Wprowadzenie do Linuksa embedded (5)

Obsługa portów szeregowych

Porty szeregowe RS232, RS485 i inne pokrewne już od wielu dziesięcioleci są dostępne w systemach mikroprocesorowych. Pomimo powolnego wypierania przez bardziej zaawansowane interfejsy, chociażby USB, nie zanosi się na to, aby w najbliższej przyszłości zostały całkowicie wyeliminowane. W zasadzie w sprzęcie powszechnego użytku nie spotyka się już urządzeń wyposażonych w RS232, ale pozostaną one nadal wykorzystywane do programowania układów, diagnostyki (debugowania) itp. Swoją bardzo długą żywotność zawdzięczają głównie prostocie, a co za tym idzie łatwej implementacji i dostępności nawet w najtańszych mikrokontrolerach. Z prostotą wiąże się również łatwa obsługa programowa. W Linuksie obsługa portów szeregowych, od strony aplikacji użytkownika jest łatwa, pomimo dużego skomplikowania podsystemu TTY.

Dodatkowe materiały na CD/FTP:
ftp://ep.com.pl, user: 19623, pass: 6c5r20n3
- wszystkie poprzednie części kursu

W większości wypadków wszelkie sterowniki dla portów szeregowych, są dostępne w jądrze i tylko w przypadku portowania jądra do nowej architektury, będziemy zmuszeni do napisania odpowiednich sterowników. Aby zapoznać czytelników z obsługą portu szeregowego, pokażemy przykład prostego konwertera RS232<->TCPIP, który dzięki zaawansowanemu systemowi będziemy mogli napisać w kilkudziesięciu liniach kodu.

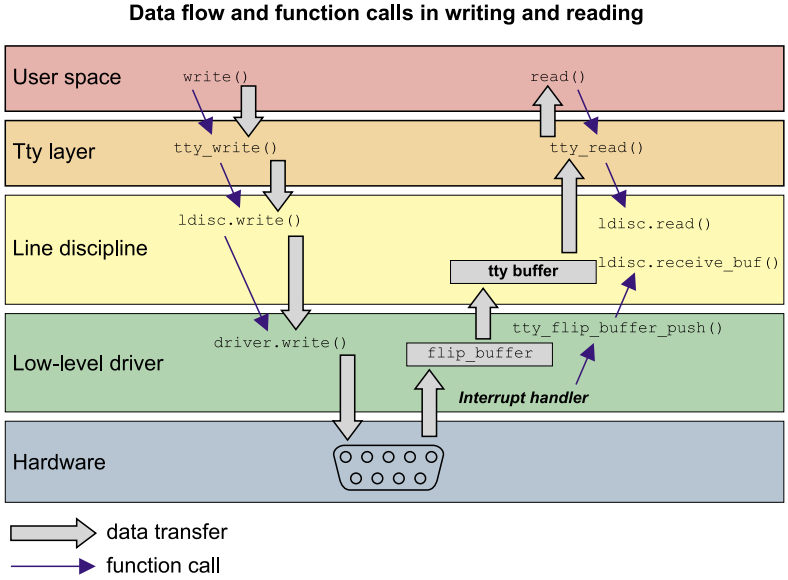
Porty szeregowe w Linuksie – architektura

Architektura, portów szeregowych w Linuksie wywodzi się z czasów, w których w systemach UNIX'owych, do głównego komputera za pomocą linii szeregowych dołączane były terminale znakowe komunikujące się z jednostką centralną. (Jako ciekawostkę na **fotografii 8** przedstawiono oryginalny terminal VT100)

Fotografia 8. Terminal znakowy VT100 (*http://www.catb.org/~esr/writings/taouu/html/ch02s02.html*)

Terminal szeregowy komunikował się z odległą jednostką centralną za pomocą magistrali szeregowej, ewentualnie z wykorzystaniem modemu, umożliwiając pracę wielu użytkownikom równocześnie na jednej maszynie. Takie terminale do dziś można spotkać w niektórych kasach biletowych. W Linuksie porty szeregowe są obsługiwane przez interfejs TTY (**T**ele**TY**pe/**T**ele**TY**pewrite), który pierwotnie był przeznaczony do obsługi właśnie takich, terminali. Jednak rozbudowana architektura oraz konstrukcja sprawiają, że doskonale radzi sobie ona również w dzisiejszych czasach. Na **rysunku 9** przedstawiono architekturę podsystemu TTY.

Architektura składa się z trzech warstw. Zadaniem sterownika modułu jest fizyczna obsługa układu peryferyjnego, oraz udostępnianie jednolitego interfejsu dla modułu dyscypliny linii. Moduł dyscypliny linii (*Line Discipline*), odpowiedzialny jest za określenie sposobu wykorzystania fizycznego portu. Port może być wykorzystany na wiele sposobów, na przykład może być wykorzystany jako klasyczna magistrala, do której dołączone są urządzenia np. mysz szeregowa, a może też pełnić rolę terminala TTY. W wypadku dołączenia myszy, port szeregowy wykorzystany jest wewnętrznie przez sterownik myszy i nie jest udostępniany warstwie TTY. Moduł dyscypliny linii udostępnia zatem dodatkową warstwę abstrakcji, uniezależniającą inne warstwy od sprzętu. W przypadku domyślnej dyscypliny N_TTY, port jest udostępniany war-



Rysunek 9. Architektura systemu TTY na podstawie *http://www.linux.it/~rubini/docs/serial/serial.html*

Listing 17. Definicja struktury *termios*
```
struct termios
{
  tcflag_t c_iflag;  /* input mode flags */
  tcflag_t c_oflag;  /* output mode flags */
  tcflag_t c_cflag;  /* control mode flags */
  tcflag_t c_lflag;  /* local mode flags */
  cc_t c_line;       /* line discipline */
  cc_t c_cc[NCCS];   /* control characters */
  speed_t c_ispeed;  /* input speed */
  speed_t c_ospeed;  /* output speed */
#define _HAVE_STRUCT_TERMIOS_C_ISPEED 1
#define _HAVE_STRUCT_TERMIOS_C_OSPEED 1
};
```

stwie TTY, która z kolei udostępnia go aplikacjom przestrzeni użytkownika w postaci plików urządzeń. W przypadku portów szeregowych RS232 są to odpowiednio pliki /dev/tty**S**x, gdzie **x** określa numer porządkowy portu. W wypadku portów szeregowych dołączanych za pomocą USB, wykorzystujące własne klasy USB będą to pliki /dev/ttyUSB**x** (x=0-n), a w przypadku klasy CDC-ACM będą to pliki /dev/ttyACM**x** (x=0-n).Dzięki dodatkowej pośredniej warstwie abstrakcji , podsystem jest bardzo elastyczny, a jego użycie w aplikacji użytkownika pomimo skomplikowania jest stosunkowo proste. Warto w tym miejscu wspomnieć, że podsystem **TTY** obsługuje także klasyczną konsolę, którą stanowi klawiatura-ekran, a zatem służy on nie tylko do obsługi portów szeregowych.

Użycie portów szeregowych w aplikacjach użytkownika

Jak wspomniano, port szeregowy w Linuksie traktowany jest jako terminal i obsługiwany przez podsystem **tty**, przez co filozofia obsługi jest nieco odmienna od obsługi portów w systemach Windows, które mają dedykowane funkcję API służące do obsługi tylko i wyłącznie portów szeregowych. Jedną z takich odmiennych rzeczy są różne tryby pracy terminala (portu szeregowego), do których należą:

Tryb pracy kanonicznej, gdzie linia **tty** pracuje jak klasyczny terminal, w trybie liniowym, co oznacza że funkcja odczytująca zwróci je, jedynie po odebraniu znaku zakończenia linii. (Znak CR)

Tryb pracy nie-kanonicznej, w którym linia **tty** przekazuje pojedyncze odebrane dane, tak jak one przychodzą z urządzenia zewnętrznego.

Ponieważ w naszych elektronicznych praktykach bardziej naturalny jest tryb pracy niekanonicznej, w dalszych rozważaniach skupimy się tylko na nim. Użycie portu szeregowego w Linuksie jest stosunkowo łatwe i sprowadza się do:

Otwarcia pliku urządzenia portu szeregowego np. **/dev/ttyS0**, za pomocą wywołania systemowego *open()*.

Skonfigurowania terminala (portu), za pomocą wywołania systemowego tcgetattr()/tcsetattr().

Odczytu danych z portu za pomocą wywołań systemowych read()/write(), ewentualnie w połączeniu z wywołaniem systemowym select()/poll(), gdy mamy do czynienia z wieloma deskryptorami, obsługiwanymi w jednym wątku.

Najbardziej skomplikowaną częścią z wyżej wymienionych jest konfiguracja terminala **tty** (portu), która wymaga zmiany kilkunastu ustawień w strukturze **termios**, reprezentującej aktualny stan terminala. Prześledzimy teraz krok po kroku jak przebiega otwarcie oraz konfiguracja portu na przykładzie hipotetycznego portu szeregowego reprezentowanego przez plik */dev/ttyS1*. Pierwszą czynnością będzie otwarcie portu za pomocą wywołania *int serfd = open(port ,O_RDWR | O_NOCTTY);*

Ustawienie flagi **O_NOCTTY** jest konieczne i informuje o tym, aby w wypadku braku przypisanego domyślnego terminala do procesu, port nie został wykorzystany jako terminal domyślny. Po otwarciu portu szeregowego, należy pobrać aktualną konfigurację terminala tty do struktury **termios**, za pomocą wywołania :

```
struct termios options;
int err = tcgetattr( serfd, &options );
error(err<0);
```

Definicję struktury *termios* umieszczono na **listingu 17**. Pole *c_iflag*, określa sposób przetwarzania znaków otrzymanych przez urządzenie, pole *c_oflag* określa sposób traktowania znaków wysłanych do urządzenia. Pole *c_cflags* zawiera flagi sterujące, które definiują techniczne parametry pracy urządzenia terminalowego. Pole *c_lflag* określa tryb pracy działania dyscypliny linii związanej z urządzeniem terminalowym. Tablica ***c_cc*** zawiera znaki specjalne określające zachowanie terminala w trybie kanonicznym oraz ustawienia trybu niekanonicznego, o czym dalej. Pola *c_ispeed* oraz *c_ospeed* zawierają prędkość wejściową i wyjściową urządzenia terminalowego.

Po odczytaniu aktualnych ustawień terminala należy je odpowiednio zmodyfikować, aby uzyskać tryb pracy niekanonicznej, wraz z zadanymi parametrami transmisji. Na początku za pomocą funkcji *setispeed* oraz *setospeed* należy ustawić prędkość pracy portu szeregowego np. na 115200bps:

```
err = cfsetispeed( &options, B115200 );
error(err<0);
err = cfsetospeed( &options, B115200 );
error(err<0);
```

W pliku nagłówkowym *<termios.h>* jest zdefiniowanych szereg stałych rozpoczynających się od dużej litery „B", określających wiele popularnych prędkości transmisji. Kolejną czynnością jest ustawienie odpowiednich flag, sterujących, tak aby uzyskać niekanoniczny tryb pracy, oraz skonfigurować pozostałe parametry portu, czyli ilość bitów danych, kontrolę parzystości, ilość bitów stopu. Na przykład poniższa konfiguracja ustawia terminal w tryb pracy 8 bitów danych, 1 bit stopu, brak kontroli parzystości, oraz brak kontroli przepływu (**listing 18**).

Wyzerowanie flagi *PARENB* skutkuje wyłączeniem generowania i sprawdzania bitu kontroli parzystości. Jeżeli ta flaga jest ustawiona, a dodatkowa flaga *PAREODD* jest wyzerowana, wówczas kontrola parzystości jest nieparzysta, w przeciwnym wypadku parzysta. Pole *CS8* ustala liczbę bitów danych na 8. Inne dopuszczalne ustawienia to *CS5...CS8*. Wyzerowanie flagi *CRTSCTS* oznacza brak sprzętowego sterowania przepływem. Ustawienie flagi *CREAD* powoduje, że port szeregowy będzie przyjmował dane. Wyzerowanie flag *~(ICANON | ECHO | ECHOE | ISIG)*, powoduje wyłączenie trybu kanonicznego, a zatem terminal tty nie będzie interpretował odebranych znaków. Wyzerowanie flag *INPCK* i *ISTRIP* powoduje, że kontrola parzystości odebranych znaków nie będzie prowadzona, a dane będą przekazywane do bufora odbiorczego niezależnie od rezultatu kontroli parzystości. Wyzerowanie flag *ICRNL | INLCR* spowoduje nieprzetwarzanie znaków końca linii, natomiast wyzerowanie flagi *IUCLC* spowoduje wyłączenia zamiany małych liter na duże podczas wysyłania. Na zakończenie są zerowane flagi, odpowiedzialne za programową kontrolę przepływu oraz flaga *OPOST* zabraniająca przetwarzania znaków do wysłania. Jak więc widzimy, flagi sterujące umożliwiają zmianę wielu różnych parametrów, nie tylko związanych z portem szeregowym, ale i z samym terminalem. Po ustawieniu trybu niekanonicznego, bez dodatkowego przetwarzania znaków, na koniec należy ustawić jeszcze parametry określające minimalną liczbę znaków , oraz czas międzyznakowy, który będzie powodował wyjście z uśpienia blokującej funkcji read() i zwrócenie przez nią odebranych danych.

```
//Set char and interchar timeout 0ms 1chars
options.c_cc[VTIME] = 0;
options.c_cc[VMIN] = 1;
```

Pole *c_cc[VTIME]* określa czas między znakowy wyrażony jednostkach będących wielokrotnością 100 ms, po przekroczeniu którego funkcja *read()* zwróci odczytane dane (*interchar-*

Listing 18. Ustalenie ramki transmisji RS232
```
//8N1 frame
options.c_cflag &= ~PARENB;
options.c_cflag &= ~CSTOPB;n
options.c_cflag &= ~CSIZE;
options.c_cflag |= CS8;
//disable hardware flow control
options.c_cflag &= ~CRTSCTS;
//Enable receiver and local mode
options.c_cflag |= (CLOCAL | CREAD);
//Raw input
options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);
//Disable parity checking
options.c_iflag &= ~(INPCK | ISTRIP);
//Don't map CR to NL or NL to CR
options.c_iflag &= ~(ICRNL | INLCR);
//Don't map uppercase to lowercase
options.c_iflag &= ~IUCLC;
//Don't ignore CR
options.c_iflag &= ~IGNCR;
//Ignore BREAK condition
options.c_iflag |= IGNBRK;
//Disable software flow control
options.c_iflag &= ~(IXON | IXOFF | IXANY);
//Raw output - other c_oflag bits ignored
options.c_oflag &= ~OPOST;
```

timeout), natomiast pole *c_cc[VMIN]* zawiera minimalną liczbę znaków, po przekroczeniu której funkcja *read()* wyjdzie z uśpienia i zwróci dane. Sterując tymi polami, mamy możliwość wpływu na sposób zachowania się procedury odbierania danych. Wpisanie w pola *VMIN*, oraz *VIME* wartości 0 spowoduje, że wywołania *read()*, oraz *write()* będą nieblokujące.

Jednak metoda cyklicznego sprawdzania stanu nie jest najlepszym rozwiązaniem w przypadku środowiska wieloprogramowego, dlatego w wypadkach, gdy istnieje konieczność obsługi wielu deskryptorów należy posłużyć się wywołaniem *poll()* lub *select()*. Po ustawieniu parametrów, należy wywołać funkcję *tcsetattr*, która spowoduje ustawienie zadanych parametrów w urządzeniu tty:

err = tcsetattr(serfd, TCSANOW, &options);
error(err<0);

Jest to ostatnia czynność, po której urządzenie tty będzie zachowywało się jak klasyczny port szeregowy, do którego jesteśmy przyzwyczajeni z innych systemów operacyjnych.

Przykład praktyczny

Płytka prototypowa **BF210** posiada wyprowadzone 3 porty szeregowe, 2 szt. RS232 oraz 1 szt. RS485. Jeden z portów **RS232** pełni rolę konsoli szeregowej aplikacji, dlatego do dyspozycji mamy jeden port **RS232** mapowany jako */dev/ttyS1* , oraz port szeregowy **RS485**, z automatyczną zmiana kierunku mapowany jako */dev/ttyS2*. Demonstrując użycie praktyczne portu pokażemy przykład prostego konwertera RS232,–>TCPIP, który będzie udostępniał port szeregowy */dev/ttyS1* (złącze J6) pracujący w konfiguracji 115200, n, 8, 1 oczekując na nadchodzące połączenia na porcie TCP 7900. Napisanie podobnej aplikacji na mikrokontroler jednoukładowy wymagało by dużego nakładu pracy, a tutaj taka aplikacja składa się dosłownie z kilkudziesięciu linii kodu. Przykład ten może mieć praktyczne zastosowanie po rozbudowaniu o autentykację użytkownika, szyfrowanie danych SSL, czy możliwość zdalnej konfiguracji parametrów portu szeregowego. Działanie aplikacji może być sprzetestowane z wykorzystaniem komputera PC, gdzie do portu RS232 podłączymy się programem terminalowym, natomiast na port TCP 7900 podłączymy się za pomocą programu telnet.

Opis aplikacji

Kod źródłowy aplikacji konwertera RS232<–>TCPIP znajduje się w pliku źródłowym *rstoeth.c*, który następnie jest kompilowany do pliku wykonywalnego **rstoeth**. Konfiguracja aplikacji zawarta jest na początku pliku za pomocą dyrektyw #define, gdzie mamy możliwość zmiany aktualnego portu szeregowego, prędkości transmisji, czy portu TCP. W praktycznym przykładzie należy uzupełnić aplikacje o odczyt parametrów z pliku konfiguracyjnego, czy protokół zdalnej zmiany ustawień portu szeregowego. Program rozpoczyna działanie od wykonania pętli main() (listing n)

Listing 19. Przykładowy program obsługi UART
```
int main(void)
{
  int lsock = create_listening_socket( LISTEN_PORT );
  int serial = open_serial_port( SERIAL_DEVICE, SERIAL_SPEED );
  printf("Waiting for device connection on port %d    .\n",LISTEN_PORT);
  for(;;)
  {
    //Accept the connection from incomming socket
    int conn_sock = accept(lsock,NULL,NULL);
    printf("Connected from remote...\n");
    struct pollfd fds[] =
    {
      { serial, POLLIN|POLLERR, 0 },
      { conn_sock, POLLIN|POLLERR, 0 }
    };
    //Internal process loop
    for(bool no_disc=true;no_disc;)
    {
      //Poll syscall sleep for data arriving
      int res = poll( fds, sizeof(fds)/sizeof(fds[0]), -1 );
      //Check error
      error(res<0);
      //Scan active fds
      for( unsigned p=0; p<sizeof(fds)/sizeof(fds[0]); p++)
      {
        //If no revents continue
        if(fds[p].revents==0) continue;
        //If error (exit )
        error( fds[p].revents & POLLERR );
        //If input data
        if(fds[p].revents & POLLIN )
        {
          //Got data from serial write to network
          if(fds[p].fd == serial)
          {
            //Got data from serial port write it to remote host
            fds_copy( conn_sock, serial );
          }
          //Got data from network write to serial
          if(fds[p].fd == conn_sock)
          {
            //Got data from network write it to serial port
            if( fds_copy( serial, conn_sock ) == 0 )
            {
              //If socket closed connection disconnect
              no_disc = false;
              close( conn_sock );
              printf("Host disconnected ...\n");
              break;
            }
          }
        }
      }
    }
    //Close open sockets
  close( lsock );
  close( serial );
  return 0;
}
```

Listing 20. Struktura *pollfd*
```
struct pollfd
  {
    int fd;                     /* File descriptor to poll.  */
    short int events;           /* Types of events poller cares about.  */
    short int revents;          /* Types of events that actually occurred.  */
  };
```

Listing 21. Funkcja fds_copy
```
//Copy data from one file descriptor to another
static int fds_copy(int to_fds, int from_fds)
{
  char trx_buf[4096];
  //Read data from first fds
  int rd = read( from_fds, trx_buf, sizeof(trx_buf) );
  error(rd < 0);
  //Write data to second fds
  int wr = 0;
  while( rd > 0)
  {
    int wres = write( to_fds, &trx_buf[wr], rd );
    error(wres<0);
    wr += wres;
    rd -= wr;
  }
  return wr;
}
```

Na początku wywoływana jest funkcja *create_listening_socket()*, która jest odpowiedzialna za utworzenie słuchającego gniazda na porcie przekazanym jako argument. Kolejną czynnością jest otwarcie portu szeregowego co jest realizowane za pomocą funkcji *open_serial_port()*, po czym, program wchodzi do pętli głównej, gdzie wywoływana jest funkcja systemowa *accept()*. Funkcja ta blokuje się w oczekiwaniu na nadchodzące połączenia, i w przypadku nawiązania

połączenia zwraca deskryptor gniazda nowego połączenia. Po wywołanie funkcji *accept()*, następuje utworzenie tablicy struktur *pollfd* dla wywołania systemowego *poll()*. Strukturę *pollfd* pokazano na **listingu 20**.

Pole *fd* zawiera deskryptor pliku, pole *events* zawiera flagi wejściowe np. *POLLIN, POLLOUT, POLLER* określające zdarzenia, na które oczekujemy na danym deskryptorze. Pole *revent* jest zwracane przez funkcję systemową *poll()* i informuje, jakie zdarzenie miało miejsce na danym deskryptorze. W naszym przypadku tworzymy tablicę dwuelementową do której wpisano deskryptory od gniazda połączenia i portu szeregowego, a w polach *events* ustawiono flagi *POLLIN|POLLER*, co będzie owocowało wybudzeniem funkcji systemowej *poll()* w momencie odebrania danych lub błędu na którymkolwiek z deskryptorów. Następnie program wchodzi do pętli wewnętrznej, której głównym elementem jest wywołanie systemowe *poll()*, które blokuje się w oczekiwaniu na nadejście danej z deskryptora portu szeregowego lub deskryptora gniazda połączenia. Jeżeli wystąpią dane odebrane na którym z deskryptorów, *pool()* wyjdzie z uśpienia i ustawi odpowiednie pola *revents* w tablicy struktur *fds*. Program sprawdza zawartość pół *revents* w tablicy *fds* i w wypadku wykrycia zdarzenia *POLLERR* kończy działanie wychodząc z błędem. Natomiast w przypadku wystąpienia zdarzenia *POLLIN*, na danym deskryptorze wywoływana jest funkcja *fds_copy* (**listing 21**), która kopiuje dane z jednego deskryptora do drugiego.

Funkcja ta jako pierwszy argument przyjmuje deskryptor , gdzie będą skopiowane odebrane dane, natomiast jako drugi argument przyjmuje deskryptor skąd będą skopiowane dane. Działanie tej funkcji sprowadza się do wywołania funkcji systemowej *read()*, która odczyta zgromadzone dane, ale nie zablokuje się, ponieważ jest wiadome dzięki wywołaniu *poll()*, że deskryptor zawiera dane do odczytania. Następnie tak odebrane dane są kopiowane za pomocą wywołania *write()* do deskryptora docelowego. Funkcja *write()* jest wywoływana w pętli, tak aby zapisać wszystkie oczekujące dane, ponieważ zgodnie ze specyfikacją może ona zapisać mniej danych niż żądano. Rezultatem działania funkcji jest zwrócenie liczby zapisanych bajtów. W wypadku, gdy zdalny host zakończy połączenie, wywołanie *poll()* zwróci w polu *revents* deskryptora gniazda flagę *POLLIN*, natomiast wywołanie *read()* zwróci wartość 0, co pozwoli na wykrycie momentu zakończenie połączenia. W momencie wykrycia rozłączenia pętla wewnętrzna jest opuszczana w wyniku czego zostanie ponownie wywołana funkcja systemowa *accept()* oczekująca na ponowne nawiązanie połączenia.

Uruchomienie przykładu

Skompilowany przykład wraz systemem dostarczany jest w postaci pliku obrazu *example5.img*. Przykład należy nagrać na kartę SD tak jak opisano w pierwszym odcinku. Po wgraniu przykładu na kartę należy BF210 dołączyć do sieci, uruchomić i zalogować się na konsoli szeregowej. Po zalogowaniu się należy wydać polecenie *ifconfig eth0*, w wyniku czego powinniśmy otrzymać informację o aktualnie przydzielonym adresie z serwera DHCP adresie IP. Adres ten należy zapamiętać celem późniejszego wykorzystania do połączenia z BF210 za pomocą programu *telnet*. Gdy już odczytaliśmy aktualny adres IP, należy uruchomić program wydając polecenie *rstoserial***.** Program powinien uruchomić się i poinformować o oczekiwaniu na połączenie:

root@bf210-at91:~# rstoeth

Waiting for device connection on port 7900

...

Gdy program jest uruchomiony należy przełączyć gniazdo portu szeregowego na J6, (lub użyć drugiego portu szeregowego w komputerze), tak abyśmy mogli wysyłać i odbierać dane z tego portu. Następie na komputerze należy wydać polecenie *telnet [zapamietany adres ip] 7900*. Program powinien dołączyć się do płytki BF210 przez port 7900. Po połączeniu możemy wpisywać znaki w oknie programu *telnet* i obserwować odebrane znaki w programie terminalowym portu szeregowego, i odwrotnie – wpisywać znaki w programie terminalowym, do którego jest dołączony port szeregowy BF210 i obserwować odebrane dane w programie *telnet*.

Lucjan Bryndza, EP