# IQRF – więcej niż radio (6)

## Praktyczny projekt sieci IQRF

**W poprzednich numerach EP zaprezentowałem teoretyczne podstawy zastosowania systemu modułów radiowych produkowanych przez firmę IQRF. Były one ilustrowane przykładami procedur pokazujących sposoby użycia wielu ciekawych funkcji tych modułów. Często były to firmowe programy demonstracyjne. Są one według mojej opinii na tyle dobrze przygotowane, że napisanie własnych tylko po to, aby nieco różniły się od firmowych uznałem za bezcelowe. Jednak to co jest dobre do poznania zasad rządzących systemem zazwyczaj nie wystarcza do zbudowania kompletnego projektu. Ponieważ w trakcie poznawania IQRF bardzo mi się ten system spodobał, postanowiłem zaprojektować i wykonać projekt sieci radiowej z wykorzystaniem modułów TR52B. Artykuł jest kontynuacją umieszczonego w EP 4/2012.**

Przy omawianiu dołączania węzłów do sieci i konfigurowania odpytywania wspomniałem o kilku ważnych komendach. Teraz przedstawię formalny zapis używanych komend, sekwencji komend i sposób wprowadzania ich do systemu.

Komendy systemowe składają się z:

- prefiksu, którym jest znak „#”,
- kodu komendy w formie pojedynczego znaku ASCII,
- jednego lub kilku argumentów komendy; argumenty są ujęte w nawiasy okrągłe i jeżeli jest ich więcej niż jeden, to są rozdzielone przecinkami.

Składnia każdej komendy jest sprowadzana przez program. Jeżeli jest ona nieprawidłowa, to komenda jest odrzucana i użytkownik otrzymuje odpowiednią informację. Przedstawiona dalej lista komend może być niekompletna, bo system jest rozwijany i dodawane są nowe polecenia, a istniejące mogą być modyfikowane. Jednak są to najważniejsze komendy pozwalające na skonfigurowanie i używanie systemu. Modyfikacje będą dotyczyć głównie komend i ich argumentów potrzebnych do obsługiwania funkcji wykonywanych przez węzły. Na przykład trudno jest przewidzieć wszystkie warianty pomiarów temperatury (rozdzielczość, czujniki), sposobu sterowania itp.

**Listing 7. Deklarowanie struktury argumentów**

```
struct ar {
  unsigned char dat;
  unsigned char dat1;
  unsigned char cmd;
  unsigned char param;
};
struct ar arg;
```

**Listing 8. Obsługa komendy 'b'**

```
case ‚b':
  if(arg.dat==0||arg.dat==0xff){
    memcpypgm2ram (comm_ret,” argument out of range”,22);
    break;
  }
  if(ReadEE(CNODA)>=MAXNODE1){
    memcpypgm2ram (comm_ret,” too many nodes !!!   „,22);
    break;
  }
  if(CheckBondNode(arg.dat)==NOERROR){
    memcpypgm2ram (comm_ret,” node already bonded „,22);
    break;
  }
  bonda=BONDA;
  argum=ReadEE(CNODA);//current pos. on node Table bond_node
  *(bond_node+argum)=arg.dat;//upgrade table witch new node
  WriteEE(bonda+argum,arg.dat);//save new node on the node table
  WriteEE(CNODA,++argum);//next pos on bond_node
  block=1;
  CmdModule(arg.dat,0,’B’);//run cmd
  block=0;
  memcpypgm2ram (comm_ret,” node bonded now !!!    „,22);
  break;
```

### Argumenty komend

Argumenty wprowadzane z konsoli znakowej są zapisywane w strukturze *arg* (**listing 7**). Składowe *arg.dat* i *arg.dat1* są liczbabi, *cmd* jest znakiem ASCII (kod komendy), natomiast *param* może być liczbą (teraz używane), lub znakiem ASCII ( do przyszłych zastosowań). Jeżeli *arg.dat* lub *arg.dat1* są liczbami z zakresu 0…255, to są one wprowadzane jako 3 znaki cyfr z zakresu 0…9. Na przykład liczba „1” musi być wprowadzona jako „001”. Sztywne wprowadzanie nieznaczących zer przed cyframi znaczącymi jest podyktowane koniecznością uproszczenia interpretera komend. Interpreter sprawdza czy każdy wprowadzany znak jest kodem ASCII cyfry z zakresu 0…9. Jeżeli tak nie jest, to jest zgłaszany błąd *Arg error*. Potem, na podstawie wprowadzonych znaków jest wyliczana wartość argumentu i zapisywana do składowej *arg.dat* struktury *arg*. Funkcja obsługi każdej z komend musi sam sprawdzić czy zakres wartości wprowadzonego argumentu jest dla niej odpowiedni i jeżeli nie, to zgłasić błąd *out of range*”. Pozostałe argumenty są pojedynczymi znakami i ich wprowadzanie nie wymaga komentarza.

### Komenda dołączania węzłów – bonding

Ta komenda ma format *#b(node)*. Argument *node* jest liczbą z zakresu 1…254. Komenda dołączenia węzła o *NODE ID =1* ma postać *#b(001)*. Na **listingu 8** pokazano fragment obsługi komendy ‘b’. Najpierw jest sprawdzany zakres argumentów. Jeżeli NODE ID węzła ma wartość 0 lub 0xFF (adresy zarezerwowane), to funkcja kończy pracę i zwraca komunikat *Argument out of range*. Potem jest sprawdzane czy kolejne dołączenie nie powoduje przekroczenia maksymalnej liczby węzłów w systemie określoną stałą *MAXNODE1*. Jeżeli tak, to jest wysyłany komunikat *Too many nodes!!!*. Kolejny test polega na sprawdzeniu czy dołączany węzeł nie jest już na liście. Jeżeli w systemie jest już węzeł o takim *NODE ID*, to dołączenie nie ma sensu i po zgłoszeniu komunikatu *Node already bonded* próba dołączania jest przerywana.

Po poprawnym przejściu testów na prawidłowość argumentu (adresu NODE ID dołączanego węzła) z pamięci EEPROM jest odczytywana pozycja wolnego miejsca w tablicy dołączonych węzłów. Potem w buforze *bond_node* i jego kopii w pamięci EEPROM pod tą pozycją jest zapisywany *NODE ID* dołączanego węzła. Następnie pozycja w tablicy jest zwiększana o 1 i zapisywana w pamięci EEPROM. Procedura *CmdModule(arg.dat,0,'B')* wysyła do węzła polecenia zmiany swojego NODE ID z 255 na wartość zapisaną w *arg.dat*.

## Komenda odłączenia węzła unbonding

Komenda *#u(node)* usuwa węzeł o *NODE ID = node* z listy dołączonych węzłów i dodatkowo może zdalnie zmienić *NODE ID* węzła na adres rozgłoszeniowy 0xFF. Na **listingu 9** pokazano obsługę tej komendy. Oprócz standardowych testów na poprawność argumentu kluczową rolę spełnia tu funkcja *ShiftNodeBuf*, która usuwa węzeł z listy i przesuwa pozostałe wpisy, tak aby lista była ciągła. *CmdModule* wysyła do węzła polecenia zmiany *NODE ID* na 0xFF.

## Komendy konfigurowania odpytywania urządzenia „d", „a", „i" oraz „j"

Konfiguracja odpytywania urządzeń jest związana z sekwencją komend „d" i „a". Jako pierwsza musi być wywołana komenda *#d(device)*. Ustawia ona numer urządzenia, które będzie konfigurowane komendą „a". Wywołanie *#device(000)* nie wykonuje żadnych ustawień, tylko zwraca wcześniej ustawiony, bieżący numer urządzenia. Numer ustawionego urządzenie jest zapisywany w pamięci EEPROM i w zmiennej *curr_device*.

Komenda „a" ma format *#a(node, command, param)*. Jej działanie opisałem wcześniej w punkcie „odpytywanie". Duża zaletą konfigurowania odpytywania jest elastyczność. Parametry *cmd* i *param* pozwalają na przesłanie zapytania o różne dane. Pokażę to na przykładzie zaimplementowanego pytania o temperaturę. Mając do dyspozycji moduł TR52B i DCC-SE-01 można mierzyć temperaturę z czujnika MCP9700A, czujnika DS18B20 lub MCP9802. System musi wysłać do węzła zapytanie zawierające:

- polecenie pomiaru temperatury – komenda „T",
- typ czujnika – w naszym przypadku jeden z trzech dostępnych,
- format przesyłania zmierzonej wartości.

Typ czujnika i format przesyłanego pomiaru jest zawarty w argumencie param (**rysunek 15**). Komenda konfigurująca odpytywanie węzła wygląda na przykład tak: *#a(003,T,5)*.

Węzeł może przekonwertować zmierzoną temperaturę i przesłać w postaci ciągu znaków ASCII. Ma to tę zaletę, że pomiar może być bezpośrednio wyświetlany po doczytaniu z węzła. Ma tez i wady. Po pierwsze konwersja zajmuje i tak szczupłe zasoby węzła i ogranicza przez to możliwość wykonania innych komend. Poza tym kiedy chcielibyśmy wykorzystać pomiar do funkcji termostatu to ciąg ASCII trzeba by na nowo konwertować na postać cyfrową. Dlatego można przesyłać pomiar w postaci liczby 16 bitowej i konwersję na potrzeby wyświetlania przeprowadzić w hoście. Na **listingu 11** pokazano procedura obsługi komendy. W pierwszej kolejności jest sprowadzane, czy adres węzła (*NODE ID*) jest dołączony do sieci. Jeżeli nie to konfiguracja węzła do odpytywania nie może być wykonana.

**Listing 9. Obsługa komendy 'u'**
```
case ,u':
if(arg.dat==0||arg.dat==0xff){
    memcpypgm2ram (comm_ret," argument out of range",22);
    break;
  }
  if(CheckBondNode(arg.dat)==ERROR){
    memcpypgm2ram (comm_ret," node not bonded       „,22);
    break;
  }
  ShiftNodeBuf();
  block=1;
  CmdModule(arg.dat,0,'U');//run cmd
  block=0;
  memcpypgm2ram (comm_ret," node unbonded          „,22);
  break;
```

**Listing 10. Obsługa komendy 'd'**
```
case ,d':
  if(arg.dat>MAXNODE){
    memcpypgm2ram (comm_ret," argument out of range",22);
    break;
  }
  if(arg.dat>0)//set current device{
    curr_device=arg.dat;
    WriteEE(CDEVA,arg.dat);}//save to eeprom
    memcpypgm2ram (comm_ret," current device       „,22);
    Dig2Ascii(curr_device,str);
    *(comm_ret+16)=*(str+5); *(comm_ret+17)=*(str+6); *(comm_
ret+18)=*(str+7);
    break;
  }
```

Użytkownik może przejrzeć konfigurację odpytywania używając komendy „i" wyświetlającej konfigurację wprowadzana komendą „a" urządzenia z argumentu komendy. Jeżeli argument jest równy 0, to komenda wyświetla ilość odpytywanych urządzeń. Obsługę tej komendy pokazano na **listingu 12**. Konfiguracja pomijania cykli jest wyświetlana po wykonaniu komendy „j".

## Komenda konfigurująca pomijanie cykli odpytywania „s"

Komenda „s" ma format *#s(device, skip_cycle)*. Oba argumenty są liczbami trzycyfrowymi. Liczba pomijanych cykli *skip_cycle* może zmieniać się w zakresie 1…256 i jest zapisywana w składowej *arg.dat1*, a adres węzła w składowej *arg.dat* struktury argumentów *arg*. Mechanizm działania pomijania cykli opisałem już wcześniej. Na **listingu 14** pokazano obsługa komendy „s". Do struktury konfiguracyjnej *ask[device].skip* jest wpisywana wartość *skip_cycle*. *Skip_cycle* dla urządzenia *device* jest również zapisywany w pamięci EEPROM.

## Komenda „C"

Do tej pory opisywałem komendy konfiguracyjne: dołączania węzłów i odpytywania. Komenda „C" pozwala

**Listing 11. Obsługa komendy 'a'**
```
case ,a':
  if(CheckBondNode(arg.dat)==ERROR)//konfigurowany węzeł nie jest
dołączony do sieci
  {
    memcpypgm2ram (comm_ret," node not bonded       „,22);
    break;
  }
    //zapisanie parametrów do eeprom
   cmda=(curr_device-1)*4; cmda+=CMDA;
    aska=(curr_device-1)*3; aska+=ASKA;
    WriteEE(cmda,arg.cmd);
    WriteEE(++cmda,0); WriteEE(++cmda,arg.param);
    WriteEE(++cmda,arg.dat);
    cmdnode[curr_device-1].cmd=arg.cmd;
    cmdnode[curr_device-1].param=arg.param;
    cmdnode[curr_device-1].node=arg.dat;
    memcpypgm2ram (comm_ret," set ask dev.      „,22);
    Dig2Ascii(curr_device,str);
    *(comm_ret+14)=*(str+5); *(comm_ret+15)=*(str+6); *(comm_
ret+16)=*(str+7);
    ask[curr_device-1].nodeask=curr_device-1;//start odpytywania
    WriteEE(aska,curr_device-1);//konfiguracja do eeprom
    break;
```

konsoli na przesłanie do węzła dowolnej innej komendy. Ma ona podobny format, jak komenda „a”. Różnica pomiędzy nimi polega na tym, że „a” jest przeznaczona do konfigurowania odpytywania. Po skonfigurowaniu mechanizm odpytywania automatycznie kompletuje komendy przeznaczone dla węzłów i wysyła je kolejno do nich. Jak wspomniałem wcześniej, komendy wykonywane w odpytywaniu nie mogą być sterowaniami i dodatkowo komendami konfigurującymi działanie węzła. Komenda „C” nie ma takich ograniczeń. Można za jej pomocą wysłać każdą komendę obsługiwaną przez dołączony węzeł. Na przykład wprowadzenie z konsoli komendy *#C(002,T,5)* powoduje wysłanie do węzła *002* zapytania o pomiar temperatury z czujnika DS18B20, a pomiar zostanie odesłany w postaci liczby 16-bitowej (**rysunek 15**). Komenda „C” jest idealna do „ręcznego” konfigurowania pracy węzłów. Do tego celu przeznaczyłem kilka specjalnych rozkazów (**rysunek 16**).

**Listing 12. Obsługa komendy 'i'**
```
case ‚i’:
if(arg.dat>MAXNODE){
  memcpypgm2ram (comm_ret,” argument out of range”,22);
  break;
}
if(arg.dat==0){
  memcpypgm2ram (comm_ret,” devices amount        „,22);
  Dig2Ascii(MAXNODE,str);
  *(comm_ret+16)=*(str+5); *(comm_ret+17)=*(str+6); *(comm_
ret+18)=*(str+7);
  break;
}
memcpypgm2ram (comm_ret,” com  ,node    ,par   „,22);
*(comm_ret+5)=cmdnode[arg.dat-1].cmd;
*(comm_ret+21)=cmdnode[arg.dat-1].param +0x30;
  Dig2Ascii(cmdnode[arg.dat-1].node,str);
*(comm_ret+12)=*(str+5); *(comm_ret+13)=*(str+6); *(comm_
ret+14)=*(str+7);
 break;
```

**Listing 13. Obsługa komendy 'j'**
```
case ‚j’:
  memcpypgm2ram (comm_ret,” skip count             „,22);
  aska=(arg.dat-1)*3; aska+=ASKA;
  Dig2Ascii(ReadEE(++aska),str);
  *(comm_ret+14)=*(str+5); *(comm_ret+15)=*(str+6); *(comm_
ret+16)=*(str+7);
  break;
```

**Listing 14. Obsługa komendy 's'**
```
case ‚s’:
  if(arg.dat1==0){
    memcpypgm2ram (comm_ret,” arg. skip out of range”,22);
    break;
  }
  if(arg.dat>MAXNODE||arg.dat==0){
    memcpypgm2ram (comm_ret,” arg. dev out of range”,22);
    break;}
    aska=(arg.dat-1)*3; aska+=ASKA;
    WriteEE(++aska,arg.dat1);//save skip cycle
    ask[arg.dat-1].skip=arg.dat1;
    memcpypgm2ram (comm_ret,” set ask skip      „,22);
    Dig2Ascii(arg.dat1,str);
    *(comm_ret+14)=*(str+5); *(comm_ret+15)=*(str+6); *(comm_
ret+16)=*(str+7);
    break;
```

**Listing 15. Obsługa komendy 'C'**
```
case ‚C’:
  if(CheckBondNode(arg.dat)==ERROR){
    memcpypgm2ram (comm_ret,” node not bonded       „,22);
   break;
  }
  if(arg.dat==0||arg.dat==0xff){
    memcpypgm2ram (comm_ret,” argument out of range”,22);//argument
out of range
    break;
  }
  while(BlinkCounter==0);
  block=1;
  if(CmdModule(arg.dat,arg.param, arg.cmd)==ERROR)//run cmd
  memcpypgm2ram (comm_ret,” command timeout    „,22);
  block=0;
  break;
```

| Param | Czujnik | Format przesyłanego pomiaru |
|---|---|---|
| 0 | MCP9700A | Dane ADSCII np. T=20,0*C |
| 1 | DS18B20 | Dane ASCII |
| 2 | MCP9802 | Dane ASCII |
| 4 | MCP9700A | 16 bitowa dana – konwersja na ASCII w hoście |
| 5 | DS18B20 | 16 bitowa dana – konwersja na ASCII w hoście |
| 6 | MCP9802 | 16 bitowa dana – konwersja na ASCII w hoście |

**Rysunek 15. Przykład typów czujnika i formatu przesyłanego pomiaru**

| komenda | node | com | param | opis |
|---|---|---|---|---|
| F | 1…254 | F | 0 | Wprowadza węzeł w tryb zdalnego przeprogramowania pamięci FLASH |
| P | 1…254 | P | Power | Ustala moc wyjściową modułu TR52B węzła |
| C | 1…254 | C | Channel | Pozwala na zmianę kanału RF modułu TR52B |

**Rysunek 16. Przykład komend do konfigurowania pracy węzłów**

Jeżeli chcemy zmienić oprogramowanie (program użytkownika) w węźle o *NODE ID=3* wysyłamy komendę *#C(003,F,0)*. Po jej odebraniu węzeł odsyła komunikat *RF PGM* i wchodzi w tryb programowania przez radio. Wykorzystując mechanizmy IQRF OS można teraz przeprogramować zawartość pamięci Flash mikrokontrolera PIC16F884 modułu TR52B. Nie trzeba chyba nikogo przekonywać, jak wielkie znaczenie praktyczne ma ta funkcja. Funkcje zmiany mocy i kanału wykorzystują procedury systemowe IQRF OS.

Funkcje F, P i C to tylko przykład możliwości zdalnej konfiguracji modułów. Rozszerzenie ich listy zależy tylko od inwencji programisty. Jak do dodatkową można traktować funkcję żądania pomiaru poziomu naładowania baterii modułu węzła. Na przykład *#C(003,B,5)* pyta węzeł o napięcie baterii, a węzeł odpowiada wartością binarną, która jest potem konwertowana na znaki ASCII przez hosta. Obsługę komendy „C” pokazano na **listingu 15**.

## Oprogramowanie węzła

Początkowo chciałem, aby program dla modułu TR52B pełniącego rolę węzła był jak najbardziej uniwersalny. Jednak szybko okazało się, że 1 kB pamięci Flash dla użytkownika to za mało do takiego zadania. Dlatego program węzła to szkielet zawierający obsługę podstawowych funkcji: „b”, „u”, „F” i „P” oraz funkcje odbioru danych z koordynatora (łącznie z testowaniem poprawności). Pozostałe funkcje są dołączane w miarę konkretnych potrzeb. Dlatego trudno mówić o oprogramowaniu węzła jako o jednym programie.

Program musi być tak napisany, by po wyzerowaniu mikrokontrolera były odczytane i ustawione wszystkie wcześniej zapamiętane, istotne ustawienia: *NODE ID* węzła i moc nadajnika. Po zaprogramowaniu pamięci modułu trzeba ustawić domyślnie maksymalną moc nadajnika i domyślny adres rozgłoszeniowy 0xFF.

Wykorzystamy tutaj fakt, że skasowana pamięć mikrokontrolera jest zapisana bajtami 0xFF. Procedura inicjalizacji sprawdza czy pod określoną lokacją (*ADR_INIT*) jest zapisana wartość np. 0x38. Jeżeli nie to wykonywana jest procedura inicjowania ustawień wartościami domyślnymi i pod *ADR_INIT* jest zapisana liczba *0x38*. Przy kolejnym zerowaniu procedura inicjalizacji zostanie pominięta. Fragment programu inicjalizacji ustawień węzła pokazano na **listingu 16**. Po zainicjalizowaniu program wchodzi w pętlę odbioru danych z koordynatora, wykonania komendy i odesłania potwierdzenia (**listing 17**).

Sprawdzenie warunku *if (checkRF(5))* zabezpiecza przed odebraniem pakietu danych przesyłanego przy zbyt niskim poziomie sygnału. Potem sprawdzane jest

czy IQRF OS odebrał prawidłowy pakiet danych z poprawnymi sumami kontrolnymi CRC. Jeżeli tak to możemy być (prawie) pewni, że dane nie zostały przekłamane.

Teraz trzeba zidentyfikować czy są one przeznaczone do tego węzła. Najpierw jest wykonywana identyfikacja sieci. Kiedy pierwszy odebrany bajt jest równy *NET ID*, to można sprawdzić, czy dwa następne bajty to 0x05 (długość komendy z koordynatora) i 0xFA (negacja bajtu długości). Jest to jeden z elementów dodatkowej kontroli poprawności przesyłanych danych. Potem sprawdzamy, czy *NODE ID* z przesłanego nagłówka jest równe *NODE ID* zapisanemu w konfiguracji modułu węzła. Po przejściu tego testu węzeł może być „pewny", że dane są przeznaczone do niego i może zacząć identyfikować i ewentualnie wykonywać przesłana komendę. Tę czynność wykonuje procedura *CheckCmd*. Jak już wspomniałem, węzeł musi obsługiwać podstawowe komendy, a obsługa pozostałych zależy od przeznaczenia węzła. Na **listingu 18** pokazano przykładową procedurę *CheckCmd* węzła, który obsługuje pomiary temperatury i poziomu napięcia baterii.

Obsługa komendy przyłączania węzła do sieci polega na sprawdzeniu czy *NODE ID* jest równe 0xFF. To sprawdzenie jest wykonywane nieco na „wszelki wypadek", ponieważ komenda „b" nie może być przysłana do węzła o *NODE ID* innym niż 0xFF. Potem pod adres w pamięci EEPROM zawierający *NODE ID* i do zmiennej *node_id* jest zapisywana nowa wartość i węzeł ma adres pozwalający mu na pracę w sieci. Komenda „u" zmienia *NODE ID* na 0xFF i zapisuje tę wartość w pamięci EEPROM modułu.

**Listing 16. Inicjacja parametrów**

```
if(eeReadByte(ADR_INIT)!=0x38){
  eeWriteByte(ADR_INIT,0x38);//wyłącz późniejszą inicjalizację
  eeWriteByte(ADR_NODE_ID,0xff);//adres rozgłoszeniowy
  eeWriteByte(ADR_POWER,7);//max. Moc  domyślna
}
//inicjalizacja po każdym zerowaniu modułu
setTXpower(eeReadByte(ADR_POWER));//inicjalizacja mocy
node_id=eeReadByte(ADR_NODE_ID);//inicjalizacja NODE ID
toutRF = 4;
```

**Listing 17. Pętla główna programu węzła**

```
while (1){
  clrwdt();
  if (checkRF(5))  // skanowanie poziomu (jakości ) sygnału
  {
    if (RFRXpacket())  //czy odebrano dane?
    {
      pulseLEDR(); // sygnalizacja odebrania danych
// skopiowanie odebranych danych bufferRF do  bufferCOM
      copyBufferRF2COM();
//sprawdzanie zwartości nagłówka
      if(bufferCOM[0]!=NET_ID) continue;//błąd identyfikatora sieci
NET_ID
      temp=~bufferCOM[2];
      if(bufferCOM[1]!=temp) continue;//błąd długości nagłówka
      if(bufferCOM[3]!=node_id){
      pulseLEDR() continue;}//błąd NODE ID - dane nie do tego węzła
      if(CheckCmd()==ERROR) continue;//błąd kodu komendy, lub
komenda nieobsługiwana przez węzeł
//komenda wykonana, pole danych zapisane
      CmdHead(bufferCOM[4],DLEN);//kompletowanie nagłówka
odpowiedzi
      PIN = 0;
      RFTXpacket();  //Wyślij dane przez radio
      pulseLEDG();
      waitDelay(10);
    }//end if (RFRXpacket())
  }//end if (checkRF(5))
}//end while(1);
```

Uruchomienie zdalnego przeprogramowanie modułu (komenda „F") jest wykonywane przez procedurę *SetFlashUpdate* (**listing 19**). Najpierw moduł odsyła droga radiową komunikat *RF PGM run*, a potem jest wykonywana

funkcja systemowa *runRFPGM* wprowadzająca moduł w tryb zdalnego programowania. Po przeprogramowaniu mikrokontroler PIC16F884 modułu TR52B jest zerowany.

Również komenda zmiany mocy wyjściowej nadajnika modułu TR52B („p") wykorzystuje funkcję systemową. W tym wypadku jest to *setTXpower*. Dodatkowo, nowoustawiona wartość mocy jest zapisywana w pamięci EEPROM pod adresem *ADR_POWER* (**listing 20**). Procedura sprawdza też czy ustawiana moc ma prawidłowy zakres wartości. Jeżeli tak, to moduł odsyła komunikat *cmd OK*, a jeżeli nie, to komunikat *No support*.

Węzeł obsługuje też komendę mierzącą napięcie baterii zasilającej. Jest to tym łatwiejsze, że IQRF OS ma wbudowaną funkcję systemową *getSupplyVoltage* wykonującą to zadanie. Obsługę komendy „B" pokazano na **listingu 21**.

**Listing 18. Identyfikacja i wykonanie komend węzła modułu TR52B**
```
unsigned char CheckCmd(void){
  switch (bufferCOM[4]){
    case ,b':  //komenda dołączania węzła
      if(node_id=0xff)
      break;//powrót z błedem
      eeWriteByte(ADR_NODE_ID,bufferCOM[3]);
      node_id=bufferCOM[3];
      return(NOERROR);
      case ,u'  //komenda odłączania węzła
      eeWriteByte(ADR_NODE_ID,0xff);
      node_id=0xff;
      return(NOERROR);
    case ,T':  //pomiar temperatury
      //pomiar z czujnika modułu TR52B
      if(bufferCOM[6]==0||bufferCOM[6]==4){
        BoardTemp();
        return(NOERROR);}
      //pomiar z czujnika DS18B20
      if(bufferCOM[6]==1||bufferCOM[6]==5){
        OneWireTemp();
        return(NOERROR);
      }
    case ,F':  //wprowadzenie w tryb PGM RF
      SetFlashUpdate();
      return(NOERROR);
    case ,B':  //Pomiar napięcia baterii
      BatteryLevel();
      return(NOERROR);
    case ,P':  //zmiana mocy nadajnika modułu
      SetPower(bufferCOM[6]);
      return(NOERROR);}
      break;
  }
  return(ERROR);
```

**Listing 19. Obsługa komendy 'F' (wprowadzanie w tryb RF PGM)**
```
void SetFlashUpdate(void){
  CmdFlash();
  CmdHead(bufferCOM[4],DLEN);//kompletowanie naglowka
  PIN = 0;
  RFTXpacket();
  runRFPGM();
}
```

**Listing 20. Obsługa komendy ustawiania mocy wyjściowej 'P' węzła TR52B**
```
void SetPower(unsigned char power){
  if(power<8){
    setTXpower(power);
    eeWriteByte(ADR_POWER,power);//default Node ID
    CmdOk();
  }
  Else CmdNoSupport();
}
```

**Listing 21. Obsługa komendy 'B'**
```
void BatteryLevel(void){
  bufferRF[5]=getSupplyVoltage();//batt;//
  bufferRF[6]=0;
}
```

**Listing 22. Pomiar temperatury z czujnika modułu**
```
void BoardTemp(void){
  getTemperature();   //systemowa funkcja pomiaru temp.
  bufferRF[7]=ADRESL;//low byte
  bufferRF[8]=ADRESH & 0x03;//high byte
  if(bufferCOM[6]==4){
    DLEN=16;
   return;
  }
  else CmdNoSupport(); //No support dla parametru innego niż 4
```

Najbardziej rozbudowana jest obsługa komendy pomiaru temperatury „T". Moduł może mierzyć temperaturę z czujnika modułu tylko z parametrem równym *04* oraz z czujnika DS18B20 z parametrem *01* (dane przekonwertowane na ciąg znaków ASCII) i z parametrem *05* (16-bitowa dana). Pomiar z czujnika modułu odsyła tylko 16-bitowy wynik i nie wykonuje konwersji na ciąg znaków ASCII. Funkcja pomiaru jest nieskomplikowana, bo wykorzystuje funkcję systemową *getTemperature* pokazaną na **listingu 22**.

Procedura pomiaru z czujnika DS18B20 wysyła dane w obu formatach jednocześnie, a host zależnie od wysłanego parametru wybiera sobie rodzaj potrzebnych danych. Można oczywiście różnicować odczyt i przesłanie temperatury w zależności od odebranego parametru, ale tu nie ma potrzeby tego robić. Takie rozwiązanie było mi potrzebne do celów testowych i jest jednym z możliwych. W praktyce można zupełnie zrezygnować z konwersji na ASCII skoro i tak jest ona wykonywana w hoście. Na **listingu 23** pokazano obsługę komendy „T" dla parametrów *01* i *05*. Konwersja na ciąg ASCII jest skopiowana z przykładowego programu zamieszczonego w materiałach szkoleniowych firmy IQRF. Trzeba pamiętać, że pomiar temperatury za pomocą DS16B20 trwa ok. 630 ms i należy to uwzględnić przy konfigurowaniu odpytywania. Przy małej liczbie pytanych urządzeń może się zdarzyć, że kolejne zapytania węzła o temperaturę będą wykonywane częściej i trzeba będzie zaprogramować odpowiednią liczbę cykli pomijanych dla tego węzła.

Wszystkie procedury obsługi komend zapisują pole argumentu i danych w protokole zwrotnym . Przed wysłaniem danych przez radio, trzeba bufor nadajnika uzupełnić o pozostałe bajty nagłówka i całość wysłać zwrotnie. Nagłówek jest kompletowany przez funkcję *CmdHead* (**listing 24**). Zapisuje ona do bufora *bufferRF* identyfikator sieci *NET ID*, liczbę danych łącznie z polem danych, adres węzła *NODE ID* i kod wykonanej komendy z ustawionym najstarszym bitem b7.

Takie rozwiązanie oprogramowania węzła można potraktować jako szkielet z zaimplementowanymi niezbędnymi funkcjami. Można go w łatwy sposób uzupełniać o własne funkcje wykonujące różne zadania. W ostateczności, gdy węzeł ma do wykonania większe zadanie można potraktować moduł węzła jako tylko komunikacyjny połączony z lokalnym hostem przez SPI. Lokalny host po wykonaniu zadania prześle komunikat zwrotny do modułu przez SPI, a ten dalej do hosta – koordynatora przez radio.

## Oprogramowanie modułu radiowego hosta – koordynatora

Moduł TR52B połączony przez magistralę SPI do hosta – koordynatora pełni tylko rolę modułu transmisyjnego. Dane do przesłania przez radio są transmitowane przez interfejs SPI obsługiwany w tle przez procedury systemu IQRF OS. Program modułu koordynatora pokazano na **listingu 25**. W nieskończonej pętli głównej są wykonywane 2 główne zadania: odbieranie danych radiowych z węzła i przesyłanie ich po SPI do koordynatora oraz odbieranie danych z hosta po SPI i przesłanie ich drogą radiową do węzłów. Program koordynatora jest „przeźroczysty" dla przesyłanych danych, to znaczy nie ingeruje w ich zawartość poza jednym wyjątkiem –

częściowo jest sprawdzana zawartość nagłówka przesyłanego z hosta do węzła. Wykonuje to krótka procedura *CheckComm* zamieszczona na **listingu 26**. Jest sprawdzane czy poprawnie wysyłane są bajty identyfikatora *NET ID* oraz bajty liczby danych i negacja liczby danych.

Procedurę tę wykonałem do testowania prawidłowości wysyłanych pakietów danych w trakcie tworzenia programu dla hosta. Jeżeli mamy pewność, że to co wychodzi z hosta jest poprawne, to można zrezygnować z tego zabezpieczenia.

## Interfejs użytkownika

Oprócz roli hosta – koordynatora mikrokontroler PIC18F67J60 pełni również funkcję serwera strony WWW. Serwer łączy się z siecią LAN poprzez interfejs Ethernet. W pamięci mikrokontrolera jest zaimplementowany stos protokółów TCP/IP firmy Microchip i strona WWW interfejsu użytkownika. Takie rozwiązanie pozwala na otwarcie strony za pomocą dowolnej przeglądarki pod warunkiem, że jest obsługiwana przez nią technologia Adobe Flash. Obsługa Flash'a jest konieczna, bo używa jej nasz serwer. Interfejs Ethernet i protokóły TCP/IP dają możliwości komunikowania się poprzez Internet , oczywiście po spełnieniu warunków dostępu do sieci. Ja w trakcie testowania bez problemu łączyłem się z moim domowym routerem zintegrowanym z modemem ADSL. Wymaga to dodatkowej konfiguracji routera i jeżeli nie mamy stałego adresu IP, to jest dość kłopotliwe, ale wykonalne. W sieci LAN jest wymagane tylko połączenie hosta kablem Ethernet do routera, switcha lub komputera, a serwer automatycznie pobierze i przydzieli sobie lokalny adres z wolnej puli w sieci i układ jest gotowy do pracy. Domyślny adres serwera to http://mchpboard . Każdy kto zna rozwiązania Microchipa wie, że jest to adres stosowany w firmowych przykładach użycia stosu TCP/IP i w firmowych modułach ewaluacyjnych np. Explorer16. Nic nie stoi na przeszkodzie by ten adres zmienić . Definicja adresu jest umieszczona w pliku TCPIPConfig.h:

```
#define MY_DEFAULT_HOST_NAME       "MCHP-BOARD"
```

W tym pliku jest tez zdefiniowany adres MAC ( domyślnie 00-04-A3-00-00-00) i można go również zmienić jeżeli z jakichś powodów jest to konieczne. Na przykład wtedy, kiedy do sieci LAN podłączymy więcej niż jeden host.

Budowanie serwera WWW potrafiącego przesyłać dane do przeglądarki i odbierać dane z przeglądarki jest opisane w materiałach szkoleniowych na stronach firmy Microchip. Nie jest to zadanie łatwe, a na pewno żmudne. Można do tego celu użyć rewelacyjnego (moim zdaniem) narzędzia TCPMaker amerykańskiej firmy *TRACESystemsInc* opisywanego w EP 4/2012. Za jego pomocą można zbudować wydajną aplikacje sterującą w bardzo krótkim czasie. Program ma same zalety, ale niestety nie jest darmowy, a nawet jak na nasze warunki dosyć drogi (około 1000 złotych). Jednak jest idealny dla kogoś, kto chce szybko i łatwo projektować strony WWW komunikujące się z aplikacjami sterowania. TCPMaker oprócz projektu strony potrafi zbudować po stronie serwera (mikrokontrolera) kompletny projekt przeznaczony dla środowiska MPLAB IDE. Po uzupełnieniu własnymi procedurami można zbudować wydajny i niezawodny system przesyłania danych oparty

**Listing 23. Pomiar temperatury z DS18B20**

```
void OneWireTemp(){
// 1 - DS18B20 połączony  0 - DS18B20 niepołączony
 if (DS_GetTemperature())
//funkcja DS_GetTemperature wykonuje się ok. 630 ms
  {
    bufferRF[5]=temperature.low8;  //16 bitowy wynik
    bufferRF[6]=temperature.high8;
    bufferRF[7] = ,T';          // konwersja na ciąg ASCII
    bufferRF[8] = ,=';
    bufferRF[12] = ,.';
    bufferRF[14] = ,*';
    bufferRF[15] = ,C';
    if (temperature.high8 & 0x80)   // temperatura ujemna?
    {  // tak
      temperature  = ~temperature;
      temperature++;
      bufferRF[9] = ,-';
    }
    else bufferRF[9] = , ,;
    tenths = temperature.low8 & 0x0F;
    temperature >>= 4;
    i = temperature.low8 / 10;
    bufferRF[10] = i + ,0';
    i = temperature.low8 % 10;
    bufferRF[11] = i + ,0';
    tmp = tenths * 625;
    i = tmp / 1000;
    tmp %= 1000;
    if ((tmp >= 500) && (i < 9)) i++;
    bufferRF[13] = i +'0';
    DLEN = 16;
  }
  else
  {
    bufferRF[7] = ,s';
    bufferRF[8] = ,e';
    bufferRF[9] = ,n';
    bufferRF[10] = ,s';
    bufferRF[11] = ,o';
    bufferRF[12] = ,r';
    bufferRF[13] = ,_';
    bufferRF[14] = ,e';
    bufferRF[15] = ,r';
    bufferRF[16] = ,r';
    DLEN = 17;
  }
}
```

**Listing 24. Kompletowanie nagłówka danych wysyłanych przez radio**

```
void CmdHead(char cmd, unsigned char ld){
  bufferRF[0]=NET_ID;
  bufferRF[1]=ld+5;
  bufferRF[2]=~bufferRF[1];
  bufferRF[3]=eeReadByte(ADR_NODE_ID);
  bufferRF[4]=cmd|0x80; //cmd dir N->C
}
```

na rozwiązaniach typu RTOS. Strona WWW interfejsu użytkownika składa się ze strony głównej i dwóch podstron. Pozwoliła mi ona na przetestowanie wszystkich rozwiązań. Dlatego nie jest to projekt rzeczywistego, przykładowego interfejsu automatyki domowej. Jednak mając do dyspozycji TCPmakera i skonfigurowaną sieć można interfejs szybko przeprogramować zależnie od wymagań użytkownika.

Z poziomu strony głównej można wybierać:

- podstronę konfigurowania sieci za pomocą przycisku *Network configuration*,
- podstronę, na której są wyświetlane dane przesyłane przez pytanie urządzenia (węzły) w czasie zdefiniowanego odpytywania; można tam oglądać odpytywanie 4 urządzeń; jest ona wybierana po naciśnięciu przycisku *Asking devices*.

Po kliknięciu na przycisk *Network configuration* otwiera się strona konfigurowania sieci (**rysunek 17**). Głównym elementem tej strony jest okno terminala znakowego *Enter command*. Terminal pozwala na wprowadzanie wszystkich komend systemowych. Komendy wprowadza się z klawiatury i są one wyświetlane w dolnej linijce okna.

Terminal ma też coś w rodzaju „ekranu", w którym mieści się 7 linijek tekstu. Tutaj wyświetlane są komunikaty tekstowe przesyłane przez serwer jako odpowiedź

**Listing 25. Program modułu TR52B koordynatora**

```
void APPLICATION()
{
  uns8 i, err;
  SWDTEN = 0; //wyłączenie watchdoga
  enableSPI();   // odblokowanie SPI
  toutRF = 4;//ramka dla odbioru RF
  while (1){
  //sprawdzanie czy sa dane odebrane przez radio z węzłów
  if (checkRF(5))  // testowanie poziomu sygnału
  {
    if (RFRXpacket())  // czy odebrano dane przez radio?
    {
      pulseLEDG();  // sygnalizowanie odbioru
      copyBufferRF2COM(); // kopiowanie odebranych danych
      startSPI(DLEN);  // odesłanie do hosta przez SPI
    }
  }
//sprawdzanie czy są dane odebrane z hosta przez SPI.
  if (getStatusSPI()==0)  //czy SPI nie zajęty
  // SPIpacketLength: długośc danych w bajtach
  //param2: SPI status
  {
    if (_SPIRX)  // czy odebrano dane po SPI
    {
      if (!_SPICRCok)  //   CRCM poprawne?
      {
        startSPI(0);  //   niepoprawne ->Restart SPI
        continue;  //    kontynuacja
      }
      waitDelay(35);  // odczekaj 350msek
      clrwdt();
      err=CheckComm();  //sprawdź poprawność  ramki
      if(err==ERROR)
      {
        pulseLEDG();
        copyBufferINFO2COM();
        startSPI(9);  //odeślij komunikat
        continue;
      }
      DLEN = SPIpacketLength;
      copyBufferCOM2RF();  // kopiuj odebrane dane
     PIN = 0;
      RFTXpacket();
      pulseLEDR();  // LED indication
      startSPI(0);  //restart SPI
    }//end if (_SPIRX)
  }//end if (getStatusSPI()==0)
}//end while(1)
}
```

na wprowadzenie komendy. Zazwyczaj jest to echo komendy i komunikat interpretera. Dla komendy „C" jest tu wyświetlana odpowiedź zaadresowanego węzła.

Przycisk *Home* jest przeznaczony do przejścia na stronę główną, a przycisk *Devices* do przejścia na stronę wyświetlania odpowiedzi odpytywanych urządzeń.

Oprócz okna terminala i przycisków nawigacyjnych na stronie umieściłem skróconą instrukcję konfigurowania sieci obejmującą głównie format komend. W trakcie pracy okazało się, ze jest bardzo przydatna pomoc, bo już po kilku dniach przerwy w pracy nad projektem bez tej pomocy konieczne było częste zaglądanie do notatek.


**Rysunek 17. Strona konfiguracji sieci**

**Listing 26. Testowanie zawartości nagłówka**

```
unsigned char CheckComm(void){
  if(bufferCOM[0]!=NET_ID) return(ERROR);
  if(bufferCOM[1]!=5) return(ERROR);
  if(bufferCOM[2]!=0xfa) return(ERROR);
  return(NOERROR);
}
```

## Ograniczenia

Zastosowany tutaj mikrokontroler PIC18F67J60 ma ograniczoną do nieco ponad 3 kB pamięć danych. Duża jej część jest wykorzystywana przez zmienne programu, pomimo że moduł MAC ma swoje 8 kB niezależnej pamięci RAM. Powoduje to, że można zdefiniować ograniczoną liczbę odpytywanych węzłów. Przypomnę: dla jednego odpytywanego węzła (urządzenia) rezerwujemy 7 bajtów pamięci RAM. Dla 100 węzłów potrzebujemy 700 bajtów. Niestety, nie mamy tyle do dyspozycji. To ograniczenie może nie być dokuczliwe, bo zazwyczaj nie pytamy o więcej niż kilka- kilkanaście węzłów. Jednak gdyby była potrzeba odpytywania dużej liczby urządzeń, można zastosować mikrokontroler o większych zasobach. Na potrzeby tego projektu przeprowadziłem próby z 32-bitowym mikrokontrolerem PIC32MX360F512L zawierającym 512 kB pamięci programu Flash i 32 kB pamięci RAM. Do testów użyłem modułu ewaluacyjnego *PIC32 Starter Kit*, płytki *PIC32 I/O Expansion Board* oraz modułu *Ethernet PICtail Plus Daughter Board* z układem ENC28J60. Taka konfiguracja wymagała zdefiniowania #define PIC32_GP_SK_DM320001 w pliku HardwareProfile.h i TCPMaker bez problemu wygenerował działający projekt serwera dla kompilatora MPLAB C for PIC32. Wszystkie niezbędne definicje obsługi interfejsu SPI, i układu ENC28J60 zostały zawarte w gotowym projekcie. Przeniesienie procedur obsługi sprzętu i konfiguracji sieci również nie sprawiło większych trudności.

Serwer na szybkim, 32-bitowym mikrokontrolerze pracuje zauważalnie szybciej, ale jest to naturalne. Rodzina PIC32 ma wbudowany bardzo wydajny rdzeń MIPS32 M4K i porównanie z nawet dość szybkim 8-bitowym rdzeniem PIC18 musi tak wypaść.

## Podsumowanie

Pokazany projekt radiowej sieci z użyciem modułów TR52B nie jest gotowym rozwiązaniem przeznaczonym do konkretnego zastosowania. Przemawia za tym użycie modułów ewaluacyjnych firmy IQRF, które trudno zastosować w działającej sieci może poza pomiarem temperatury. Również host został przetestowany na płytce nie przeznaczonej dla gotowego urządzenia. Poza tym jak łatwo zauważyć projekt strony WWW serwera jest przystosowany raczej do testowania rozwiązań, niż do ich zastosowania w konkretnej aplikacji. Jednak użyte tu i przetestowane rozwiązania mogą być prosto i małym nakładem pracy – pod warunkiem posiadania wszystkich niezbędnych narzędzi programowych – przekształcone w gotową działającą aplikację. Projekt był testowany w ograniczonym zakresie na podatność na zakłócenia i bezawaryjną pracę w dłuższym okresie czasu. Nie zauważyłem tendencji do trwałego zawieszania się całości mimo, że nie jest tu używany watchdog.

Jak wspomniałem na wstępie zamierzam przeprowadzić próby z wykorzystaniem sieci IQRF MESH. Zastosowane tam rozwiązania pozwalają na zwiększenie zasięgu radiowego i pewności dostarczania pakietów danych do węzła. Może to mieć znaczenie w sieciach, gdzie te zalety będą miały kluczowe znaczenie.

**Tomasz Jabłoński, EP**