz! **http://sklep.avt.pl**

W przypadku braku dostępności na stronie sklepu osoby zainteresowane zakupem płytek drukowanych (PCB) prosimy o kontakt via e-mail: **kity@avt.pl**.

codeLock – efektowny zamek kodowy

Zamki kodowe, podobnie jak termometry, termostaty czy miniaturowe radyjka, to elementarz każdego elektronika amatora. Któż z nas nie ma w swoim portfolio tego typu urządzeń, które – mimo oczywistej prostoty – dają dużo radości z własnoręcznego konstruowania nowych rozwiązań. Również i ja na liście skonstruowanych oraz zaprojektowanych przez siebie urządzeń mam kilka takich systemów, cechujących się różnym stopniem skomplikowania. Niemniej jednak postanowiłem, iż tym razem zaprojektuję urządzenie, które pogodzi pozornie sprzeczne założenia. Z jednej strony chciałem, by odznaczało się ono dużą prostotą implementacji oraz nieskomplikowaną obsługą, a z drugiej strony – efektownym i nowoczesnym interfejsem użytkownika.



Postanowiłem zbudować prosty zamek kodowy, do którego kod użytkownika wprowadzać będziemy za pomocą elektronicznego pokrętła wydającego dźwięk „tykania", charakterystyczny dla pokręteł kodowych wielkich sejfów bankowych, znanych chociażby z filmowych produkcji. Ponadto założyłem, że wyświetlane przez zamek kodowy cyfry przesuwać się będą w rytm obracania wspomnianego pokrętła, tak jakbyśmy używali cylindrycznego bębna z nadrukowanymi nań znakami (coś w rodzaju starych liczników w magnetofonach z lat 80.). Jak łatwo się domyślić, w roli wspomnianego wcześniej pokrętła zastosowałem prosty enkoder inkrementalny, zaś funkcję wyświetlacza, za pomocą którego udało się osiągnąć efekt animacji zmiany cyfr, pełni prosta matryca diod LED o organizacji 5×7 punktów. I właśnie na bazie powyższych założeń powstał projekt urządzenia codeLock, którego schemat pokazano na **rysunku 1.**

Jak widać, zaprojektowany system mikroprocesorowy jest bardzo prosty, a jego serce stanowi niewielki, ale bardzo nowoczesny mikrokontroler ATtiny1604 firmy Microchip (dawniej Atmel), taktowany wewnętrznym oscylatorem RC o częstotliwości 10 MHz i realizujący całą założoną funkcjonalność urządzenia. Mikrokontroler nasz steruje pracą szeregowego rejestru przesuwnego 74HC4094 (wyprowadzenia PA3/SCK→Clock, PA0/MOSI→Data), dzięki któremu realizuje obsługę matrycowego wyświetlacza LED w konfiguracji wspólnej anody (wyprowadzenia PA7...PA4, PA2 mikrokontrolera), obsługuje enkoder inkrementalny z wbudowanym przyciskiem (dzięki zastosowaniu przerwania od zmiany stanu pinów portu PORTB mikrokontrolera – w naszym przypadku pinu PB1), steruje pracą przekaźnika LOAD (poprzez prosty klucz tranzystorowy NPN) oraz odpowiedzialny jest za generowanie dźwięku poprzez wbudowany głośniczek SMD, co realizuje za pomocą wbudowanych w swoją strukturę układów czasowo-licznikowych: TCA0 (pracującego w trybie PWM) oraz TCB0 (pracującego w trybie Periodic Interrupt). Wybór mikrokontrolera ATtiny1604 oraz podłączonego do jego wyprowadzeń rejestru szeregowego 74HC4094 mógłby się wydawać dość wątpliwy, jeśli wziąć pod uwagę, że bez

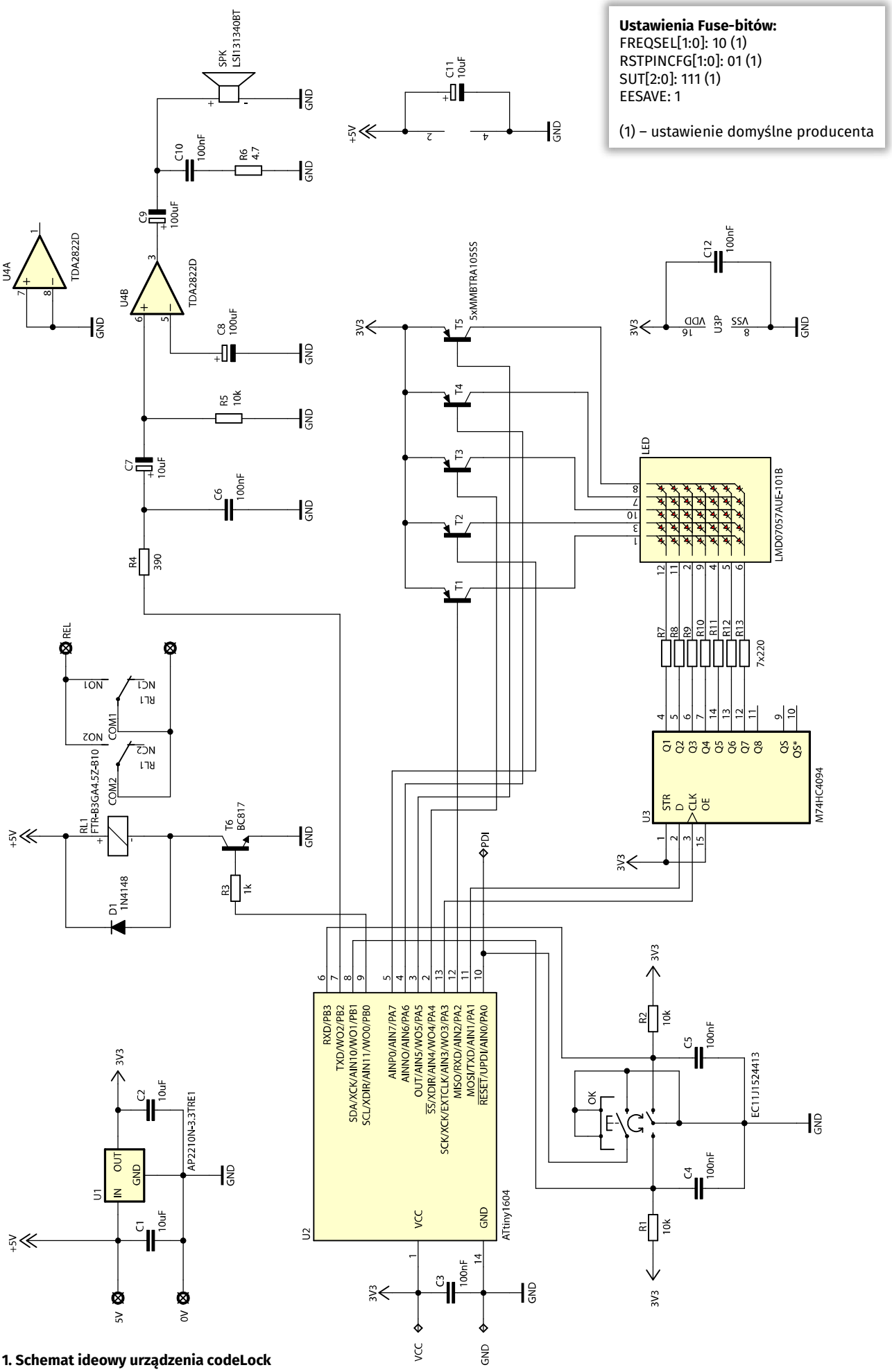
Wykaz elementów:

Rezystory: (SMD 0805)
R1, R2, R5: 10 kΩ
R3: 1 kΩ
R4: 390 Ω
R6: 4,7 Ω
R7...R13: 220 Ω

Kondensatory:
C1, C2: ceramiczny X7R 10 µF (SMD 0805)
C3...C6, C10, C12: ceramiczny X7R 100 nF (SMD 0805)
C7, C11: tantalowy 10 µF/6,3 V (A/3216-18W)
C8, C9: tantalowy 100 µF/6,3 V (B/3528-21W)

Półprzewodniki:
U1: AP2210N-3.3TRE1 (SOT-23)
U2: ATtiny1604 (SO-14)
U3: M74HC4094 (SO16)
U4: TDA2822D (SO-8)
LED: wyświetlacz matrycowy 5×7 LED typu LMD07057AUE-101B lub w wybranym kolorze
T1...T5: MMBTRA105SS (SOT-23)
T6: BC817 (SOT-23)
D1: 1N4148 (miniMELF)

Inne:
OK: enkoder SMD z przyciskiem typu EC11J1524413
RL1: przekaźnik SMD typu FTR-B3GA4.5Z-B10
SPK: głośniczek SMD typu LSI131340BT-08-105



Rysunek 1. Schemat ideowy urządzenia codeLock

problemu dałoby się tu wybrać mikrokontroler o odpowiedniej liczbie portów wyjściowych, zamiast stosować procesor i rejestr przesuwny. To tylko pozory! Po pierwsze, nie chciałem stosować mikrokontrolera o większej liczbie (niepotrzebnych) wyprowadzeń, a co za tym idzie – o niewygodnej do lutowania dla amatora obudowie (TQFP32). Jeszcze istotniejszy jest jednak fakt, że zastosowanie rejestru przesuwnego 74HC4094 zdecydowanie upraszczało projekt obwodu drukowanego (tak, tak!); nie wspominając już o cenie układu tego rodzaju, wynoszącej około 1 zł. Co warto również zaznaczyć, do sterowania rejestru przesuwnego używany jest wbudowany w strukturę mikrokontrolera interfejs SPI, przez co jego obsługa stała się niezmiernie prosta i szybka (zegar SPI_{SCK}=5 MHz).

Wspomniane wcześniej wspólne anody wyświetlacza LED sterowane są poprzez proste klucze tranzystorowe T1…T5 (ze zintegrowanymi w ich strukturze rezystorami: bazowym i podciągającym), z uwagi na dość duże prądy o wartościach rzędu 35 mA (7×5 mA). Z kolei wspólne katody naszych elementów LED obsługiwane są przez wyprowadzenia rejestru przesuwnego i, jak już można się domyślić, do ich obsługi (i obsługi wspólnych anod) zastosowany został doskonale znany mechanizm multipleksowania. Jest to standardowe rozwiązanie problemu tego typu, a polega na sekwencyjnym sterowaniu kolejnych kolumn wyświetlacza LED, w którym przeprowadzane są kolejne i następujące po sobie poniższe operacje:

- wyłączamy wszystkie wspólne anody (a tym samym wszystkie elementy LED),
- na port wspólnych katod wystawiamy (poprzez rejestr przesuwny) „wzór” do wyświetlenia (aktywny stan „0”),
- załączamy wybraną wspólną anodę (aktywny stan „0”), wyświetlając tym samym wcześniejszy „wzór” na wybranej kolumnie diod LED,
- powtarzamy powyższy proces dla kolejnych wspólnych anod.

Opisany proces, wykonywany dostatecznie szybko (w naszym wypadku 60 razy na sekundę dla każdej wspólnej anody), pozwala na obsłużenie 35 elementów LED (diod reprezentujących wyświetlaną treść), przy udziale wyłącznie 7 wyprowadzeń mikrokontrolera. Prawda, że proste? A jakie efektywne! Już teraz uprzedzę Czytelników, że użyty w tym celu zostanie układ czasowo-licznikowy RTC wbudowany w strukturę mikrokontrolera, który pracował będzie w trybie Periodic Interrupt i wywoływał stosowne przerwanie systemowe (od przepełnienia) 300 razy na sekundę (czyli 60 razy dla każdej wspólnej anody), obsługując właściwy mechanizm multipleksowania. Ale wróćmy do schematu ideowego naszego urządzenia, gdyż kilka niezbędnych słów komentarza wymaga blok generujący dźwięk. Jak działa ten mechanizm? Jak już wspomniałem wcześniej, do generowania dźwięku zaprzęgnięto dwa układy czasowo licznikowe: TCA0 pracujący w trybie PWM oraz TCB0 pracujący w trybie Periodic Interrupt. Timer TCA0 generuje na wyprowadzeniu PB2 (WO2) mikrokontrolera 8-bitowy przebieg PWM, którego wypełnienie zależne jest od wartości jego rejestru porównania CMP2.

Przebieg ten przechodzi następnie przez prosty filtr dolnoprzepustowy R4/C6 o częstotliwości odcięcia około 4 kHz, na którego wyjściu otrzymujemy sygnał DC proporcjonalny do wypełnienia zasilającego filtr przebiegu PWM. W ten prosty sposób realizujemy książkowy przykład najprostszego przetwornika DAC. Nie ma on, co prawda, zbyt wyszukanych parametrów, gdyż nie mamy możliwości większego odstrojenia się pasmem częstotliwości przebiegu PWM od pasma użytecznego sygnału (bez utraty jego rozdzielczości), ale do naszych zastosowań w zupełności wystarczy. Tak przetworzony sygnał PWM wchodzi na wejście prostej końcówki mocy, zbudowanej na bazie popularnego układu wzmacniacza audio małej mocy (pod postacią TDA2822D produkcji firmy STMicroelectronics w jego typowej aplikacji), przy czym, co łatwo zauważyć, użyty został jedynie jeden ze wzmacniaczy wbudowanych w strukturę tego stereofonicznego układu, gdyż nie zależy nam na zwiększaniu mocy sygnału audio (np. z zastosowaniem układu mostkowego wzmacniaczy). Sygnał wyjściowy układu TDA2822D trafia, po odcięciu składowej stałej (kondensator C9), na miniaturowy głośniczek SMD. Ale jak generowany jest sam dźwięk? Za ten etap odpowiada drugi układ czasowo-licznikowy, mianowicie TCB0. Układ ten, pracujący w trybie Periodic Interrupt, wywołuje stosowne przerwanie sprzętowe (8000 razy na sekundę), które ładuje z pamięci flash mikrokontrolera kolejne wartości spróbkowanego wcześniej sygnału dźwiękowego do rejestru porównania licznika TCA0 (CMP2BUF), powodując stosowne zmiany napięcia na wyjściu filtra R4/C6. Wspominane powyżej próbki dźwiękowe zostały wcześniej zdigitalizowane (a następnie zapisane w pamięci Flash) z częstotliwością próbkowania 8 kHz i rozdzielczością 8-bitów, co zapewne wyjaśni Czytelnikom konfigurację obu timerów.

Tyle kwestii funkcjonalnych – przejdźmy zatem do zagadnień implementacyjnych. Mam świadomość, że nie jest to *rocket science* ani rozwiązanie na wskroś uniwersalne, ale chciałem pokazać Czytelnikom, jak w efektywny i efektowny sposób ogarnąć tego rodzaju zagadnienie programistyczne, czyniąc sam proces programowania niezmiernie przyjemnym. Nieskromnie powiem, że w moim przekonaniu właśnie w ten przejrzysty sposób powinno się konstruować moduły obsługi danych peryferiów, gdyż jakakolwiek modyfikacja sprowadza się wtedy do kosmetycznych i prostych do wykonania zmian.

Na początek plik nagłówkowy mechanizmu multipleksowania, który pokazano na **listingu 1**, a dzięki któremu porządkujemy późniejszy kod źródłowy, czyniąc go bardzo czytelnym – i jednocześnie upraszczając proces wprowadzania potencjalnych zmian. Plik ten definiuje główne ustawienia sprzętowe oraz wprowadza niezbędne zmienne.

Jak widać, w ramach pliku nagłówkowego zadeklarowano szereg zmiennych globalnych (typu volatile – z uwagi na ich wykorzystanie zarówno w programie głównym, jak i funkcji ISR). Przechowują one treść wyświetlaną

```
//Definicje portów rejestru przesuwnego
#define SERIAL_PORT_NAME PORTA
#define SERIAL_DAT_MASK PIN1_bm
#define SERIAL_CLK_MASK PIN3_bm

//Porty rejestru przesuwnego (DAT, CLK), jako wyjściowe ze stanem 0
#define SERIAL_PORT_AS_OUTPUT SERIAL_PORT_NAME.DIRSET = SERIAL_DAT_MASK|SERIAL_CLK_MASK

//Definicje portu wspólnych anod – tranzystory sterujące
#define ANODE_PORT_NAME PORTA

//Definicje konfiguracji poszczególnych wspólnych anod
#define ANODE1_MASK PIN2_bm
#define ANODE2_MASK PIN7_bm
#define ANODE3_MASK PIN4_bm
#define ANODE4_MASK PIN6_bm
#define ANODE5_MASK PIN5_bm

//Port wspólnych anod, jako port wyjściowy
#define ANODE_AS_OUTPUT ANODE_PORT_NAME.DIRSET = ANODE1_MASK|ANODE2_MASK|ANODE3_MASK|ANODE4_MASK|ANODE5_MASK
//Wszystkie wspólne anody wyłączone ("1", gdyż sterujemy bazami tranzystorów PNP)
#define ANODE_BLANK ANODE_PORT_NAME.OUTSET = ANODE1_MASK|ANODE2_MASK|ANODE3_MASK|ANODE4_MASK|ANODE5_MASK


//Deklaracje zmiennych globalnych
extern volatile uint8_t Column[5];              //Zmienna przechowująca treść wyświetlaną na wyświetlaczu LED
extern volatile uint8_t readyForUpdate;         //Zezwolenie na atomową zmianę zmiennych


void initMultiplex(void);
void showDigit(uint8_t Digit, uint8_t Offset);
```

Listing 1. Plik nagłówkowy mechanizmu multipleksowania

```
//Definicje wzorców cyfr
const uint8_t Font5x8[] =
{
    ~0x3E, ~0x51, ~0x49, ~0x45, ~0x3E,          // 0
    ~0x00, ~0x42, ~0x7F, ~0x40, ~0x00,          // 1
    ~0x42, ~0x61, ~0x51, ~0x49, ~0x46,          // 2
    ~0x21, ~0x41, ~0x45, ~0x4B, ~0x31,          // 3
    ~0x18, ~0x14, ~0x12, ~0x7F, ~0x10,          // 4
    ~0x27, ~0x45, ~0x45, ~0x45, ~0x39,          // 5
    ~0x3C, ~0x4A, ~0x49, ~0x49, ~0x30,          // 6
    ~0x01, ~0x71, ~0x09, ~0x05, ~0x03,          // 7
    ~0x36, ~0x49, ~0x49, ~0x49, ~0x36,          // 8
    ~0x06, ~0x49, ~0x49, ~0x29, ~0x1E,          // 9
    ~0x3E, ~0x51, ~0x49, ~0x45, ~0x3E,          // 0 – ponownie, dla mechanizmu animacji
    ~0x14, ~0x36, ~0x7F, ~0x36, ~0x14,          // Strzałka
    ~0x08, ~0x08, ~0x3E, ~0x08, ~0x08,          // Plus
    ~0x08, ~0x08, ~0x08, ~0x08, ~0x08           // Minus
};
//Definicje dla portu sterującego wspólnymi anodami wyświetlaczy LED (aktywny stan "0", gdyż sterujemy bazami tranzystorów PNP)
const uint8_t colPattern[] =
{
    ANODE1_MASK,                                // Wspólna anoda kolumny 1 (pierwsza z lewej)
    ANODE2_MASK,                                // Wspólna anoda kolumny 2
    ANODE3_MASK,                                // Wspólna anoda kolumny 3
    ANODE4_MASK,                                // Wspólna anoda kolumny 4
    ANODE5_MASK,                                // Wspólna anoda kolumny 5 (pierwsza z prawej)
};
```

Listing 2. Definicje niezbędnych stałych mechanizmu multipleksowania

```
void initMultiplex(void)
{
    //Inicjalizacja interfejsu SPI sterującego rejestrem przesuwnym
    initSerial();

    //Porty wspólnych anod i interfejsu szeregowego, jako wyjściowe ze stanami nieaktywnymi na wyjściach
    SERIAL_PORT_AS_OUTPUT;
    ANODE_BLANK;
    ANODE_AS_OUTPUT;

    //Konfiguracja zegara RTC w celu generowania przerwania do obsługi multipleksowania wyświetlacza LED (300 Hz)
    while(RTC.STATUS & (RTC_CTRLABUSY_bm|RTC_CNTBUSY_bm|RTC_PERBUSY_bm|RTC_CMPBUSY_bm));
                                                //Czekamy na zakończenie synchronizacji RTC (4 flagi)
    RTC.PER = 108;                              //300 Hz @ fRTC = 32768 Hz (ISR OVF co 3.33 ms)
    RTC.CTRLA = RTC_PRESCALER_DIV1_gc|RTC_RTCEN_bm;  //Preskaler = 1, uruchomienie zegara RTC @ fRTC = 32768 Hz
    RTC.INTCTRL = RTC_OVF_bm;                   //Uruchomienie przerwania od przepełnienia zegara RTC
}
```

Listing 3. Funkcja konfigurująca mechanizm multipleksowania

na wyświetlaczu LED, a także upraszczają proces jej aktualizacji. Niemniej jednak już na tym etapie musimy zdefiniować kilka stałych opisujących wzorce znaków oraz upraszczające dostęp do portów sterujących, gdyż zależy nam na tym, by nasza procedura obsługi przerwania – multipleksująca wyświetlacz – była jak najkrótsza. Definicje, o których mowa, pokazano na **listingu 2**.

Jak widać, wszystkie spośród definicji omówionych powyżej zostały umieszczone w pamięci RAM mikrokontrolera. Jest to pewnego rodzaju „marnotrawstwo”, gdyż stałe te z powodzeniem można (a może nawet wypada?) umieścić w pamięci Flash mikrokontrolera, aby nie marnować cennej pamięci RAM, zwłaszcza że wartości tych stałych nasz kompilator i tak musi umieścić, a następnie odczytać z tejże pamięci Flash na starcie programu obsługi aplikacji (bo niby skąd miałby wziąć te wartości, by podstawić pod odpowiednie tablice?). Dokładnie tak postępowałem dotychczas, pisząc oprogramowanie embedded, jednak dostęp do pamięci Flash jest nieco wolniejszy niż odczyt stałych z pamięci RAM (dokładnie 5 taktów zegara zamiast 2). W związku z tym zdecydowałem się na powyższe rozwiązanie, zwłaszcza że wykorzystanie pamięci RAM w naszej aplikacji utrzymuje się na poziomie 20%. Niby niewielki przyrost szybkości, ale jednak mierzalny. Skądinąd jest to zgodne z podejściem twórców Androida, scharakteryzowane w zdaniu: „dlaczego nieużywana pamięć RAM ma leżeć odłogiem”? Abstrahując już od celowości i sensowności takiego postępowania, brnijmy dalej. Pora na przedstawienie funkcji konfigurującej zarówno mechanizm multipleksowania, jak i niezbędne ustawienia sprzętowe, której ciało pokazano na **listingu 3**.

Dalej, na **listingu 4** przedstawiono z kolei funkcję obsługi przerwania od przepełnienia licznika RTC, odpowiedzialną za realizację mechanizmu multipleksowania wyświetlacza LED.

I na sam koniec, na **listingach 5** i **6** przedstawiono dwie proste funkcje narzędziowe odpowiedzialne za konfigurację interfejsu SPI mikrokontrolera oraz za przesłanie bajtu danych do rejestru przesuwnego.

Prawda, że proste? Niemniej jednak warto choćby na chwilę zastanowić się nad znaczeniem nieopisanej wcześniej zmiennej readyForUpdate. Jest to zmienna, która funkcji

```
//Przerwanie obsługi wyświetlacza LED wywoływane co 3.33 ms
//(60 razy na sekundę dla każdej kolumny LED)

ISR(RTC_CNT_vect)
{
    static uint8_t Nr;  //Numer kolejnej kolumny przeznaczonej do wyświetlenia

    //Skasowanie flagi OVF, gdyż nie jest kasowana sprzętowo
    RTC.INTFLAGS = RTC_OVF_bm;
    //Wyłączenie wszystkich wspólnych anod wyświetlacza LED
    ANODE_BLANK;
    //Wystawienie wzoru na port katod wyświetlacza LED (rejestr przesuwny)
    serialTransfer(Column[Nr]);
    //Włączenie odpowiedniej wspólnej anody wyświetlacza LED (aktywny stan "0")
    ANODE_PORT_NAME.OUTCLR = colPattern[Nr];
    //Wybranie kolejnej wspólnej anody wyświetlacza LED
    if(++Nr > 4) Nr = 0;
    //Zezwolenie na atomową zmianę zmienej Column[] w funkcji Main
    if(Nr == 0) readyForUpdate = 1; else readyForUpdate = 0;
}
```

Listing 4. Funkcja obsługi przerwania realizująca mechanizm multipleksowania

```
void initSerial(void)
{
    //MSB, jako pierwsze, tryb Master, zegar = 5 MHz @ fCLK = 10 MHz, włączenie SPI
    SPI0.CTRLA = SPI_MASTER_bm|SPI_CLK2X_bm|SPI_PRESC_DIV4_gc|SPI_ENABLE_bm;
    //Wyłączenie funkcjonalności pinu SS dla trybu Host
    SPI0.CTRLB = SPI_SSD_bm;
}
```

Listing 5. Funkcja odpowiedzialna za konfigurację interfejsu SPI mikrokontrolera

głównej aplikacji użytkownika wskazuje moment atomowej aktualizacji zmiennych volatile procedury obsługi przerwania mechanizmu multipleksowania. Potrzeba wprowadzenia takiej zmiennej wynikała z konieczności synchronizacji chwili aktualizacji zmiennych, dokonywanej w aplikacji głównej, z pracą funkcji multipleksującej wyświetlacz LED – tak by nie występowało zjawisko „mieszania" zawartości zmiennych z kolejnych przebiegów funkcji multipleksującej. Aktualizacja, o której mowa powyżej, następuje po pełnym cyklu multipleksu dla całego wyświetlacza LED. I tutaj brakuje nam jeszcze jednej funkcji, mianowicie wyświetlającej stosowny wzorzec znaku na elemencie LED, której ciało pokazano na **listingu 7**.

Jak widać, funkcja przyjmuje argument przesunięcia w pionie wzorca znaku o liczbę zdefiniowanych pikseli obrazu (w zakresie 0…8), co zostanie użyte w mechanizmie animacji zmian wyświetlanych cyfr, o czym pisałem na wstępie. Tyle, jeśli chodzi o funkcje obsługi wyświetlacza LED. Przejdźmy teraz do grupy funkcji odpowiedzialnych za generowanie dźwięku. Zacznijmy jak zwykle od pliku nagłówkowego, pokazanego na **listingu 8**, dzięki któremu porządkujemy późniejszy kod źródłowy (czyniąc go bardzo czytelnym, a jednocześnie upraszczając proces wprowadzania zmian). Plik ten zarówno definiuje główne ustawienia sprzętowe, jak i wprowadza niezbędne zmienne.

Dalej, na **listingu 9** zaprezentowano funkcję konfigurującą timer TCA0 jako generator przebiegu PWM, zaś na **listingu 10** – funkcję konfigurującą timer TCB0, będący podstawą czasu mechanizmu generowania dźwięku.

Następnie, na **listingu 11**, uwidoczniono ciało funkcji obsługi przerwania timera TCB0 pracującego w trybie Periodic Interrupt, odpowiedzialną za wysyłanie próbek dźwięku na port mikrokontrolera obsługujący głośnik SMD.

Jak widać, funkcja pobiera kolejne próbki sygnału z tablicy (Tada[], Tick[] lub Error[]) zapisanej w pamięci Flash mikrokontrolera (z uwagi na jej rozmiar). Z myślą o bardziej dociekliwych Czytelnikach dodam, że 3 zdigitalizowane i bardzo krótkie próbki dźwięku zajmują prawie 10 kB wspomnianej pamięci Flash – i to mimo faktu, że są 8-bitowe oraz, oględnie mówiąc, nie grzeszą jakością. Na szczęście w naszym zastosowaniu niedogodność ta pozostaje właściwie bez większego znaczenia. I na koniec, na **listingu 12** przedstawiono funkcję inicjującą proces odtwarzania dźwięku.

Tyle w kwestiach implementacyjnych. Przejdźmy zatem do schematu montażowego naszego urządzenia, pokazanego na **rysunku 2**.

Jak widać, zaprojektowano bardzo zgrabną, dwustronną, niewielką płytkę drukowaną ze zdecydowaną przewagą elementów SMD umieszczonych po obu stronach laminatu. Montaż urządzenia rozpoczynamy od warstwy TOP, na której w pierwszej kolejności przylutowujemy wszystkie półprzewodniki, w tym wyświetlacz LED. Proces ten najłatwiej wykonać przy użyciu stacji lutowniczej na gorące powietrze (tzw. Hot-Air) i odpowiednich stopów lutowniczych. Jeśli jednak nie dysponujemy tego rodzaju sprzętem, można również

```
uint8_t serialTransfer(uint8_t Byte)
{
    //Inicjujemy transmisję bajtu do rejestru przesuwnego (począwszy od bitu MSB)
    SPI0.DATA = Byte;
    //Czekamy na jego przesłanie
    while(!(SPI0.INTFLAGS & SPI_IF_bm));
    //Zwracamy przesłany przez układ Slave bajt
    return SPI0.DATA;
}
```

Listing 6. Funkcja odpowiedzialna za przesłanie bajtu danych do rejestru przesuwnego

```
void showDigit(uint8_t Digit, uint8_t Offset)
{
    uint8_t Index, prevByte, nextByte;

    //Ustalamy index początku wzorca cyfry, którą to zamierzamy wyświetlić
    Index = Digit*5;

    //Czekamy na zezwolenie na aktualizację zawartości wyświetlacza LED
    while(readyForUpdate == 0);

    //Aktualizujemy zawartość wyświetlacza uwzględniając przesunięcie wzorca cyfry
    for(uint8_t i=0; i<5; ++i)
    {
        prevByte = Font5x8[Index] >> Offset;
        nextByte = Font5x8[Index+5] << (8-Offset);

        Column[i] = prevByte|nextByte;
        Index++;
    }

    //Kasujemy zezwolenie na aktualizację zawartości wyświetlacza LED
    readyForUpdate = 0;
}
```

Listing 7. Funkcja odpowiedzialna za wyświetlenie wzorca znaku na wyświetlaczu LED

```
//Definicje portu PWM
#define PWM_PORT_NAME PORTB
#define PWM_PORT_MASK PIN2_bm                                   //PB2 -> WO2

//Definicje dla mechanizmu odtwarzania próbek dźwięku PWM
#define START_PLAYING TCB0.CTRLA |= TCB_ENABLE_bm               //Uruchomienie timera TCB0
#define STOP_PLAYING TCB0.CTRLA &= ~TCB_ENABLE_bm               //Zatrzymanie timera TCB0

//Definicje typów odtwarzanych dźwięków
#define SOUND_TADA 0
#define SOUND_TICK 1
#define SOUND_ERROR 2

//Prototypy funkcji
void initPWM(void);
void initSound(void);
void playSound(uint8_t Type);
```

Listing 8. Plik nagłówkowy mechanizmu generowania dźwięku

```
//Konfiguracja timera TCA0 generującego przebieg PWM na wyprowadzeniu PB2 (WO2) mikrokontrolera

void initPWM(void)
{
    //Port PWM-a, jako wyjściowy
    PWM_PORT_NAME.DIRSET = PWM_PORT_MASK;  //PB2 -> WO2
    //Rozdzielczość PWM = 8 bitów, częstotliwość PWM = 39 kHz @ fCLK = 10 MHz (fCLK/Preskaler*(PER+1))
    TCA0.SINGLE.PER = 255;
    //Tryb Single-slope PWM, włączenie porównania na kanale 2 (WO2)
    TCA0.SINGLE.CTRLB = TCA_SINGLE_WGMODE_SINGLESLOPE_gc|TCA_SINGLE_CMP2EN_bm;
    //Włączenie timera TCA0, Preskaler = 1
    TCA0.SINGLE.CTRLA = TCA_SINGLE_CLKSEL_DIV1_gc|TCA_SINGLE_ENABLE_bm;
}
```

Listing 9. Funkcja konfigurującą timer TCA0, jako generator przebiegu PWM

```
//Konfiguracja timera TCB0 odpowiedzialnego za wysyłanie próbek dźwięku na port PWM-a mikrokontrolera

void initSound(void)
{
    //Tryb Periodic Interrupt
    TCB0.CTRLB = TCB_CNTMODE_INT_gc;
    //Preskaler = 1 (tymczasem BEZ uruchamiania timera)
    TCB0.CTRLA = TCB_CLKSEL_CLKDIV1_gc;
    //Przerwanie Capture 8000 razy na sekundę @ fCLK = 10 MHz
    TCB0.CCMP = 1249;
    //Włączenie przerwania Capture (jedyne dla wszystkich trybów pracy timera)
    TCB0.INTCTRL = TCB_CAPT_bm;
}
```

Listing 10. Funkcja konfigurującą timer TCB0 będący podstawą czasu mechanizmu generowania dźwięku

```
//Przerwanie wywoływane 8000 razy na sekundę odpowiedzialne za wysyłanie próbek dźwięku
//na port PWM-a mikrokontrolera

ISR(TCB0_INT_vect)
{
    static uint16_t pcmNr;              //Numer próbki PCM z tablicy próbek

    //Kasujemy flagę Capture, gdyż nie jest kasowana sprzętowo
    TCB0.INTFLAGS = TCB_CAPT_bm;

    //Wysyłamy kolejną próbke PCM na wyjście PWM lub zatrzymujemy proces,
    //gdy wszystkie próbki zostały już wysłane
    switch(soundType)
    {
        case SOUND_TADA: TCA0.SINGLE.CMP2BUF = pgm_read_byte(&Tada[pcmNr]); break;
        case SOUND_TICK: TCA0.SINGLE.CMP2BUF = pgm_read_byte(&Tick[pcmNr]); break;
        case SOUND_ERROR: TCA0.SINGLE.CMP2BUF = pgm_read_byte(&Error[pcmNr]); break;
    }

    if(++pcmNr > soundLenght-1)
    {
        pcmNr = 0;
        STOP_PLAYING;
    }
}
```

Listing 11. Funkcja obsługi przerwania timera TCB0 odpowiedzialna za wysyłanie próbek dźwięku

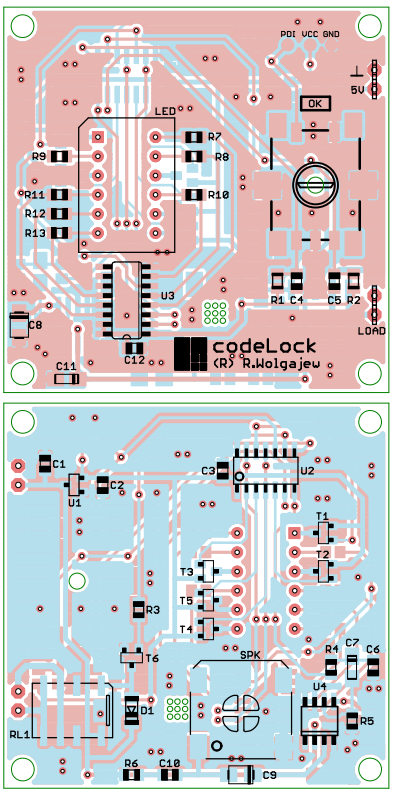
```
void playSound(uint8_t Type)
{
    soundType = Type;

    switch(Type)
    {
        case SOUND_TADA: soundLenght = TADA_ELEMENTS; break;
        case SOUND_TICK: soundLenght = TICK_ELEMENTS; break;
        case SOUND_ERROR: soundLenght = ERROR_ELEMENTS; break;
    }

    START_PLAYING;
}
```

Listing 12. Funkcja inicjująca proces odtwarzania dźwięku

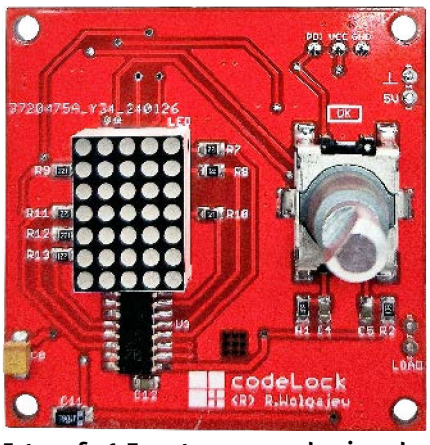
zastosować metodę z wykorzystaniem typowej stacji lutowniczej. Najprostszym sposobem montażu elementów o tak dużym zagęszczeniu wyprowadzeń, niewymagającym jednocześnie posiadania specjalistycznego sprzętu, jest użycie zwykłej stacji lutowniczej, dobrej jakości cyny z odpowiednią ilością topnika oraz dość cienkiej plecionki rozlutowniczej, która umożliwi usunięcie nadmiaru cyny spomiędzy wyprowadzeń układów. Należy przy tym uważać, by nie uszkodzić termicznie tego rodzaju elementów. Następnie lutujemy elementy bierne, po czym przechodzimy na warstwę BOTTOM. Tutaj, podobnie jak poprzednio, montaż rozpoczynamy od przylutowania wszystkich półprzewodników (w tym układów scalonych), po czym montujemy pozostałe elementy bierne, następnie głośniczek SMD oraz przekaźnik. W tym momencie wracamy na warstwę TOP, gdzie przylutowujemy enkoder SMD. To etap, na którym urządzenie gotowe jest do uruchomienia. Na **rysunku 3** pokazano zmontowane urządzenie codeLock od strony warstwy TOP, zaś na **rysunku 4** – od strony warstwy BOTTOM.

Omówmy jeszcze kwestię obsługi naszego urządzenia. Tu sprawa jest niezmiernie prosta. W chwili bezczynności i oczekiwania na rozpoczęcie wprowadzania kodu urządzenie wyświetla znak „↕”. Pokręcanie osią enkodera powoduje stosowną zmianę cyfr na wyświetlaczu LED, okraszoną bardzo efektowną animacją (przekręcania się „bębna” z cyframi), z towarzyszącym jej dźwiękiem tykania. Wciśnięcie oski enkodera powoduje z kolei zatwierdzenie bieżącej cyfry i przejście do kolejnego kroku, który wygląda dokładnie tak samo. Wprowadzenie 4 cyfr kodu inicjuje jego sprawdzenie i – w przypadku zgodności z zapamiętanym – powoduje chwilowe pokazanie znaku „+” (okraszone specjalnym efektem dźwiękowym „ding”) oraz załączenie przekaźnika LOAD, po czym urządzenie przechodzi do stanu wyjściowego, czyli do początku wprowadzania kodu użytkownika (wyświetlając znak „↕”). Z kolei, jeśli wprowadzony kod nie zgadza się z kodem użytkownika, urządzenie pokaże znak „–”, po czym odtworzy dźwięk błędu i przejdzie, jak poprzednio, do początku wprowadzania tegoż kodu (wyświetlając znak „↕”). Tutaj nasuwa się pytanie: w jaki sposób wprowadzamy kod użytkownika, który ma zostać zapamiętany przez urządzenie (a następnie sprawdzany)? Również bardzo prosto. Jest to możliwe wyłącznie podczas włączania urządzenia. Jeśli podczas rozruchu wciśnięty będzie przycisk enkodera, układ przejdzie do trybu wprowadzania kodu użytkownika. Sam proces wprowadzania tego kodu wygląda dokładnie tak samo, jak normalna obsługa urządzenia, podczas której oczekuje ono na wpisanie kolejnych 4 cyfr kodu, ale z małym wyjątkiem. Otóż, podczas normalnego użytkowania urządzenia, czyli podczas wpisywania kodu do sprawdzenia, mikrokontroler odlicza czas bezczynności użytkownika (10 s), po upływie którego kasowane są wszystkie podane dotąd cyfry i proces wprowadzania startuje od nowa. W przypadku trybu wprowadzania kodu użytkownika wspomniany czas bezczynności nie ma zastosowania, w związku z czym urządzenie nie przejdzie do normalnego trybu pracy, dopóki nie zostanie wpisany cały kod użytkownika (4 cyfry). Po zakończeniu tej czynności nowy kod zapamiętany zostanie w pamięci EEPROM mikrokontrolera, zaś urządzenie przejdzie do trybu normalnego.

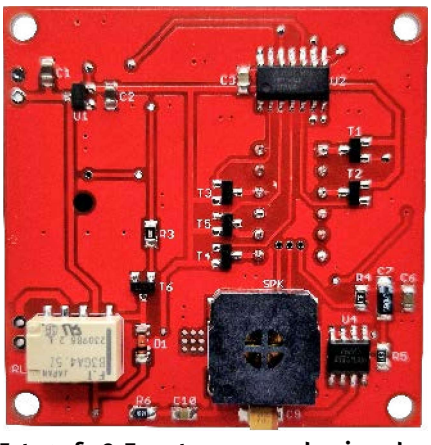
Robert Wołgajew, EP



Rysunek 2. Schemat montażowy urządzenia codeLock (a – strona TOP, b – strona BOTTOM)



Fotografia 1. Zmontowane urządzenie codeLock od strony warstwy TOP



Fotografia 2. Zmontowane urządzenie codeLock od strony warstwy BOTTOM