IQRF – więcej niż radio

Programowanie modułu TR52B

W poprzednim wydaniu Elektroniki Praktycznej opisaliśmy

Jeśli do aplikacji jest potrzebny nieskomplikowany moduł do transmisji radiowej, ale jednocześnie dający unikalne możliwości, warto zainteresować się ofertą czeskiej firmy Microrisc. Czesi wpadli na pomysł, aby połączyć moduł radiowy z mikrokontrolerem. W poprzednim wydaniu Elektroniki Praktycznej opisaliśmy warstwę sprzętową. W tym pokażemy przykłady programów wykonanych dla modułu transmisyjnego TR52B.

Dodatkowe materiały na CD/FTP:
ftp://ep.com.pl, user: 14464, pass: 87f371o5
- poprzednie artykuły o IQRF

Listing 1. Prosty program przykładowy

```
#include „../includes/template-basic.h”
void APPLICATION()
{
  uns8  i;
  for (i=0; i<3; i++)
  {
    pulseLEDR();     //systemowa funkcja migania czerwoną dioda LED
    waitDelay(100); // opóźnienie 1s (100 ticks x 10 ms)
  }
  while (1)          //nieskończona pętla
  {
    clrwdt();        //kasowanie licznika watchdoga
  }
}
```

Do pisania programów i ich uruchamiania będą potrzebne: zestaw ewaluacyjny CK-USB-02, lub CK-USB-04, zainstalowany driver USB i program IQRF IDE (dodatkowo z kompilatorem i edytorem), przynajmniej 2 moduły TR52B i zestaw DK-EVAL-03, lub DK-EVAL-04. Ten ostatni zestaw jest potrzebny do zasilania drugiego modułu radiowego. Pierwszy moduł będzie umieszczony i zasilany z modułu programatora na przykład CK-USB-04. Ja do testów użyłem nowszych wersji modułów: CK-USB-04 i DK-EVAL-04, bo dają możliwość dołączania modułów DCC Kits.

Przykład 1. Podstawy programów: migająca dioda LED, obsługa klawiszy DCC-IO-01 i sterowanie przekaźnikami DCC-RE-01

Zaczniemy od nieskomplikowanego programu przykładowego E00-START. Pokazano go na **listingu 1**. Program uruchamiany pod kontrolą systememu IQRF OS różni się od typowego programu w języku C. Nie ma tutaj funkcji *main()*, a aplikacja użytkownika zaczyna działanie od wykonywania funkcji *void APPLICATION()*. Program nie może jej opuścić, wiec musi ona zawierać pętlę nieskończoną.

Program EE0-START błyska trzykrotnie czerwoną diodą LED modułu TR52B (co 1 sekundę) a potem wchodzi w pętlę nieskończoną. Do realizacji tego zadania są użyte 2 funkcje systemowe: *pulseLEDR* i *waitDelay()*. Funkcja *PulseLEDR* wykonuje jedno mignięcie czerwoną diodą LED modułu TR52B. Czas świecenia się diody jest ustawiany kolejną funkcją systemową *setOnPulsingLED(u8 ticks)*. Ten czas można ustawić z rozdzielczością 10 ms (1 tick) w zakresie od 10 ms (parametr *ticks=1*) do 2550 ms (parametr *ticks=255*). System domyślnie ustala czas świecenie się diod LED na 50 ms. Funkcja *pulseLEDR* inicjuje działanie mignięcia diody, ale odliczanie czasu świecenia odbywa się już w tle programu głównego. Dokładnie tak samo działa bliźniacza funkcja *pulseLEDG* migająca dioda zieloną. 1-sekundowa przerwa pomiędzy błyśnięciami diody jest odliczana przez funkcję systemową *waitDelay(u8 tick)*. Parametr zakres jej nastaw jest taki sam, jak czasów świecenia się diod – od 10 do 2550 ms. Na **listingu 2** pokazano przykład programu ze zmodyfikowaną pętlę nieskończoną, w której jest wykonywane naprzemienne mruganie diodami zieloną i czerwoną.

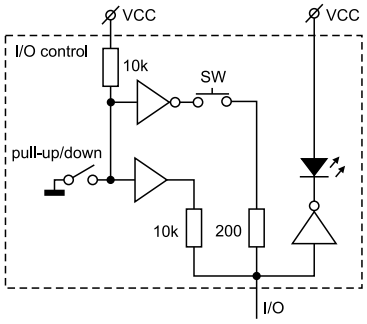
Po dołączeniu modułu DCC-IO-01 z zestawu DCC kits do CK-USB-04 z zamontowanym modułem TR52B, zyskamy możliwość

Listing 2. Zmodyfikowana pętla nieskończona z listingu 1

```
while (1)
{
  pulseLEDG();
  waitDelay(50);
  pulseLEDR();
  waitDelay(50);
  clrwdt();
}
```

Listing 3. Testowanie stany klawiszy

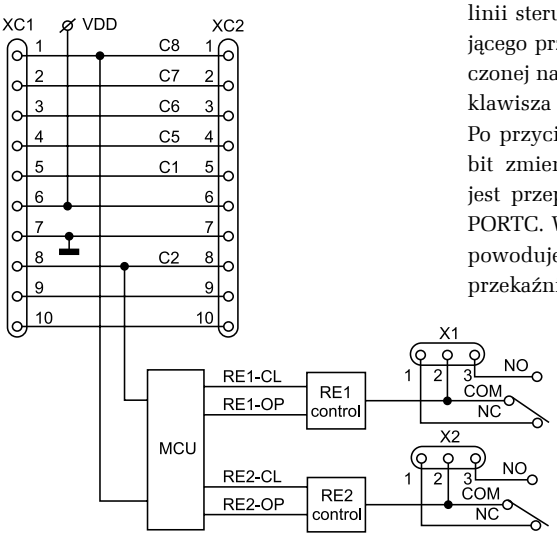
```
//definicja linii portów połączonych z przyciskami
#define SW5_TRIS  TRISA.0
#define SW5             PORTA0
#define SW6_TRIS  TRISC.2
#define SW6             PORTC2
SW5_TRIS =1;  //wejście
SW6_TRIS =1;
while (1)
{
  waitDelay(10);
  clrwdt();
  while(SW5||SW6)       //czekaj na puszczenie klawiszy
  clrwdt();
  waitDelay(10);
  while(SW5==0&&SW6==0) //czekaj na przyciśnięcie klawiszy
  clrwdt();
  waitDelay(10);
  if((SW5)              //przyciśnięty SW5
  {
    pulseLEDG();
    continue;
  }
  if((SW5)              //przyciśnięty SW5
  {
    pulseLEDG();        //miga dioda zielona
    continue;
  }
  if((SW6)              //przyciśnięty SW6
  {
    pulseLEDR();        //miga dioda czerwona
    continue;
  }
  clrwdt();
}
```



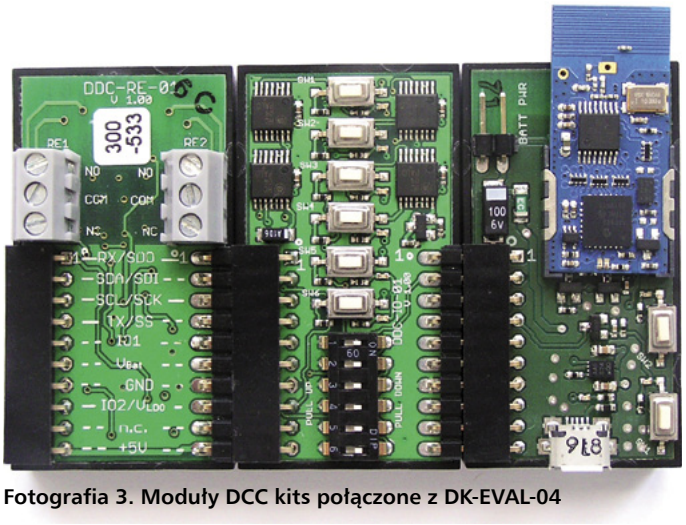
Rysunek 1. Sterowanie zmianą poziomu linii I/O w module DCC-IO-01

odczytywania stanu klawiszy. Układ generowania zmiany stanu po przyciśnięciu klawisza jest rozbudowany, ale dzięki temu ma możliwość zaprogramowania aktywnego poziomu wysokiego lub niskiego (**rysunek 1**). Każdy z 6 torów modułu ma możliwość załączenia podciągania do plusa zasilania lub ściągania do masy dzięki 6-pozycyjnemu przełącznikowi. Zmiana poziomu linii I/O na przeciwny do wybranego na przełączniku następuje po przyciśnięciu przycisku SW. Na **listingu 3** zamieszczono pętlę niekończoną, w której najpierw jest testowany warunek puszczenia klawiszy *C1* i *C2* dołączonych do linii RA0 i RC2 mikrokontrolera PIC18F886 zamontowanego na module TR52B. Jeżeli klawisze są zwolnione, to są na nich poziomy niskie, bo tak skonfigurowałem te linie w module DCC-IO-01. Po zwolnieniu klawiszy program zaczyna sprawdzać warunek przyciśnięcia każdego z nich. Przyciśniecie klawisza SW5 jest sygnalizowane mignięciem diody zielonej, a przyciśniecie klawisza SW6 mignięciem diody czerwonej.

W module przekaźników DDC-RE-01 z zestawu DCC kits zastosowano przekaźniki bistabilne (**rysunek 2**). Jest to rozwiązanie, które zapewnia minimalny pobór mocy, bo prąd płynie przez cewki tylko w momencie przełączania. Wadą takiego rozwiązania jest to, że każdy przekaźnik wymaga 2 linii sterujących. Problem został rozwiązany w ten sposób, że w module DDC-RE-01 dodatkowo zamontowano układ mikrokontrolera sterującego przekaźnikami bistabilnymi. Dzięki temu sterowanie jest realizowane jak w zwykłym przekaźniku. Po połączeniu DCC-RE-01 z modułem TR52B (poprzez CK-USB-04, **fotografia 3**) można sterować przekaźniki liniami C8 (port RC5) i C2 (port(RC2). Uwaga: w wypadku, gdy do komunikacji z hostem jest używany interfejs SPI, linia C8 nie może być używana do sterowania przekaźnikami. Na **listingu 4** pokazano definicje linii sterujących i fragment programu sterującego przekaźnikiem RE2. W pętli nieskończonej najpierw jest testowane przyciśnięcie klawisza SW5 dołączonego do linii PORTA0. Po przyciśnięciu jest negowany najmłodszy bit zmiennej *re*. Potem, wartość tego bitu jest przepisywana do bitu 2 rejestru portu PORTC. W efekcie każde przyciśnięcie SW5 powoduje kolejne załączanie i wyłączanie przekaźnika RE2.



Rysunek 2. Sterowanie przekaźnikami w DDC-RE-01



Fotografia 3. Moduły DCC kits połączone z DK-EVAL-04

Listing 4. testowanie sterowania przekaźnika

```
#define SW5_TRIS  TRISA.0
#define SW5       PORTA0

#define RE1_TRIS  TRISC.5  // C8/sterowanie przekaźnikiem RE1
#define RE1_IO    PORTC.5
#define RE2_TRIS  TRISC.2  // C2/IO2 pin połączony z  RE2
#define RE2       PORTC.2

//******************************************************
void APPLICATION()
{
  uns8  re;
  SW5_TRIS =1;   //wejście
  RE2 = 0;       //przekaźnik wyłączony
  RE2_TRIS = 0;  // wyjście
  re=0;
  while(1)
  {
    while(SW5)   //czekaj na puszczenie klawiszy
    clrwdt();
    waitDelay(10);
    while(SW5==0)//czekaj na przyciśnięcie klawiszy
    clrwdt();
    waitDelay(10);
    if((SW5)
    {            //przyciśnięty SW5
      re=~re;    //przełączenie przekaźnika
      re&=1;
      if(re==0) RE=0; else
      RE=1;
    }
  }
}
```

Przykład 2. Transmisja danych pomiędzy dwoma modułami TR52B

Transmisja danych pomiędzy modułami jest jednym z podstawowych zadań wykonywanych przez system operacyjny IQRF OS.

Przed użyciem moduł radiowy powinien być skonfigurowany. Należy zaprogramować:

- Moc nadajnika – *funkcja setTXpower()*,
- Prędkość transmisji – *funkcja setRFspeed()*,
- Pasmo radiowe – funkcja *setRFband()*,
- Nr kanału radiowego w obrębie wybranego pasma – funkcja *setRFchannel()*,
- Trybu pracy łącza radiowego – *funkcja setRFMode()*.

Moc nadajnika może mieć 8 dyskretnych wartości ustalanych argumentem funkcji *setTXpower()*. Najniższa jest dla argumentu 0, a największa dla argumentu 7. Domyślnie OS ustawia moc na największą i jeżeli dla aplikacji jest to odpowiednie, to można nie wywoływać funkcji *setTxPower()*.

Producent zaleca używanie domyślnej prędkości transmisji 19 kb/s. Dla celów testowych można ją zmienić na: 1,2 kb/s, 57,6 kb/s lub 86,2 kb/s. W tym przykładzie nie będziemy zmieniać wartości domyślnej, więc nie ma potrzeby wywoływania funkcji *setRFspeed()*.

Pasmo radiowe można ustawiać na zakres 868 MHz lub 916 MHz. Domyślnie system wybiera pracę w paśmie 868 MHz. Kanały radiowe wybiera się w taki sposób, by zakłócenia interferencyjne nie zakłócały transmisji danych. W załączniku numer2 dokumentu IQRF OS *User's guide* jest umieszczona mapa dostępnych kanałów w zależności od pasma i prędkości transmisji. Dla pasma 868 MHz i prędkości 19,2 kb/s są dostępne 62 kanały.

Najbardziej rozbudowana funkcją inicjalizacyjną jest setRFmode(mode). Opis znaczenia bitów jej argumentu 8-bitowego zamieszczono w **tabeli 1**.

Przykładowa inicjalizację części radiowej modułu pokazano na **listingu 5**. Moduły radiowe mogą się ze sobą kontaktować bezpośrednio w topologii *peer-to-peer* lub poprzez firmową sieć IQMESH. Do wyboru topologii jest przeznaczony najstarszy bit (_NTWF) rejestru PIN. Kiedy _NTWF jest wyzerowany, to moduły komunikują się *peer-to-peer*, a kiedy jest ustawiony, to jest włączona komunikacja w sieci IQMESH.

Czas, w którym spodziewamy się, że zostanie bez problemu odebrany cały pakiet jest programowany przez zapisanie zmiennej *toutRF*. Jest to zmienna 8 bitowa a jej wartość określa liczbę odliczanych tików o czasie trwania 10 ms każdy, od momentu rozpoczęcia odbioru do momentu jego zakończenia. Wartość wpisana do *toutRF* będzie zależna od liczby przesyłanych danych i prędkości transmisji. Liczba wysyłanych bajtów jest wpisywana do zmiennej *DLEN*.

Odbiór danych można rozpocząć od testowania poziomu sygnału z nadajnika. Jeżeli jest on niższy niż wymagany, to nie powinno się próbować odbierać danych ze względu na możliwość wystąpienia błędów transmisji. Do testowania jest używana funkcja *checkRF()* z argumentem określającym akceptowany poziom sygnału.

Przykładową pętlę odbioru danych pokazano na **listingu 6**. Jeżeli funkcja *RFRXpacket()* zwróci „1", to oznacza, że system operacyjny IQRF OS odebrał w tle odebrano kompletny pakiet danych. Otrzymywane pakiety są sprawdzone przez mechanizmy IQRF OS pod kątem poprawności sumy kontrolnej, preambuły i bajtów startowych. Pole danych zostaje przepisane do bufora *bufferRF*. Po odebraniu pakietu jest aktualizowana wartość zmiennej *DLEN* zawierającej liczbę odebranych danych. Funkcja *copyBufferRF-2COM()* przepisuje wartość bufora odbioru danych z radia *bufferRF* do bufora interfejsu SPI wykorzystywanego do komunikacji z hostem. Wywołanie funkcji *startSPI* z argumentem *DLEN* informuje system operacyjny, że są dane do wysłania przez SPI. Trzeba pamiętać, że moduł radiowy ma interfejs SPI Slave i nie może zainicjować transmisji. Musi to zrobić host wyposażony w interfejs SPI Master.

Tabela 1. Argumenty funkcji setRFMode()

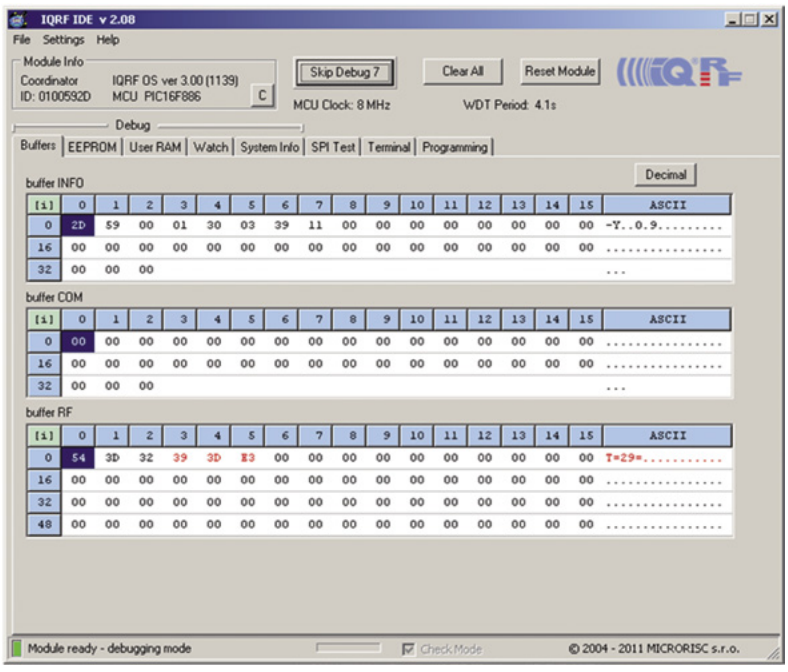
Parametr	Funkcja
bit7 S	Praca w trybie RX 1 – moduł pozostaje w trybie RX po wykonaniu RFRXpacket() i RFTXpacket() 0 – moduł wyłącza tryb RX po wykonaniu RFRXpacket() i RFTXpacket()
Bit 6 W	Czekanie na koniec pakietu 1 – Czekaj na zakończenie odbioru pakietu, nawet jeśli upłynie czas toutRF 0 – RFRXpaket() jest kończona, gdy upłynie czas toutRF
Bity 5,4 TT	Tryb TX 00 – dla trybu STD RX(standardowa preambuła 3 ms). 01 – dla trybu LP RX (wydłużona preambuła 30 ms). 10 – dla trybu XLP RX (wydłużona preambuła 750 ms). 11 – zarezerwowane.
Bity 3,2 FF	Filtrowanie przychodzącego sygnału – określa poziom przychodzącego sygnału, który jest akceptowany. Sygnał o niższym poziomie jest ignorowany. Odpowiednik funkcji checkRF(x). 00 – X=5. 01 – X=20. 10 – X=35. 11 – x=50.
Bity 1,0 RR	Tryb RX 00 – tryb STD RX. 01 – tryb LP RX. 10 – tryb XLP RX. 11 – tryb RFIM (RFRXpacket() jest kończona, gdy poziom sygnału spada poniżej wartości ustawionej przez bity FF.

Listing 5. Inicjalizacja części radiowej

```
setRFband(0);      //  wybranie pasma 868 MHz
setRFspeed(3);     // wybranie prędkości 57.6 kb/s bit
setRFchannel(25); // wybranie kanału w paśmie 868MHz
setRFMode(0);      //tryb pracy
toutRF=10;         //time out na 100msek
PIN=0;             //tryb peer-to-peer
```

Listing 6. Pętla odbioru danych przez moduł radiowy

```
while(1)
{
  if(checkRF(5))      //poziom sygnału akceptowalny
  {
    if(RFRXpacket()   //pakiet został odebrany
    {
      pulseLEDR();    //sygnalizacja odbioru
      copyBufferRF2COM(); //odebrane dane są kopiowane
                          //z bufferRF do bufferCOM
      startSPI(DLEN);
    }
  }
.....
 Fragment programu wysyłający odebrane dane po SPI
.....
}
```



Rysunek 4. Okno debugera po odebraniu pakietu z pomiarem temperatury

Listing 7. Przykład wysyłania danych przez radio
```
appInfo();            //dane aplikacji do bufora bufferINFO
copyBufferINFO2RF()l  //kopiuj bufferINFO do bufferRF
setNonetMode();       // wyłącz tryb sieciowy
//PIN=0;              //alternatywnie do setNonetMode
DLEN=10;              //do wysłania 10 bajtów
RFTXpacket();         //wysłanie pakietu
```

Listing 8. Pomiar, konwersja i wysłanie temperatury
```
  uns16 temperature;
  getTemperature();  //odczytaj temperaturę
  temperature.high8=ADRESH & 0x03; //"sklejanie" 10-bitowej liczby
  temperature.low8=ADRESL;
  temperature*=75;   //konwersja napięcie/temperatura
  temperature>>=8;
  temperature-=50;
  bufferRF[0]=temperature.low8/10;//konwersja dotąd
  bufferRF[0]+=0x30;
  bufferRF[1]=temperature.low8%10;
  bufferRF[1]+=0x30;
  ufferRF[0]='T';
  bufferRF[2]='='
  PIN=0;//peer-to-peer
  DLEN=4;//4 bajty danych
  RFTXpacket();      //wysłanie pakietu danych
```

Komunikacja moduł –host przez SPI będzie opisana dokładniej w dalszej części artykułu.

Nadawanie danych również nie jest skomplikowane. Trzeba wpisać dane do bufora bufferRF, zapisać w zmiennej DLEN liczbę wysyłanych bajtów, wyzerować (dla konfiguracji *peer-to-peer*) rejestr PIN i wywołać funkcje nadawania *RTFXpacket();*. Na **listingu 7** pokazano przykład wysłania przez łącze radiowe danych aplikacji odczytanych za pomocą funkcji *appInfo()*.

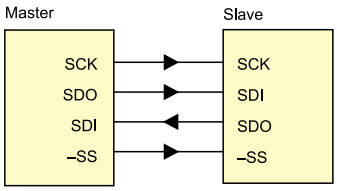
Przykład 3 odczytywanie i przesyłanie temperatury z modułu TR52B

Moduł TR52B jest wyposażony w czujnik temperatury MCP9700, w którym mierzona temperatura jest konwertowana na napięcie. To napięcie jest podawane na wejście RA3/AN3 mikrokontrolera i można je mierzyć wbudowanym przetwornikiem analogowo cyfrowym. Nic nie stoi na przeszkodzie, aby samemu skonfigurować przetwornik i wykonać pomiar, ale dużo wygodniej jest to zrobić używając funkcji systemowej *getTemperature()*. Po jej wywołaniu do rejestrów *ADRESH* i *ADRESL* jest wpisywana liczba 10-bitowa z przetwornika analogowo cyfrowego. Użytkownik musi w swoim programie wykonać konwersję tej liczby 10-bitowej na mierzoną temperaturę – najlepiej w postaci dwóch znaków ASCII z liczbami dziesiątek i jedności. Konwersje wykonuje się zgodnie z następującym wzorem:

Ta=ADRES*75/256 – 50,

gdzie ***ADRES=(ADRESH&3):ADRESL***.

Na **listingu 8** zamieszczono fragment programu odczytujący temperaturę i konwertujący ją na 2 znaki ASCII wpisywane do bufora *bufferRF*. Po skompletowaniu pakietu w buforze jest on wysyłany przez łącze radiowe.



Rysunek 5. Magistrala SPI

Do testowania łącza radiowego i pomiaru temperatury napisałem dwa programy: jeden dla TR52B podłączonego do płytki ewaluacyjnej CK-USB-04, a drugi dla TR52B mierzącego temperaturę i zasilanego z baterii poprzez moduł DK-EVAL-04. Do testowania działania użyłem funkcji *debug()* zaimplementowanej w programie modułu TR52B połączonego z płytką ewaluacyjnym CK-USB-04. Na **listingu 9** pokazano program z funkcją *debug()*. Drugi program jest identyczny, ale bez tej funkcji.

Najpierw jest zerowany bufor bufferRF i OS jest programowany na działanie peer-to-peer. Potem program odczytuje swój czujnik temperatury i po konwersji wysyła jej wartość przez łącze radiowe. W nieskończonej pętli sprawdzany jest warunek odebrania pakietu. Jeżeli pakiet zostanie odebrany, to program interpretuje to jako żądanie przesłania temperatury dlatego mierzy temperaturę i odsyła jej wartość. Pierwsze wysłanie temperatury przed pętlą nieskończoną służy do zainicjowania wymiany danych. Tak zaprogramowane dwa moduły przesyłają nawzajem do siebie zmierzone temperatury.

Jak już wspomniałem do testowania użyłem funkcji *debug ()* umieszczonej w pętli nieskończonej zaraz po odebraniu pakietu przez funkcję *RFRXpacket()*. Na **rysunku 4** pokazano okno *Buffers* debugera pakietu IQRF IDE z wpisanymi danymi z pomiaru temperatury do bufora *bufferRF*. Pierwsze 4 bajty bufora *bufferRF* zawierają ciąg znaków ASCII „T=29", bo temperatura zmierzona przez moduł wynosi 29°C.

Przykład4. Komunikacja z hostem przez interfejs SPI

Komunikacja modułu radiowego z zewnętrznym hostem odbywa się przez szeregowy interfejs SPI. Interfejs SPI jest transmituje dane w konfiguracji master-slave. Składa się z 4 linii:

- zegarowej SCK (sygnał zegarowy jest generowany przez układ master),
- danych odbieranych SDI,
- danych nadawanych SDO,
- linii aktywacji interfejsu SS (obsługiwanej przez master).

Do jednego układu master można dołączyć wiele układów slave. Ten rodzaj protokołu szeregowego nie zawiera adresowania, więc w czasie transmisji danych sygnał sterujący powinien wybrać układ, do którego jest adresowana transmisja. Teoretycznie może być wybrany (linią SSx) tylko jeden z układów slave, jednak w praktyce spotyka się rozwiązania, w których transmisja jest wykonywana do kilku identycznych układów slave. Na **rysunku 5** pokazano połączenie linii interfejsu dla pojedynczego układu slave. Ze względu na oddzielne linie danych wejściowych i wyjściowych, transmisja SPI może być wykonywana w trybie full duplex.

W czasie wymiany danych używa się dwóch rejestrów przesuwnych i dowolnej

Listing 9. program mierzący i wysyłający temperaturę
```
void APPLICATION()
{
  setNonetMode();//wyłącz tryb sieciowy
  getTemperature();
  temperature.high8=ADRESH & 0x03;
  temperature.low8=ADRESL;
  temperature*=75;
  temperature>>=8;
  temperature-=50;
  bufferRF[2]=temperature.low8/10;
  bufferRF[2]+=0x30;
  bufferRF[3]=temperature.low8%10;
  bufferRF[3]+=0x30;
  bufferRF[0]='T';
  bufferRF[1]='=';
  PIN=0;//peer-to-peer
  DLEN=4;//2 bajty danych
  RFTXpacket();
  pulseLEDG();
  //petla nieskończona odczytu temperatury
  while(1){
    clrwdt();
    if(RFRXpacket()){
      debug();
      waitDelay(250);//czekaj 2,5sek
      pulseLEDR();
      getTemperature();
      temperature.high8=ADRESH & 0x03;
      temperature.low8=ADRESL;
      temperature*=75;
      temperature>>=8;
      temperature-=50;
      bufferRF[2]=temperature.low8/10;
      bufferRF[2]+=0x30;
      bufferRF[3]=temperature.low8%10;
      bufferRF[3]+=0x30;
      bufferRF[0]='T';
      bufferRF[1]='=';
      PIN=0;//peer-to-peer
      DLEN=4;//2 bajty danych
      RFTXpacket();
  }
}
```

Tabela 2. Statusy modułu zwracane przez IQRF IDE

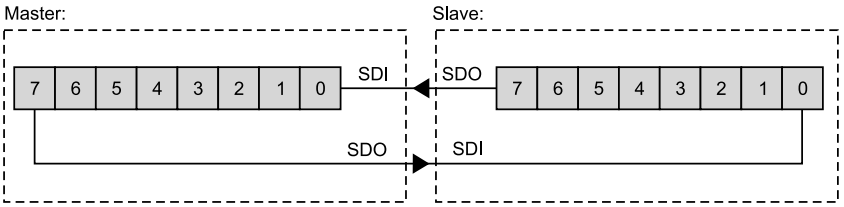
Wartość	Opis
00	SPI nieaktywne (wyłączone przez komendę disableSPI())
07	SPI zatrzymane przez komendę stopSPI()
3F	SPI niegotowe (pełny bufor, ostatnie CRC prawidłowe). Dalsza transmisja może być odblokowywana funkcją startSPI(0)
3E	SPI niegotowe (pełny bufor, ostatnie CRC nieprawidłowe). Dalsza transmisja może być odblokowywana funkcją startSPI(0)
41 do 40+Nmax	SPI gotowe do przesłania tego statusu – 0x40 bajtów
80	SPI gotowe – tryb komunikacji
81	SPI gotowe – tryb programowania
82	SPI gotowe – tryb debugowania
83	Zarezerwowane dla starszych wersji modułów radiowych
FF	SPI nieaktywne – błąd sprzętowy

długości, ale najczęściej są to rejestry 8- lub 16-bitowe. Połączone rejestry układów master i slave tworzą bufor kołowy (**rysunek 6**).

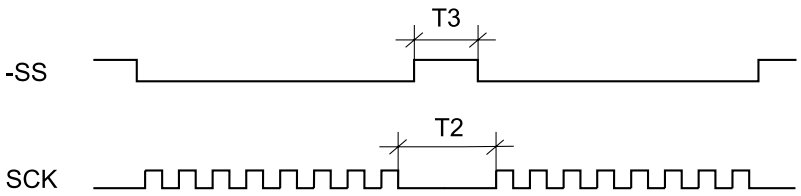
Aby odczytać bajt z układu slave, master musi do niego wysłać 8 cykli zegarowych, czyli również jeden bajt. Dane są przesuwane od najstarszego do najmłodszego bitu w czasie narastającego zbocza SCK. Wiemy, że zaimplementowany w module TR52B system operacyjny IQRF OS obsługuje interfejs SPI w trybie slave. Masterem musi być zewnętrzny mikrokontroler. OS nakłada pewne ograniczenia czasowe na transmisję. Mogą być one istotne, gdy używany szybkiego mikrokontrolera. Zegar SCK może mieć maksymalną częstotliwość 250 kHz, a po uaktywnieniu linii SS (aktywny jest poziom niski) trzeba odczekać 10 µs. Pomiędzy przesłaniem 2 kolejnych bajtów trzeba odliczyć opóźnienie 100 µs (**rysunek 7**).

Obsługa SPI w IQRF OS jest zaimplementowana jako maszyna stanu. Master może w każdej chwili zapytać o status wysyłając przez SPI komendę *SPI_CHECK*. Program użytkownika uruchomiony w module może również zapytać o status własnego interfejsu używając funkcji *getStatusSPI()*. Kiedy moduł radiowy jest połączony płytką ewaluacyjną (np. CK-USB-04), to jego status jest odczytywany w programie IQRF IDE, jeżeli pole *Check Mode* jest zaznaczone (**rysunek 8**). Zawsze po wysłaniu komendy *SPI_CHECK* (bajt 0x00) moduł odpowiada bajtem statusu, którego możliwe wartości umieszczono w **tabeli 2**.

Rysunek 8. Status modułu w IQRF IDE



Rysunek 6. Wymiana danych przez SPI



Rysunek 7. Ograniczenia czasowe w transmisji SPI: $T2_{MIN}$ – 100 µs, $T3_{MIN}$ – 20 µs

B7	B6	B5	B4	B3	B2	B1	B0
CTYPE		SPIDLEN					

CTYPE:
B7=1 bufferCOM się zmienia
B7=0 bufferCOM się nie zmienia
SPIDLEN – ilość przesyłanych danych od 1 do Nmax

Rysunek 9. Format bajtu PTYPE

Drugim wyróżnionym bajtem po *SPI_CHECK* jest *SPI_CMD* (0xF0). Wysłanie do modułu *SPI_CMD* powoduje, że OS interpretuje otrzymywany ciąg znaków jako komendę, na którą musi odpowiedzieć. Po bajcie komendy *SPI_CMD* jest wysyłany bajt PTYPE (**rysunek 9**).

Jeśli bit B7 w polu CTYPE jest ustawiony, to dane przesyłane przez Master są odczytywane i zapisywane przez slave. Wtedy możliwe są dwie sytuacje:

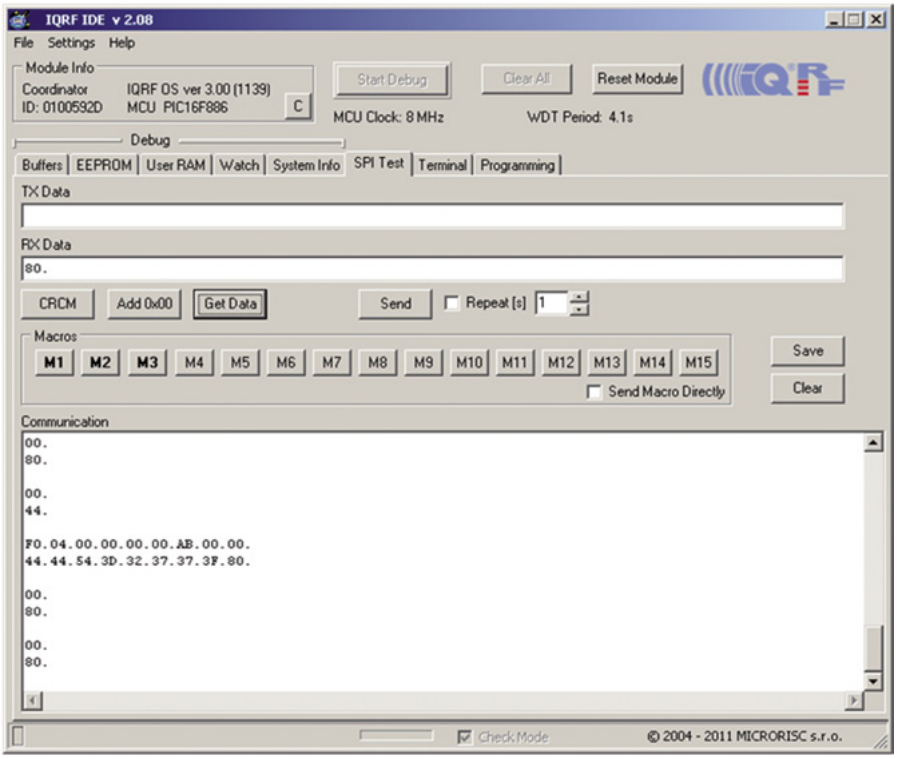
– master przesyła ważne dane do slave; oczywiście, slave w tym czasie przesyła dane do mastera, ale master je ignoruje.
– master przesyła ważne dane do slave i jednocześnie odbiera dane ze slave.

Wyzerowanie bitu B7 oznacza, że master wysyła dane nieistotne dla slave (i są one ignorowane) tylko po to, aby wygenerować sygnał zegarowy i odczytać dane ze slave. Najczęściej są to same bity zerowe. Co ważne, funkcje systemowe nie sygnalizują faktu odebrania danych.

Przy wysyłaniu bajtów *SPI_CMD* i *PTYPE* moduł zawsze odpowiada bajtem statu-

Master	SPI_CMD	PTYPE	DM_1	DM_2	---	$DM_{SPIDLEN}$	CRCM
Slave	SPISTAT	SPISTAT	DS_1	DS_2	---	$DS_{SPIDLEN}$	CRCS

Rysunek 10. Wymiana danych dla komendy SPI_CMD



Rysunek 11. Debugowanie transmisji SPI

Listing 10. Pomiar temperatury i przesłanie prze SPI

```
void APPLICATION()
{
uns8 i;
  for(i=0;i<20;i++) bufferRF[i]=0;
  setNonetMode();//wyłącz tryb sieciowy
  enableSPI();
  startSPI(0);
  getTemperature();
  temperature.high8=ADRESH & 0x03;
  temperature.low8=ADRESL;
  temperature*=75;
  temperature>>=8;
  temperature-=50;
  bufferRF[2]=temperature.low8/10;
  bufferRF[2]+=0x30;
  bufferRF[3]=temperature.low8%10;
  bufferRF[3]+=0x30;
  bufferRF[0]='T';
  bufferRF[1]='=';
  PIN=0;//peer-to-peer
  DLEN=4;//2 bajty danych
  RFTXpacket();
  pulseLEDG();
  while(1){
    clrwdt();
    if(RFRXpacket()){
      //debug();
      waitDelay(250);
      pulseLEDR();
      getTemperature();
      temperature.high8=ADRESH & 0x03;
      temperature.low8=ADRESL;
      temperature*=75;
      temperature>>=8;
      temperature-=50;
      bufferRF[2]=temperature.low8/10;
      bufferRF[2]+=0x30;
      bufferRF[3]=temperature.low8%10;
      bufferRF[3]+=0x30;
      bufferRF[0]='T';
      bufferRF[1]='=';
      PIN=0;//peer-to-peer
      DLEN=4;//2 bajty danych
      RFTXpacket();
      copyBufferRF2COM();//przepisz odczytaną temperaturę z RF do COM
      startSPI(4);//próbuj wysłać przez SPI
      clrwdt();
    }
  }
}
```

Listing 11. Pętla oczekiwania na dane z hosta

```
while(1){
  clrwdt();
  if(getStatusSPI()){
    pulseLEDG();
    continue;
  }
  if(_SPIRX){ //odebrano dane z hosta
  pulseLEDG();
  if(!_SPICRCok){//błąd CRCM
  startSPI(0); pulseLEDR();
  continue;}
  else
  {
    waitDelay(35);
    break;}
  }
}
```

sowym. Ramka komendy musi kończyć się prawidłowo wyliczoną sumą kontrolną CRC według zależności:

```
SPI_CMD xor PTYPE xor DM1 xor DM2
……. xor DM SPIDLEN xor 0x5F.
```

Wymiana danych odbywa się pod kontrolą układu master. Układ slave nie może inicjować transmisji, a jedynie odpowiedzieć swoim statusem. Aby dane z modułu radiowego mogły być przesłane do hosta, to ten musi cyklicznie wysyłać zapytanie o status *SPI_CHECK*. Producent zaleca, aby odbywało się to co 10 ms. Sam system operacyjny IQRF IDE pyta o status co 400 ms. Kiedy zawartość statusu informuje, że są dane do odczytania, to master musi skompletować i wysłać ramkę odczytująca te dane ze slave. Jeżeli moduł ma do wysłania na przykład 3 bajty danych, to wykonuje funkcję *startSPI(3)*. Wtedy na zapytanie o status wysłane przez host zostanie odesłana odpowiedź 0x43 (**rysunek 10**).

Gdy master (host) chce przesłać dane do modułu slave, to w pierwszej kolejności musi odczytać status 0x80. W przeciwnym razie może okazać się, że dane wysłane poprzednio nie zostały jeszcze przez aplikacje użytkownika odczytane z *bufferCOM*. Testowanie statusu zapobiega napisaniu bufora *bufferCOM* nowymi danymi, gdy poprzednie jeszcze nie zostały odczytane.

Konfiguracja sprzętowa jest identyczna, jak w wypadku z poprzedniego przykładu. Również oprogramowanie modułu zasilanego bateryjnie i mierzącego temperaturę nie ulega zmianie. W tym przykładzie, po odebraniu przez TR52B połączony z CK-USB-02 temperatury przesłanej drogą radiową prześlemy ją przez interfejs SPI do hosta. Jego rolę będzie pełnił CK-USB-02, a do testowania połączenia przez SPI wykorzystamy zakładkę SPI Test pakietu IQRF IDE. Na **listingu 10** pokazano zmodyfikowany program do pomiaru temperatury.

Przed transmisją danych należy włączyć interfejs SPI za pomocą funkcji *enableSPI()*. Jeżeli tego nie zrobimy, na każde zapytanie o status OS będzie odpowiadał bajtem 0x00 informującym, że interfejs SPI jest nieaktywny. Po przesłaniu temperatury zawartość bufora *bufferRF* jest kopiowana do bufora *bufferCOM* przez funkcję *copyBufferRF2COM()*. Dalej, przesłanie pierwszych czterech bajtów bufora *bufferCOM* do hosta zostanie zlecone systemowi operacyjnemu za pomocą wywołania funkcji *startSPI(4)*. Nie oznacza to, że dane zostaną przesłane do hosta, bo jak wiemy, moduł pełni rolę interfejsu slave. Jednak po wysyłaniu *startSPI(4)* moduł zacznie odpowiadać statusem 0x44 na zapytanie o status przez układ mastera. Host (master), po odczytaniu takiego statusu musi skompletować komendę *SPI_CMD* o takiej długości, aby odczytać 4 bajty danych. Przebieg transmisji dla działającego programu z list. 6 pokazano na **rysunku 11**. Najpierw host, którym jest CK-USB-04 wysyła bajt 0x00 pytając o status. Moduł odpowiada bajtem 0x80 informującym, że interfejs SPI jest aktywny, ale nie ma nic do przesłania. Na kolejne zapytanie o status moduł opowiedział bajtem 0x44. Jest to informacja dla hosta, że moduł ma do przesłania 4 bajty. Aby je odczytać host wysyła komendę *SPI_CMD* z bajtem *PTYPE=04*, czterema bajtami zerowymi, sumą CRC i dwoma bajtami zerowymi. *PTYPE=04* jest informacją dla modułu, że należy zignorować bajty przesyłane w polu danych. W czasie przesyłania zerowych bajtów moduł transmituje zawartość swojego bufora *bufferCOM*, a w czasie przesyłania CRC moduł odsyła swoje CRC. Wysłanie po CRC 2 bajtów zerowych pozwala na odczytanie 2 bajtów statusu po transmisji. Można uznać, że transmisja zakończyła się poprawnie, jeżeli moduł na zapytanie odpowiada bajtem 0x80.

Program przedstawiony na list. 10 ma pewną wadę. Kopiuje zawartość *bufferRF* do *bufferCOM* bez potwierdzenia, że poprzednie dane zostały przeczytane. Można temu zaradzić wstrzymując odczytanie następnego pomiaru do momentu odczytania poprzedniego. Na **listingu 11** pokazano fragment programu, który czeka na odebranie w tle prawidłowego pakietu z komendą *SPI_CMD*. Trzeba też pamiętać, że bajt *PTYPE* musi mieć ustawiony najstarszy bit, bo w przeciwnym wypadku IQRF OS nie potraktuje go jako pakiet, który zawiera ważne dane i go zignoruje. Jak wiemy, przy wyzerowanym najstarszym bicie *PTYPE* pakiet wysłany przez hosta i odebrany przez TR52B służy tylko do wymuszenie przesłania danych przez układ slave.

Przykład 5 Moduł DCC-SE-01 Odczytywanie temperatury

Moduł DCC-SE-01 zawiera:

- czujnik temperatury z interfejsem I²C typu MCP9802 A3,
- czujnik temperatury DS18B20 z interfejsem 1-wire,
- czujnik natężenia światła z fotorezystorem,
- układ regulacji napięcia wejściowego z potencjometrem.

Ponadto, na płytce jest umieszczony stabilizator LDO i złącze magistrali I²C dla układów zewnętrznych. Jego schemat ideowy pokazano na **rysunku 12**. Do połączenia elementów DCC-SE-01 z modułem TR52B użyto wszystkich linii I/O: C1, C5, C6, C7 i C8. Po połączeniu modułu sensorów zajęte są linie interfejsu SPI i nie można się komunikować z hostem zewnętrznym. Najczęściej nie jest to problemem dla zdalnych modu-

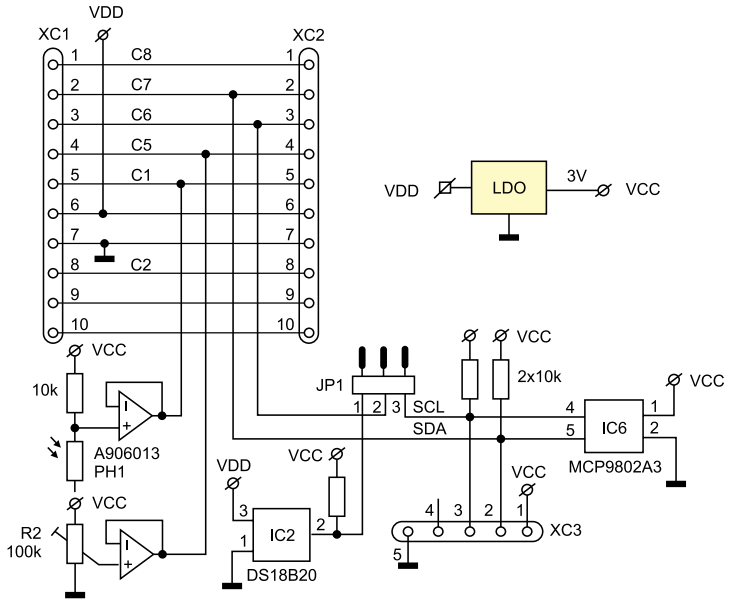
łów, bo większości wypadków i tak będą pracować autonomicznie i nie wymagają hosta. W tym przykładzie zajmiemy się odczytywaniem temperatury z DS18B20. Jest to znany i popularny czujnik mierzący temperaturę w szerokim zakresie, z dość dobrą dokładnością. Jego zaletą jest, że do komunikacja i zasilanie mogą odbywać się z użyciem pojedynczego przewodu. Transmisja 1-wire przebiega w konfiguracji master-slave. Układem master jest najczęściej mikrokontroler, a układami slave dołączane do niego układy – tu termometr DS18B20.

Na wymianę informacji poprzez magistralę 1-Wire składają się z cztery sekwencje:
- inicjalizacji (*reset and presence pulses*),
- zapisu zera,
- zapisu jedynki,
- odczyt bitu.

Gdy nie jest wykonywana żadna z sekwencji linia DQ musi być na poziomie wysokim (stan *idle*). Każda wymiana informacji z termometrem DS18B20 zaczyna się sekwencją inicjalizacji, wyzerowaniem przez mikrokontroler (master) linii DQ. Nazywa się to impulsem zerowania. Czas jego trwania wynosi nie mniej niż 480 µs (**rysunek 13**). Po zakończeniu impulsu zerowania linia portu mikrokontrolera przechodzi w stan wysokiej impedancji, a na linii DQ pojawia się stan wysoki wymuszony zewnętrznym rezystorem łączącym linię DQ z plusem zasilania. Jeżeli do magistrali dołączony jest układ DS18B20, to powinien on po wykryciu stanu wysokiego następującego po impulsie zerowania wymusić poziom niski trwający od 60 do 240 µs (*presence pulse*). To informacja dla układu master, że do magistrali jest dołączony układ slave i można przesłać dane.

Do omawiania obsługi magistrali 1-wire zastosowałem przykłady umieszczone w pliku DCC-SE-01-*dallas.c*. Ten plik można znaleźć w katalogu example\DCC na płycie CD dostarczanej z zestawem ewaluacyjnym.

Jak wynika ze schematu DCC-SE-01, linia DQ jest połączona (poprzez zworkę JP1) z wyprowadzeniem C6 (linia RC3). Na **listingu 12** pokazano procedurę inicjowania magistrali 1-wire . Funkcja zwraca wartość „1”, jeśli impuls *presence* zostanie wykryty lub „0” w przeciwnym wypadku. Sekwencję zapisywania logicznego „0” przez mikrokontroler do termometru pokazano na **rysunku 14**. Mikrokontroler wymusza poziom niski na magistrali przez co najmniej 60 µs, ale nie dłużej niż przez 120 µs, bo układ termometru może to zinterpretować jako sekwencję zerowania. Sekwencja zapisywania logicznej „1” (**rysunek 15**) rozpoczyna się od wymuszenia



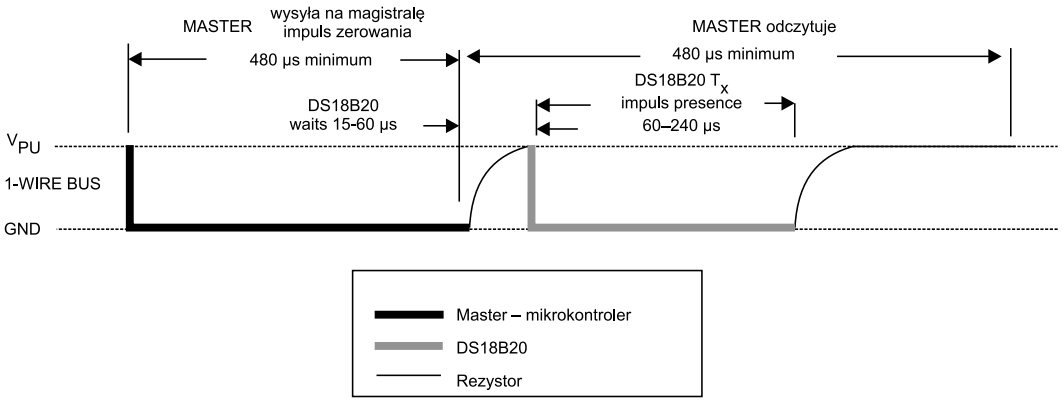
Rysunek 12. Schemat ideowy DCC-SE-01

Listing 12. Inicjalizacja magistrali

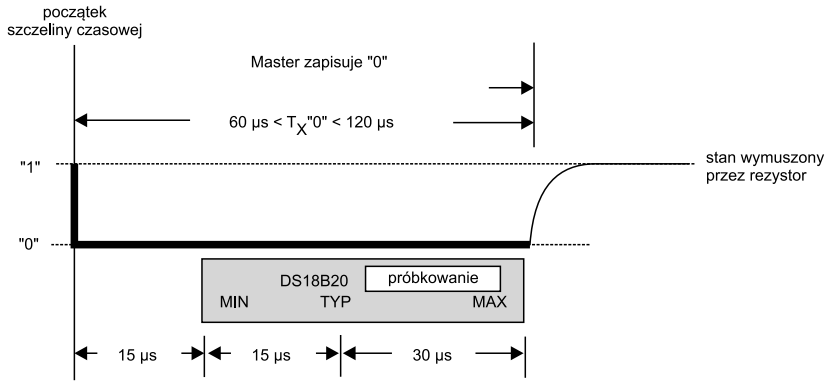
```
#define DQ_TRIS  TRISC.3    // pin DS18B20 (pin DQ)
#define DQ_IO  PORTC.3

bit  OW_Reset(void)
{
  bit result;
  uns8 i;
  DQ_TRIS = 0;              // DQ wyjściowa
  DQ_IO = 0;
  waitMS(1);                // Impuls zerowania  (min. 480us)
  DQ_TRIS = 1;              // DQ wejściowa
  waitDelay_10us(7);        // opóźnienie  70us
  if (DQ_IO)  result = 0;
  else        result = 1;   // impuls Presence został wykryty

  waitDelay_10us(50);       //opóźnienie 500us (min. 410us)
  return(result);
}
```

Listing 13. zapis danej na magistrali 1-wire

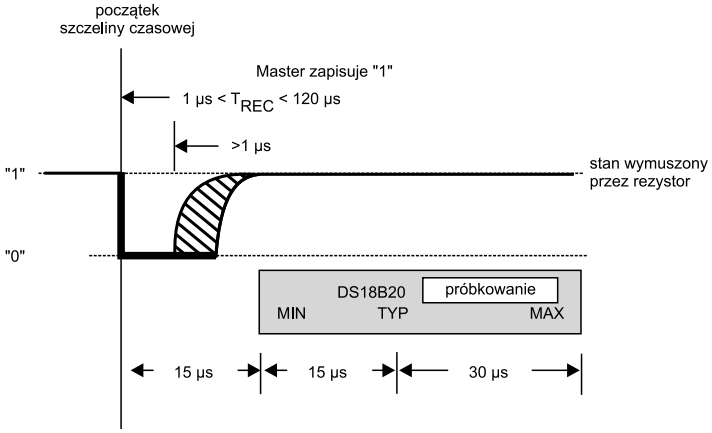
```
void OW_WriteBit(uns8 val)
{
    DQ_TRIS = 0;                        // DQ wyjściowa
    DQ_IO = 0;

    if (val)                           //zapisujemy jedynkę na magistralę
    {
        nop();                              // opóźnienie 6us
        nop();
        nop();
        nop();
        DQ_TRIS = 1;                        // DQ wejściowa
        waitDelay_10us(6);                 // opóźnienie 600usek
    }
    Else                               //zapisujemy zero na magistral e
    {
        waitDelay_10us(6);                                //opóźnienie 600usek
        DQ_TRIS = 1;                       // Input
        waitDelay_10us(1);
    }
}
```



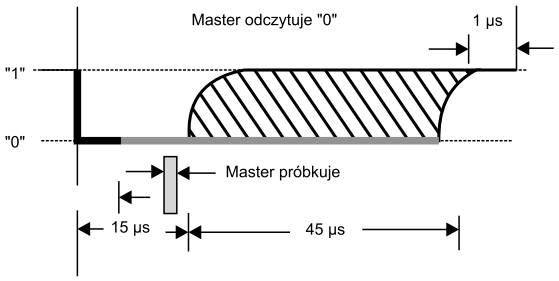
Rysunek 13. Sekwencja inicjowania magistrali 1-wire



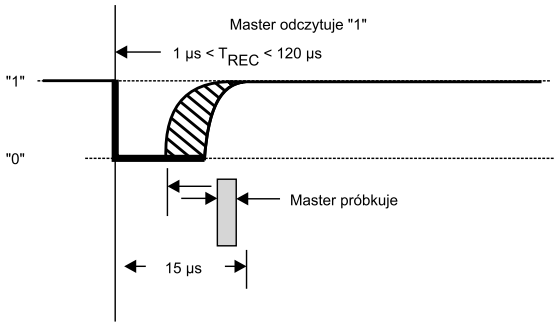
Rysunek 14. Zapisywanie zera logicznego



Rysunek 15. Zapisywanie jedynki logicznej



Rysunek 16. Odczytanie zera logicznego



Rysunek 17. Odczytanie jedynki logicznej

Listing 14. odczytanie bitu z magistrali

```
bit OW_ReadBit(void)
{
bit result;

  DQ_TRIS = 0;     // DQ wyjscie
  DQ_IO = 0;
  nop();           // czekaj  6us
  nop();
  nop();
  nop();
  DQ_TRIS = 1;     // DQ wejście
  nop();           // czekaj  6us
  nop();
  nop();
  nop();
  nop();
  nop();
  nop();
  if (DQ_IO)  result = 1;
  else        result = 0;
  waitDelay_10us(6);  // czekaj 600us
    return (result);
}
```

Odczytywanie danych rozpoczyna się od wymuszenia przez master na magistrali poziomu niskiego przez minimum 1 µs. Dane wysyłane przez DS18B20 są dostępne przez nie dłużej niż 15 µs i przed upływem tego czasu mikrokontroler musi je przeczytać. Jeżeli na magistrali jest poziom niski, to odczytane zostaje „0” (**rysunek 16**), a jeżeli wysoki, to „1” (**rysunek 17**). Przykład funkcji odczytującej pojedynczy bit z magistrali 1-wire zamieszczono na **listingu 14** (*OW_readBit()*). Procedury odbioru i wysyłania bitu na magistralę zastosujemy do napisania

Listing 15. Procedury zapisania o odczytania bajtu z DS18B20

```
void OW_WriteByte(uns8 val)  //zapisanie bajtu do DS18B20
{
  uns8 i;
  for (i = 0; i < 8; i++)  // wysłanie 8 bitów , pierwszy LSB
  {
    OW_WriteBit(val & 0x01);
    val >>= 1;
  }
}

uns8 OW_ReadByte(void)  //odczytanie bajtu z DS18B20
{
  uns8 i, result;
  result = 0;
  for (i = 0; i < 8; i++)
 {
    result >>= 1;
    if (OW_ReadBit())
    result |= 0x80;
  }
  return result;
}
```

funkcji odczytywania i zapisywania bajtów poprzez 1-wire (**listing 15**).

poziomu niskiego na magistrali przez nie dłużej niż 15 µs. Po tym czasie na magistrali musi się pojawić poziom wysoki trwający co najmniej 45 µs. Na **listingu 13** pokazano procedurę zapisania bitu na magistrali.

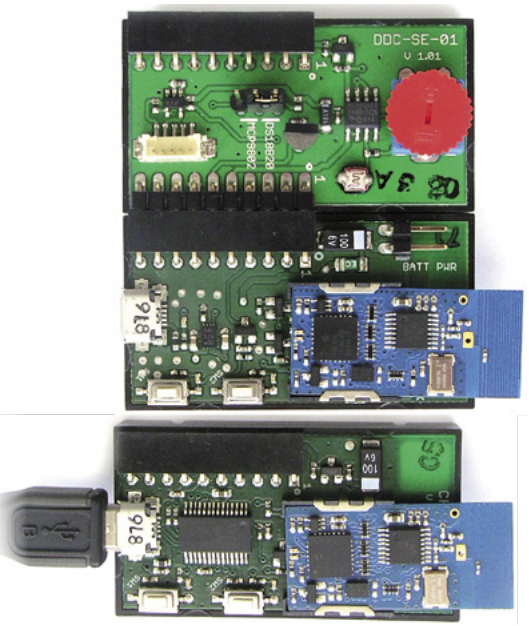
Ponieważ specyfikacja magistrali i układu DS18B20 dopuszcza dołączenie wielu termometrów do jednej linii DQ, to sterowanie termometrem wymaga wysyłana do niego szeregu komend sterujących. Dokładne informacje Czytelnik może znaleźć w dokumentacji technicznej układu. Tutaj skupimy się tylko na możliwości odczytania temperatury z jednego termometru. Do tego celu zostanie wykorzystana procedura *DS._GetTemperature()* pokazana na **listin-**

gu 16. Jedyne na co należy zwrócić uwagę, to konieczność zablokowania wszystkich przerwań w trakcie wykonywania pomiaru. Jest to konieczne, bo magistrala 1-wire wymaga odmierzania stosunkowo krótkich czasów rzędu pojedynczych mikrosekund, a zgłaszanie przerwań na pewno zaburzy tę funkcjonalność.

Widok kompletu komponentów użytych w teście pokazano na **rysunku 18**. Do testowania przesyłania temperatury użyjemy dwóch programów dostarczonych wraz z zestawem. Pierwszym będzie wspomniany *DCC-SE-01-dallas.c*. Po skompilowaniu wgrywamy go do modułu TR52B i umieszczamy w gnieździe karty SIM modułu DK-EVAL-04. Ten moduł łączymy z DCC-SE-04 i włączamy zasilanie zworką BATT PWR. Zasilony moduł radiowy mierzy i wysyła droga radiowa temperaturę z układu DS18B20 co około 5 s. Każde wysłanie temperatury jest sygnalizowane mrugnięciem czerwonej diody LED na module TR52B.

Drugi program to również firmowy *E02-RX.c* pokazany na **listingu 17**. Po odblokowaniu interfejsu SPI w nieskończonej pętli jest sprawdzany warunek odebrania pakietu danych przez radio. Jeżeli pakiet zostanie odebrany, to mrugnie czerwona dioda LED i zawartość *bufferRF* jest kopiowana do *bufferCOM*. Na końcu zawartość *bufferCOM* jest wysyłana przez interfejs SPI. Ta ostatnia czynność pozwala na podglądanie zwartości *bufferCOM* w zakładce *Terminal IQRF IDE* (**rysunek 19**). Oczywiście, warunkiem jest dołączenie DK-USB-04 przez łącze USB do komputera.

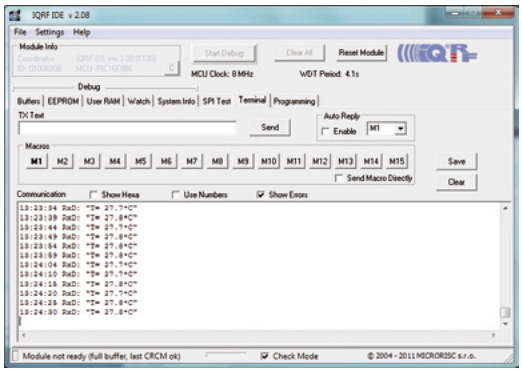
W pliku *DCC-SE-01-i2c.c* jest zawarty kompletny program odczytywania drugiego termometru MCP9802 A3 umieszczonego w module DCC-SE-01. MCP9802 A3 komunikuje się z mikrokontrolerem za pomocą magistrali I²C. Przestawienie zworki z pozycji DS18B20 w pozycję MCP9802 powoduje odłączenie linii DQ od termometru DS18B20 i połączenie SCL magistrali I²C z linią RC3. Wgranie tego programu do modułu zasilanego bateryjnie powoduje przesyłanie temperatury z tego czujnika dokładnie tak samo, jak w wypadku DS18B20. Program obsługujący termometr korzysta ze sprzętowego interfejsu



Fotografia 18. Zestaw do testowania przesyłania temperatury z czujnika DS18B20

Listing 16. odczytanie temperatury z układu
```
// DS18B20 commands
#define DS_SKIP_ROM     0xCC  // skip ROM
#define DS_CONVERT_T    0x44  // start temperature conversion
#define DS_READ_RAM     0xBE  // read the scratchpad
it DS_GetTemperature(void)
{
  GIE = 0;                    // wyłączenie wszystkich przerwań
  if (!OW_Reset())            //test na obecność układu
  goto SENSOR_NOT_FOUND;
  OW_WriteByte(DS_SKIP_ROM);  //niezbędne komendy sterujące
  OW_WriteByte(DS_CONVERT_T); // Start pomiaru
  while (!OW_ReadBit());      // czekaj na zakończenie pomiaru
  if (!OW_Reset())
  goto SENSOR_NOT_FOUND;
  OW_WriteByte(DS_SKIP_ROM);  //komenda sterująca
  OW_WriteByte(DS_READ_RAM);
  temperature.low8 = OW_ReadByte(); // odczyt rejestru temperatury
  temperature.high8 = OW_ReadByte();
  OW_Reset();                 // koniec odczytu bufora RAM
  GIE = 1;                    // Odblokowanie przerwań i powrót z prawidłowym
                              // wynikiem
  return 1;
SENSOR_NOT_FOUND:
  GIE = 1;                    // Odblokowanie przerwań i powrót z sygnalizacją
                              // błedu
  return 0;}
```

Listing 17. Program E02-RX.c
```
void APPLICATION() {
  enableSPI();              // Enable SPI
  while (1)
  {
    clrwdt();               // zerowanie watchdoga
    if (RFRXpacket())       // czy odebrany pakiet przez radio
    {
      pulseLEDR();          // sygnalizacja odebrania - migniecie czerwonego LED
      copyBufferRF2COM();   // skopiowanie odebranych danych
                            // z bufferRF do bufferCOM
      startSPI(DLEN);       // wysłanie bufferCOM przez SPI
    }
  }
}
```



Rysunek 19. Okno terminala z przesłaną temperaturą

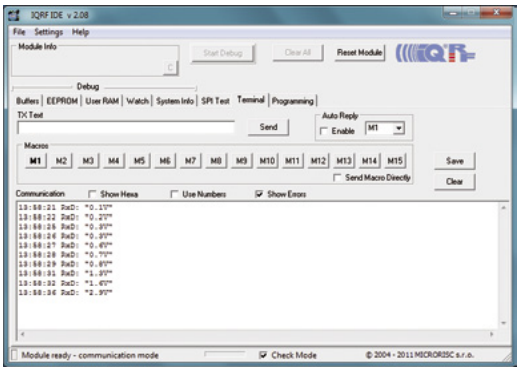
Listing 18. Inicjalizacja po zerowaniu mikrokontrolera
```
void APPLICATION()
{
  TRISA.5 = 1;     //  piny C5 (AN4) jako wejściowe
  TRISC.6 = 1;
  TRISB.4 = 1;
  InitADC();       //inicjalizacja przetwornika
  pulseLEDG();     // sygnalizacja początku działania aplikacji
  waitDelay(10);
  H_limit = 0x0FFF;
  L_limit = 0x0FFF;
.....
.....
```

Listing 19. Inicjalizacja przetwornika analogowo cyfrowego
```
void InitADC(void)
{
  ANSEL.4 = 1;            // pin C5 (AN4) jako wejście analogowe
  ADCON0  = 0b01010001;   // ustawienie ADC  (kanał AN4 ), Fosc/8
  ADCON1  = 0b10000000;   //format wyniku konwersji - right justified
}
```

Listing 20. Procedura pomiaru napięcia
```
void ReadAnalogInput(void)
{
  GO = 1;          // start konwersji  ADC
  while(GO);       // czekaj na zakończenie konwersji  ADC
  ADC_result.high8 = ADRESH & 0x03;  // zapisanie 10 bitowego wyniku
  ADC_result.low8  = ADRESL;  //  z  ADRESH i  ADRESL
}
```

Listing 20. Pętla pomiaru i przesyłania wyniku droga radiową

```
while(1)
{
  clrwdt();
  waitDelay(100);                //opóźnienie 1 sekundy
  ReadAnalogInput();             //wykonanie odczytu
  //testowanie czy wynik spełnia warunek przesłania
  if ((ADC_result > H_limit) || (ADC_result < L_limit))
  {
    if (ADC_result < TOLERANCE)  // ustawienie nowego limitu
    {
      H_limit = TOLERANCE;
      L_limit = 0;
    }
    else
    {
      H_limit = ADC_result + TOLERANCE;
      L_limit = ADC_result - TOLERANCE;
    }
    voltage = ADC_result * 30; //konwersja pomiaru
    voltage /= 1024;
    bufferRF[0] = voltage/10;  //zapisanie bufora bufferRF
    bufferRF[0] += ,0';
    bufferRF[1]  = ,.';
    bufferRF[2] = voltage % 10;
    bufferRF[2] += ,0';
    bufferRF[3] = ,V';
    pulseLEDR();
    PIN  = 0;
    DLEN = 4;
    RFTXpacket(); //przesłanie pakietu
  }
}
```



Rysunek 20. Okno terminala z przesłanym napięciem

MSSP mikrokontrolera PIC16F688 pracującego w trybie I²C master.

Przykład 6. Moduł DCC-SE-01 Pomiar napięcia

Do pomiaru napięcia z wyjścia potencjometru P2 jest wykorzystany przetwornik A/C mikrokontrolera. Ten sam przetwornik mierzy napięcie z wyjścia czujnika temperatury (konwerter temperatura/napięcie) umieszczonego na module TR52B. Jednak tam do odczytywania napięcia odpowiadającego mierzonej temperaturze używaliśmy funkcji systemowej *getTemperature()*. W tym wypadku trzeba będzie krok po kroku zainicjować programowo przetwornik, a potem go obsługiwać. Tu również posłużę się gotowym firmowym przykładem *DCC-SE-01_ptm.c*.

Wyjście potencjometru (bufora) jest połączone z wyprowadzeniem C5. C5 jest połączone równolegle do portów RA5, RC6 i RB4 w mikrokontrolerze modułu TR52B. Po uruchomieniu aplikacji użytkownika te linie musza być skonfigurowane jako wejściowe. Przetwornik jest inicjowany funkcją *InitADC()*, co można zobaczyć na **listingu 18** i **listingu 19**.

Odczytywanie napięcia na wejściu AN4 wykonuje procedura *ReadAnalogInput* zamieszczona na list. 19. Wynik konwersji jest zapisywany do 16 bitowej zmiennej *ADC_result*. Pomiar napięcia jest wykonywany cyklicznie co około 1 s. Przyjęto założenie, że pomiar będzie wysyłany, jeżeli zmieni się o określoną wartość. Żeby wymusić przesłanie trzeba zmieniać napięcie wejściowe przez kręcenie ośką potencjometru umieszczonego na module DCC-SE-01. Pętlę główną programu pokazano na **listingu 20**, a wynik odczytywania przesłanej wartości w IRQF na **rysunku 20**. Jak w poprzednim przykładzie, do komputera z uruchomionym IRQF IDE jest podłączony poprzez USB moduł DK-USB-04 z zaprogramowanym modułem TR52B programem E02-RX.

Podsumowanie

Tworzenie programów w systemie IQRF jest stosunkowo łatwe. Mamy tu do dyspozycji dość silne wsparcie w postaci wbudowanego OS. Jest ono szczególnie widoczne w zadaniach wykorzystujących transmisję radiową i komunikację pomiędzy modułem radiowym a zewnętrznym hostem. Oczywiście, trzeba poznać działanie systemu i funkcji systemowych. Bez tego trudno będzie pisać poprawnie działające i wydajne programy. Pomocny w tym są przygotowane przez producenta programy przykładowe i dobrze przemyślany pakiet IQRF IDE. Zwłaszcza przykłady programów dają wyczerpującą odpowiedź na szereg wątpliwości, które mogą pojawić się podczas nauki użytkowania i programowania IQRF OS.

Tomasz Jabłoński
tomasz.jablonski@ep.com.pl