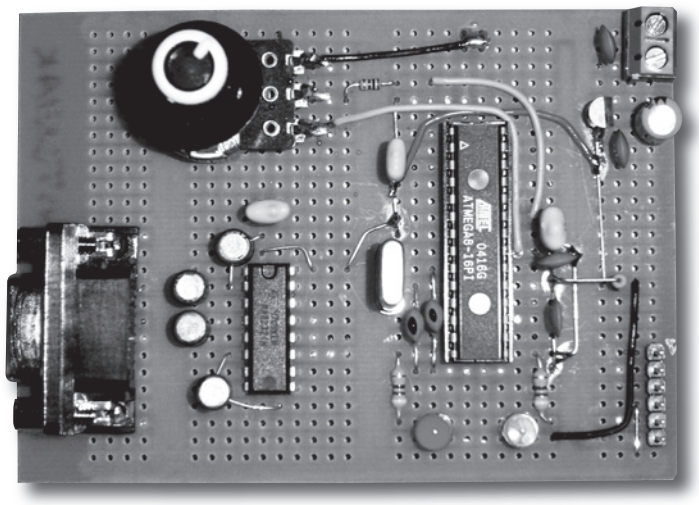
Programowanie portu szeregowego w systemach operacyjnych Linux i Windows, część 1

Umiejętność programowej obsługi interfejsu RS232 od strony komputera PC jest dziś istotnym elementem elektronicznego rzemiosła. W kolejnych częściach niniejszego kursu piszemy jak w praktyce oprogramować port szeregowy w środowiskach Linux i Windows. Wiele miejsca poświęcamy pisaniu przenośnych aplikacji GUI, które korzystają z interfejsu szeregowego i zachowują się tak samo w systemach Windows jak i Linux. Wszystkie omawiane zagadnienia poparte są szczegółowo opisanymi praktycznymi przykładami.

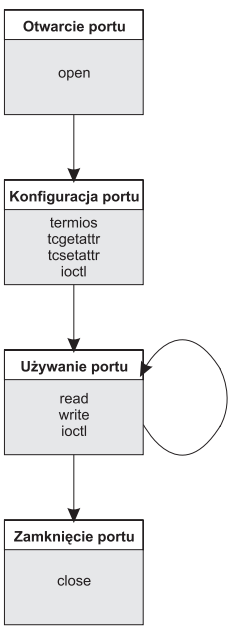


Popularność Linuksa w ostatnich latach wyraźnie wzrosła i wiele wskazuje na to, że zjawisko to będzie się z roku na rok nasilać. Choć dla nas – elektroników – podstawową platformą pozostaje Windows, to powstaje coraz więcej programów narzędziowych, kompilatorów dla mikrokontrolerów i innych programów, które używamy w swojej codziennej praktyce. Jednym z najznamienitszych przykładów jest słynny kompilator języka C dla mikrokontrolerów AVR (AVRGCC), czy ARM. Być może w najbliższym czasie Linux nie zawojuje zupełnie naszego świata, ale z pewnością stanie się w nim bardziej obecny niż dziś. Gdy elektronik decyduje się napisać własne oprogramowanie, to chyba najistotniejszym zagadnieniem z jakim musi się uporać staje się oprogramowanie portów komputera PC. W tym kursie bierzemy pod lupę port szeregowy i patrzymy jak obsługuje się go w Linuksie i pod Windows.

Cel niniejszego kursu jest dwojaki. Po pierwsze, ma on na praktycznych przykładach pokazać jak można oprogramować port szeregowy w Linuksie oraz – niejako przy okazji – w systemie Windows. Nie twierdzę przy tym, że omówię wszystkie szczegóły i szczególiki związane z oprogramowaniem RS232 w każdym z tych systemów. Pokażę natomiast w praktyce jak wykorzystać interfejs szeregowy komputera PC w najbardziej typowych zastosowaniach. Drugim istotnym zagadnieniem jakie przedstawię jest pisanie aplikacji przenośnych, które korzystają z obsługi RS232 i posiadają graficzny interfejs użytkownika (*GUI – Graphical User Interface*). Jest to zagadnienie, które jest obecnie „na topie” wśród zagadnień związanych z tematyką tworzenia oprogramowania. Istnieje wyraźny trend, aby pisane aplikacje posiadały swoje wersje nie tylko dla Windows, ale także dla Linuksa i innych systemów operacyjnych. Powinny przy tym, w miarę możliwości, wyglądać i zachowywać się tak samo pracując pod kontrolą każdego z nich. Problem ten omówię w dalszych częściach kursu. Pisząc o nim zachowam kierunek migracji z Windows na Linuksa, gdyż właśnie Windows pozostaje platformą „bazową”, w której większość z nas porusza się znacznie sprawniej niż w Linuksie. Innymi słowy, będziemy pisać aplikacje GUI pod Windows ale tak, aby dały się łatwo przenieść na Linuksa, a potem je przeniesiemy. Wszystko to będzie zilustrowane praktycznymi przykładami. Jednym z nich będzie cyfrowy woltomierz odczytywany przez interfejs RS232C. Dodatkową korzyścią jaką otrzymają Czytelnicy będzie przykładowa klasa obsługująca RS232 zaimplementowana w języku C++, która wystąpi w dwóch wersjach: dla Windows i dla Linuksa. Klasa ta będzie miała identyczny interfejs dla obu tych platform, dzięki czemu będzie ją można stosować w obu systemach bez konieczności dokonywania jakichkolwiek zmian w programie głównym (korzystającym z tej klasy). Jej wykorzystanie będzie możliwe w każdym środowisku programistycznym zawierającym kompilator C++, niezależnie od tego czy do tworzenia GUI wykorzystamy MFC, Qt, VCL/CLX, dowolny toolkit C++, czy też jakiekolwiek inne oprogramowanie.

Porty w Linuksie

Nie jest przesadą stwierdzenie, że w Linuksie wszystko jest plikiem. W istocie, niemal wszystkie „byty” z jakimi mamy do czynienia powiązane są w ten, czy inny sposób z jakimś plikiem. Nie inaczej jest z portami. Wszyst-



Rys. 1. Typowy cykl pracy portu szeregowego (Linux)

Tab. 1. Pola struktury termios

Pole	Typ	Opis
c_cflag	unsigned int	Opcje sterowania
c_lflag	unsigned int	Opcje lokalne (dyscyplina linii)
c_iflag	unsigned int	Opcje wejścia
c_oflag	unsigned int	Opcje wyjścia
c_cc	unsigned char*	Tablica znaków specjalnych
c_ispeed	unsigned int	Prędkość wejściowa
c_ospeed	unsigned int	Prędkość wyjściowa

kie urządzenia zewnętrzne, do których zaliczają się porty komputera, reprezentowane są przez specjalne **pliki urządzeń** znajdujące się w katalogu */dev*. Plików tych jest bardzo dużo, ale nas interesuje tylko kilka z nich. Przykładowo – porty równoległe (o ile jest ich więcej niż jeden, co się właściwie nie zdarza) reprezentowane są przez pliki */dev/lp0*, */dev/lp1* itd.. Jeśli posiadamy tylko jeden port równoległy, to związany jest z nim plik */dev/lp0* (oczywiście przyporządkowanie to może być zmienione). W przypadku portów szeregowych domyślne przyporządkowanie jest następujące:

COM1 (adres 0x3f8) – plik */dev/ttyS0*
COM2 (adres 0x2f8) – plik */dev/ttyS1*
COM3 (adres 0x3e8) – plik */dev/ttyS2*
COM4 (adres 0x2e8) – plik */dev/ttyS3*

W niektórych starszych dystrybucjach Linuksa noszą one nazwy cua0...cua3.

Aby dowiedzieć się o nich więcej, przejdźmy do katalogu */dev* (komenda *cd /dev* z dowolnej lokacji) i wpiszmy *ls ttyS0 –l*. Komenda ta pokaże nam podstawowe właściwości pliku *ttyS0*. Oto przykładowy wynik jej działania:

```
crw–rw–rw–   1 root   uucp   4,
    64   kwi  14  2001  ttyS0
```

Spójrzmy na pierwsze pole z lewej określające rodzaj pliku i prawa dostępu do niego. Litera c wskazuje, że jest to **plik znakowy** (*char*). Oznacza to, że dostęp do reprezentowanego przez ten plik urządzenia odbywa się „znak po znaku”, czyli „bajt po bajcie”. Następne 9 znaków określa prawa dostępu, co objaśniono poniżej:

pierwsze 3 litery – prawa właściciela pliku (*u – user*);

drugie 3 litery – prawa grupy (*g – group*);

trzecie 3 litery – prawa pozostałych użytkowników (*o – other*);

gdzie:

r – prawo do czytania (*Read*);

w – prawo do pisania (*Write*);

x – prawo do wykonywania (*eXecute*).

W nowo zainstalowanym systemie zwykle jest tak, że właściciel (w tym przypadku *root*) ma pełne prawa pisania i czytania *rw–*, zaś zarówno grupa jak i pozostali użytkownicy nie mają żadnych praw – nie mogą ani pisać do portu, ani z niego czytać. Sytuacja taka uniemożliwia oczywiście uruchomienie jakichkolwiek aplikacji korzystających z portu szeregowego w trybie zwykłego użytkownika. Aby to zmienić, należy dodać im prawa pisania i czytania. W tym celu lo-

Tab. 2. Opcje funkcji tcsetattr

Opcja	Opis
TCSANOW	Dokonaj zmiany natychmiast
TCSADRAIN	Dokonaj zmiany po zakończeniu transmisji danych
TCSAFLUSH	Wyczyść (*flush*) bufory wejściowy i wyjściowy, po czym dokonaj zmiany

List. 1. Przykład otwarcia portu ttyS0

```
//*********************************************************
//
//    Port ttyS0 (COM1) opening example
//    Returns: 0 when error occured
//             1 when everything OK
//
//*********************************************************
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
#include <errno.h>

//File descriptor
int fd;

int Open(void)
{
    fd=open(„/dev/ttyS0”, O_RDWR | O_NOCTTY | O_NDELAY);
    if (fd<0)
    {
        //Opening error
        return 0;
    }
    else
    {
        //Here port configuration
        //...
    }
    return 1;
}
```

List. 2. Ogólny schemat konfiguracji portu szeregowego

```
//Copy of termios structure
struct termios options;

//Getting the current settings for the port
tcgetattr(fd, &options);


//Setting baudrate (19200 for example)
cfsetispeed(&options, B19200);
cfsetospeed(&options, B19200);

//Modifying c_cflag by bitwise OR and AND
options.c_cflag &= ...;
options.c_cflag |= ...;

//Modifying c_lflag by bitwise OR and AND
options.c_lflag &= ...;
options.c_lflag |= ...;

//Modifying c_iflag by bitwise OR and AND
options.c_iflag &= ...;
options.c_iflag |= ...;

//Modifying c_oflag by bitwise OR and AND
options.c_oflag &= ...;
options.c_oflag |= ...;


//Setting the new settings for the port immediately
tcsetattr(fd, TCSANOW, &options);
```

Tab. 3. Predefiniowane stałe dla pola c_cflag

Stała	Opis
CBAUD	Maska bitowa dla bitów określających prędkość (obecnie używanie niezalecane)
B0	0 bodów (wyłącz DTR)
B50	50 bodów
B75	75 bodów
B110	110 bodów
B134	134 body
B150	150 bodów
B200	200 bodów
B300	300 bodów
B600	600 bodów
B1200	1200 bodów
B1800	1800 bodów
B2400	2400 bodów
B4800	4800 bodów
B9600	9600 bodów
B19200	19200 bodów
B38400	38400 bodów
B57600	57600 bodów
B76800	76800 bodów
B115200	115200 bodów
EXTA	Zewnętrzny zegar taktujący
EXTB	Zewnętrzny zegar taktujący
CSIZE	Maska bitowa dla bitów określających liczbę bitów danych
CS5	5 bitów danych
CS6	6 bitów danych
CS7	7 bitów danych
CS8	8 bitów danych
CSTOPB	Flaga włączona: 2 bity stopu Flaga wyłączona: 1 bit stopu
CREAD	Włącz odbiornik
PARENB	Włącz kontrolę parzystości
PARODD	Flaga włączona: odd parity Flaga wyłączona: even parity
HUPCL	Wystaw 0 na DTR przy zamknięciu przez ostatni proces
CLOCAL	Linia lokalna – nie można zmienić aktualnego właściciela portu
CNEW_ RTSCTS CRTSCTS	Włącz sprzętową kontrolę przepływu (hardware flow control)

Tab. 4. Predefiniowane stałe dla pola c_lflag

Stała	Opis
ISIG	Włącz wysyłanie sygnałów SIGINTR, SIGSUSP, SIGDSUSP i SIGQUIT
ICANON	Flaga włączona: canonical input Flaga wyłączona: raw input
XCASE	Mapuj wielkie litery na małe (przestarzała)
ECHO	Włącz wysyłanie odebranych znaków (echo)
ECHOE	Po odebraniu znaku kasowania (erase) wyślij kombinację znaków BS-SP-BS.
ECHOK	Wyślij znak NL (0x0A) po odebraniu znaku kasowania wiersza (kill)
ECHONL	Odsyłaj znak NL (echo)
NOFLSH	Wyłącz czyszczenie bufora wejściowego po przerwaniu lub odebraniu znaku przerywającego quit
IEXTEN	Włącz rozszerzone funkcje portu (dot. dyscypliny linii)
ECHOCTL	Odsyłaj znaki kontrolne (echo) jako kombinację ^znak
ECHOPRT	Poprzedź usuwane znaki znakiem \\ (backslash)
ECHOKE	Echem znaku kasowania wiersza jest odpowiednia kombinacja znaków SP i BS

Tab. 5. Predefiniowane stałe dla pola c_iflag

Stała	Opis
INPCK	Włącz kontrolę parzystości
IGNPAR	Ignoruj błędy parzystości
PARMRK	Zaznaczaj błędy parzystości
ISTRIP	Zeruj bit parzystości (po kontroli parzystości)
IXON	Włącz programową kontrolę przepływu (wychodzącą)
IXOFF	Włącz programową kontrolę przepływu (wchodzącą)
IXANY	Dowolny znak (nie tylko VSTART) włącza przepływ danych
IGNBRK	Ignoruj znak przerwania
BRKINT	Wyślij sygnał SIGINT gdy odebrano znak przerwania
INLCR	Mapuj znak NL (0x0A) na CR (0x0D)
IGNCR	Ignoruj znak CR
ICRNL	Mapuj znak CR na NL
IUCLC	Mapuj wielkie litery na małe (jeśli ustawiona flaga IEXTEN w c_lflag)
IMAXBEL	Włącz sygnał dźwiękowy (bell) przy przepełnieniu bufora wejściowego

gujemy się jako *root* (poleceniem *su*), po czym będąc w katalogu */dev* wpisujemy:

chmod go+rw ttyS0

Komenda ta dodaje prawa pisania do portu i czytania z niego wszystkim użytkownikom. Dopiero teraz można uruchamiać programy używające RS232, także w trybie zwykłego użytkownika.

Cykl pracy portu szeregowego

Wiemy już z grubsza jak w systemie plików Linuksa reprezentowane są porty szeregowe. Trzeba jeszcze wiedzieć, co można z nimi robić, czyli jakie operacje wykonywane są na tych plikach w celu realizacji transmisji pomiędzy PC, a dołączonym do niego urządzeniem. Na **rys. 1** przedstawiono typowy cykl pracy portu szeregowego. Wyróżniłem na nim cztery główne fazy pracy oraz związane z nimi słowa kluczowe – nazwy funkcji, struktur itp. Fazy te są następujące:

1. Otwarcie portu
   Otwarcie portu równoznaczne jest z otworzeniem pliku ttySx do pisania i/lub czytania. My rozważymy jedynie sytuacje, w których port otwarty jest zarówno do czytania jak i do pisania (transmisja dwukierunkowa).
2. Konfiguracja portu
   Jest to niezwykle istotna faza pracy z interfejsem szeregowym. Tutaj konfigurujemy port tak, aby spełniał wymagania jakie nakłada na niego nasza aplikacja. Trzeba zaznaczyć, że w Linuksie istnieje ogromna liczba opcji konfiguracji, a od ich prawidłowego wybrania zależy poprawność pracy pisanego oprogramowania. W tej części kursu omówię najistotniejsze z nich.
3. Używanie portu (pisanie, czytanie, funkcje specjalne itp.)
   W fazie „używania portu" realizuje się komunikację PC z urządzeniem zewnętrznym, czyli protokół komunikacyjny. Program pozostaje w tej fazie przez niemal cały czas działania aplikacji.
4. Zamknięcie portu
   Równoznaczne z zamknięciem pliku ttySx.

Otwieranie portu

Port szeregowy, podobnie jak każdy inny plik, może być otworzony za pomocą funkcji *open*. Przyjmuje ona dwa argumenty:

- pierwszy argument zawiera pełną ścieżkę dostępu do pliku portu (w przypadku portu COM1 będzie to */dev/ttyS0*),
- drugi argument określa opcje otwierania portu, które można zmieniać wykorzystując predefiniowane maski bitowe.

Wartością zwracaną jest deskryp-

List. 3. Pobranie liczby bajtów pozostających w buforze wejściowym

```
//**********************************
//
//    Getting number of bytes
//    available in inpu queue
//
//**********************************
#include <unistd.h>
#include <termios.h>

//File descriptor
int fd;

//Bytes available in input queue
int bytes;

//Get the number of bytes available
ioctl(fd, FIONREAD, &bytes);
```

tor pliku portu, który później (przy czytaniu lub pisaniu) służy jako jego identyfikator. Jeśli otwarcie przebiegło pomyślnie deskryptor ma wartość dodatnią. W przypadku błędu otwarcia, którym może być brak odpowiednich praw dostępu lub używanie portu przez inną aplikację, funkcja *open* zwraca wartość mniejszą od 0. Przykład jej użycia znajduje się na **list. 1**. Znaczenie wykorzystanych opcji jest następujące:

O_RDWR – otwarcie portu do czytania i do pisania;

O_NOCTTY – nie ustawienie tej opcji sprawi, że inne urządzenia mogą mieć wpływ na pracę otwieranego portu;

O_NDELAY – ustawienie tej flagi oznacza, że program nie reaguje na stan linii DCD. Nie ustawienie tej flagi spowoduje, że program będzie uśpiony dopóki linia DCD nie osiągnie stanu logicznego 0 (*space*).

W typowych zastosowaniach otwieranie portu z innymi ustawieniami nie ma większego sensu. Nie będziemy więc szczegółowo analizować znaczenia innych opcji. Oczywiście, jeśli ktoś miałby potrzebę skorzystania z nich (i wiedziałby co robi), może ich użyć.

Konfiguracja portu

Gdy port szeregowy jest otwarty należy go odpowiednio skonfigurować. Sterownik linuksowy zapewnia nam ogromną liczbę opcji, z których duża część służy do tego, aby ułatwić używanie RS232 do przesyłania danych w sposób zorientowany liniowo (przydatne na przykład w komunikacji z drukarkami). Dla nas takie działanie jest niepożądane – chcemy mieć pełną kontrolę nad tym co wysyłamy, a także chcemy mieć pewność, że to co odczytujemy z bufora wejściowego jest naprawdę tym co zostało odebrane łączem szeregowym. Na przykład, nie zależy nam na tym, aby sterownik ignorował znak powrotu karetki CR (0x0D), gdyż dla nas może to być istotna dana pomiarowa. Podobnie nie chcemy, aby mapował małe litery na wielkie i odwrotnie. Kolejną przyczyną, dla której nie skorzystamy z większości opcji jakie zapewnia nam Linux jest fakt, że nie są one dostępne pod Windows i jako takie tylko przeszkadzałyby w przenoszeniu aplikacji okienkowych z systemu Windows do Linuksa. Obowiązuje tu oczywiście ta sama zasada co przy otwieraniu portu – kto chce może z nich skorzystać.

Konfiguracja portu odbywa się poprzez modyfikowanie zawartości pól struktury *termios*. Jest to struktura zdefiniowana w pliku *termios.h*

Tab. 6. Predefiniowane stałe dla pola c_oflag

Stała	Opis
OPOST	Przetwarzaj znaki przed wysłaniem (Sposób przetwarzania określają pozostałe poniższe stałe. Są one ignorowane gdy flaga OPOST nie jest ustawiona)
OLCUC	Mapuj małe litery na wielkie
ONLCR	Mapuj znak NL (0x0A) na parę CR-NL (0x0D-0x0A)
OCRNL	Mapuj znak CR na znak NL
ONLRET	Włączona: znak NL powoduje automatyczny powrót karetki
NLDLY	Maska bitowa dla opóźnienia przy przejściu do nowego wiersza (przestarzałe)
NL0	Brak opóźnienia
NL1	Odczekaj 100 ms po przejściu do nowej linii
CRDLY	Maska bitowa dla flag CR0...CR3 – opcji opóźnień przy powrocie karetki (przestarzałe)
CR0	Brak opóźnienia dla znaku CR
CR1	Opóźnienie po znaku CR zależy od aktualnej pozycji w kolumnie
CR2	Odczekaj 100 ms po wysłaniu znaku CR
CR3	Odczekaj 150 ms po wysłaniu znaku CR
TABDLY	Maska bitowa dla flag TAB0...TAB3 – opcji opóźnień po znaku tabulacji (przestarzałe)
TAB0	Brak opóźnienia
TAB1	Opóźnienie po znaku TAB zależy od aktualnej pozycji w kolumnie
TAB2	Odczekaj 100 ms po wysłaniu znaku TAB
TAB3	Rozszerz znaki tabulacji do znaków spacji (SP)
BSDLY	Maska bitowa dla flag BS0...BS3 – opcji opóźnień po znaku backspace (BS) (przestarzałe)
BS0	Brak opóźnienia
BS1	Odczekaj 50 ms po wysłaniu znaku BS
VTDLY	Maska bitowa dla flag VT0...VT3 – opcji opóźnień po znaku VT (przestarzałe)
VT0	Brak opóźnienia
VT1	Odczekaj 2 s po wysłaniu znaku BS
FFDLY	Maska bitowa dla flag VT0...VT3 – opcji opóźnień po znaku 0xFF (przestarzałe)
FF0	Brak opóźnienia
FF1	Odczekaj 2 s po wysłaniu znaku 0xFF

Tab. 7. Tablica znaków specjalnych c_cc (* – patrz opis)

Stała	Opis	Kombinacja przycisków
VINTR	Przerwanie (interrupt)	CTRL-C
VQUIT	Koniec (quit)	CTRL-Z
VERASE	Kasuj znak (erase)	Backspace (BS)
VKILL	kasuj linię	CTRL-U
VEOF	Koniec linii	CTRL-D
VEOL	Koniec linii – powrót karetki	CR
VEOL2	Nowa linia	LF
VMIN	Minimalna liczba bajtów do odczytania*	-
VSTART	Wznów transmisję	CTRL-Q (XON)
VSTOP	Wstrzymaj transmisję	CTRL-S (XOFF)
VTIME	Czas oczekiwania na blok danych (wyrażony w dziesiątych częściach sekundy)*	-

Tab. 8. Wartości parametru cmd funkcji ioctl

Wartość	Opis	Funkcja POSIX (options – struktura termios)
TCGETS	Pobranie aktualnych ustawień portu	tcgetattr
TCSETS	Ustaw nowe ustawienia portu natychmiast	tcsetattr(fd, TCSANOW, &options)
TCSETSF	Wyczyść (flush) bufory wejściowy i wyjściowy po czym ustaw nowe ustawienia portu	tcsetattr(fd, TCSAFLUSH, &options)
TCSETSW	Poczekaj na opróżnienie buforów wejściowego i wyjściowego po czym ustaw nowe ustawienia portu	tcsetattr(fd, TCSADRAIN, &options)
TCSBRK	Wystaw sygnał break na zadany czas	tcsendbreak, tcdrain
TCXONC	Konfiguracja programowej kontroli przepływu	tcflow
TCFLSH	Przepłukanie bufora wejściowego i/lub wyjściowego	tcflush
TIOCMGET	Pobranie stanu linii sterujących	-
TIOCMSET	Zmiana stanu linii sterujących	-
FIONREAD	Pobranie liczby bajtów znajdujących się w buforze wejściowym (jeszcze nie odczytanych)	-

i odpowiada za przechowywanie informacji konfiguracyjnych związanych z urządzeniami terminalowymi. Zawiera ona kilkadziesiąt flag, które można modyfikować za pomocą specjalnych masek bitowych i w ten sposób zmieniać ustawienia portu szeregowego. Możemy decydować czy i jak dane wejściowe i wyjściowe mają być przetwarzane, włączać i wyłączać odbiornik i tak dalej. Struktura *termios* zawiera 7 pól wymienionych w **tab. 1**.

Ogólny schemat modyfikacji *termios* przedstawiony jest na **list. 2**. Modyfikacja polega na skopiowaniu zawartości tej struktury do pewnej zmiennej, stosownej modyfikacji tej zmiennej, a następnie na skopiowaniu jej zawartości z powrotem do struktury *termios*. W pierwszej kolejności deklarujemy zmienną *options* typu *struct termios*. Następnie pobieramy aktualne ustawienia portu za pomocą funkcji *tcgetattr*, która oprócz adresu zmiennej *options* przyjmuje deskryptor pliku portu zwrócony wcześniej przez funkcję *open*. Teraz zmienna *options* zawiera kopię aktualnej struktury *termios* systemu. Możemy dowolnie modyfikować tę kopię (jej pola) poprzez bitowe operacje OR i AND z odpowiednimi stałymi charakterystycznymi dla poszczególnych pól struktury *termios* (omówię je za chwilę). Wykonanie operacji OR ze stałą oznacza włączenie odpowiadającej jej opcji, zaś wykonanie operacji AND z negacją tej stałej – wyłączenie. **Należy przy tym pamiętać, że:**

Po pierwsze – nie wystarczy **nie włączyć** jakiejś opcji przez OR, aby nie była ona włączona! Należy ją w tym celu **wyłączyć** operacją AND! Wynika to stąd, że struktura *termios* dla danego portu jest wspólna dla wszystkich aplikacji. Przykładowo, jeśli nie chcemy aby nasz program korzystał z mapowania znaku CR (0x0D) na znak NL (0x0A) i odwrotnie, to musimy koniecznie wyłączyć te funkcje pisząc

options.c_iflag &=~(ICRNL | INLCR);

List. 4. Odczyt i modyfikacja stanu linii sterujących portu

```
//*****************************************
//
//  Getting and setting the control signals
//
//*****************************************
#include <unistd.h>
#include <termios.h>

//File descriprot
int fd;

//Variable for holding the control signals
int status;

//Getting the control signals
ioctl(fd, TIOCMGET, &status);


//  Setting the control signals
//  - logic 0 to DTR
ioctl(fd, TIOCMGET, &status);

status &= ~TIOCM_DTR;

ioctl(fd, TIOCMSET, &status);
```

Jeśli tego nie zrobimy, to pomimo, że nie włączyliśmy tych opcji może się zdarzyć, że nasza aplikacja zacznie wykazywać takie cechy! Wystarczy, że wcześniej używana była aplikacja, która je włączyła. Znaczenie stałych użytych w powyższej instrukcji przedstawię za chwilę.

Po drugie – nie wolno zmieniać pól struktury *termios* poprzez bezpośrednie przypisanie pewnej wartości, na przykład takiej

options.c_iflag = 0;

Postępowanie takie może być bardzo szkodliwe, gdyż inne aplikacje mogą korzystać z pewnych opcji, których my nie powinniśmy zmieniać.

Gdy już dokonamy wszystkich niezbędnych zmian w zmiennej *options* należy wpisać je do właściwej struktury *termios* w systemie. Służy do tego funkcja *tcsetattr*, która oprócz deskryptora pliku i adresu zmodyfikowanej kopii *termios* przyjmuje dodatkowy argument określający sposób jej działania. Możliwe wartości tego argumentu przedstawiono w **tab. 2**.

Zauważmy, że ustawienie prędkości transmisji (co ciekawe – osobno dla nadajnika i odbiornika) nie odbywa się poprzez odpowiednie operacje logiczne OR, lecz za pomocą funkcji *cfsetispeed* i *cfsetospeed*. Robimy tak dlatego, że starsze wersje systemów umieszczały informację o prędkości w polu *c_cflag* (patrz następny punkt), zaś nowe umieszczają ją w polach *c_ispeed* i *c_ospeed* struktury *termios*. Użycie *cfsetispeed* i *cfsetospeed* uwalnia nas od zastanawiania się nad tym, gdzie w istocie informacje te się znajdują.

Konfiguracja: pole c_cflag – opcje sterowania

Zawartość pola *c_cflag* kontroluje liczbę bitów danych w ramce RS232, steruje kontrolą parzystości, określa liczbę bitów stopu oraz po-

Tab. 9. Stałe określające stan linii sterujących portu

Stała	Opis
TIOCM_DTR	Sterowanie stanem DTR
TIOCM_RTS	Sterowanie stanem RTS
TIOCM_CTS	Sterowanie stanem CTS
TIOCM_CAR	Sterowanie stanem DCD
TIOCM_CD	Synonim dla TIOCM_CAR
TIOCM_DSR	Sterowanie stanem DSR

zwala włączyć sprzętową kontrolę przepływu (*hardware flow control*). Stałe pozwalające włączać i wyłączać poszczególne opcje przedstawiono w **tab. 3**. Wartości określające *baudrate* interesują nas tylko w kontekście użycia funkcji *cfsetispeed* i *cfsetospeed* – jak napisałem wyżej, nie należy ustawiać prędkości transmisji korzystając z pola *c_cflag*. Zupełnie nieprzydatne są z naszego punktu widzenia stałe EXTA i EXTB. Natomiast istotną rolę pełnią pola CSIZE i CS5...CS8. Pozwalają one na wybór liczby bitów danych. W ogromnej większości przypadków ustawimy 8 bitów danych przy użyciu następujących instrukcji:

```
options.c_cflag &= ~CSIZE;
options.c_cflag |= CS8;
```

Stała CSTOPB ustawia liczbę bitów stopu – włączona daje 2 bity stopu, wyłączona – 1 bit. CREAD włącza odbiornik – aby móc odbierać dane nie wystarczy otworzyć port z opcją O_RDWR (funkcja *open*), musi być dodatkowo ustawiona flaga CREAD. Flaga PARENB włącza kontrolę parzystości. Jeśli ją włączymy, to za pomocą PARODD możemy określić rodzaj parzystości – *odd parity* lub *even parity*. Podsumowując, kombinacja stałych PARENB, PARODD, CSIZE i CS5... CS8 określa rodzaj ramki. Popularną ramkę 8n1 otrzymujemy za pomocą następujących instrukcji:

```
options.c_cflag &= ~PARENB;
options.c_cflag &= ~CSTOPB;
options.c_cflag &= ~CSIZE;
options.c_cflag |= CS8;
```

Powinniśmy ustawić flagę CLOCAL. Dzięki temu proces związany z naszą aplikacją będzie miał wyłączność na korzystanie z portu. Flaga CRTSCTS (nowa CNEW_RTSCTS) włącza sprzętową kontrolę przepływu. W typowych zastosowaniach nie jest nam ona potrzebna, dlatego flagę tę wyzerujemy.

Konfiguracja: pole c_lflag – opcje lokalne

W **tab. 4** przedstawiłem stałe dla pola *c_lflag*. Pole to pozwala określić, czy dane wejściowe będą przetwarzane przed umieszczeniem ich w buforze wejściowym. Generalnie pozwala ono na ustawienie wejścia zorientowanego liniowo (*canonical input*) lub wejścia bez przetwarzania danych (*raw input*). Wejście typu *canonical input* uzyskamy następująco:

```
options.c_lflag |= (ICANON |
    ECHO | ECHOE);
```

zaś wejście bez przetwarzania (*raw input*) następująco:

```
options.c_lflag &= ~(ICANON |
    ECHO | ECHOE | ISIG);
```

Opcje pozwalające uzyskać echo są ciekawe i w wielu przypadkach mogą być użyteczne. Jednak nie przydadzą się zbytnio w typowych zastosowaniach łącza szeregowego, dlatego wyłączymy je wybierając wejście typu *raw input*. Należy pamiętać, że włączenie echa przy współpracy z urządzeniami, które odpowiadają na zadawane im pytania (np. w architekturze Master–Slave) może doprowadzić do nieskończonej pętli, która zablokuje port.

Konfiguracja: pole c_iflag – opcje wejścia

W **tab. 5** zamieściłem stałe dla pola *c_iflag*. Pole to określa zachowanie się portu przy wykryciu błędów parzystości oraz odpowiada za włączenie programowej kontroli przepływu (*software flow control*). Flagi INPCK i PARMRK określają jak zaznaczane są znaki obciążone błędem parzystości – mogą być one na przykład zerowane (zamiana na znak \0, czyli po prostu na liczbę 0). Jeśli włączona jest opcja PARENB w polu *c_cflag*, to należy też zdefiniować, co sterownik ma robić z błędnymi bajtami.

Pole *c_iflag* umożliwia także włączenie mapowania pewnych bajtów na inne bajty przed umieszczeniem ich w buforze wejściowym oraz ignorowanie pewnych wartości. Służą do tego stałe INLCR, IGNCR, ICRNL, IUCLC. My chcemy, aby w buforze wejściowym znajdowały się dokładnie te znaki, które wysłało urządzenie dołączone do PC – w związku z tym wyłączymy mapowanie CR na NL (i odwrotnie) oraz ignorowanie znaku CR, a także mapowanie wielkich liter na małe. Oczywiście użyjemy do tego następującej instrukcji:

```
options.c_iflag &= ~(ICRNL | INLCR | IGNCR | IUCLC);
```

Wyłączymy także programową kontrolę przepływu pisząc:

```
options.c_iflag &= ~(IXON | IXOFF
    | IXANY);
```

Konfiguracja: pole c_oflag – opcje wyjścia

Stałe dla opcji wyjścia przedstawia **tab. 6**. Ich znaczenie jest w pewnym sensie analogiczne do opcji pola *c_iflag* – określają sposób przetwarzania bajtów przed umieszczeniem ich w buforze wyjściowym i wysłaniem do urządzenia, z którym komunikuje się komputer. Jak nietrudno się domyślić – w typowych zastosowaniach nie skorzystamy z tych ułatwień. W przypadku pola *c_oflag* mamy ułatwione zadanie. Otóż wszelkie manipulacje na wysyłanych danych następują tylko wtedy, gdy ustawiona jest flaga OPOST. W takiej sytuacji pozostałe flagi pola *c_oflag* określają sposób ich przetwarzania. My wyłączymy OPOST i tym samym uzyskamy wyjście bez przetwarzania – *raw output*. Posłuży do tego instrukcja:

```
options.c_oflag &= ~OPOST;
```

Konfiguracja: tablica znaków specjalnych c_cc

Tab. 7 zawiera tablicę znaków specjalnych, do której wskaźnik stanowi pole *c_cc* struktury *termios*. Zawiera ona kody znaków specjalnych, w szczególności znaków służących realizacji programowej kontroli przepływu, czyli VSTART (XON) i VSTOP (XOFF). Pola VMIN i VTIME pozwalają na wykorzystanie wbudowanej w sterownik funkcji kontroli przeterminowania operacji odczytu danych. Ich zawartość jest ignorowana gdy wybrano wejście typu *canonical input* lub gdy skonfigurowano port z włączoną opcją NDELAY podczas otwarcia lub przez wywołanie odpowiedniej funkcji *fcntl* (opis w dalszej części). Elementy VMIN i VTIME pozwalają poinformować sterownik, że oczekujemy bloków danych o ściśle określonej długości. Znaczenie tych pól jest następujące: VTIME określa czas oczekiwania na odebranie pierwszego znaku z bloku danych o długości VMIN. Jeśli znak ten nadejdzie w czasie VTIME (od wywołania funkcji *read*), to operacja odczytywania będzie czekać, aż liczba odebranych znaków osiągnie VMIN i dopiero wtedy się zakończy zwracając liczbę odczytanych bajtów (patrz opis funkcji *read*). Jeśli zaś w czasie VTIME nie nadejdzie ani

jeden znak, to funkcja *read* zwróci 0. Dzięki temu informujemy sterownik, że oczekujemy bloków o długości VMIN i wszelkie wywołania funkcji *read* zwrócą albo oczekiwaną liczbę bajtów danych, albo 0.

Jeśli wartość VMIN ustawimy na 0, to VTIME określać będzie maksymalny czas oczekiwania na każdy nadchodzący bajt. Oznacza to, że jeśli wywołamy funkcję *read* w chwili, gdy w buforze wejściowym nie ma danych, to jeśli nie nadejdą one w czasie VTIME – funkcja zwróci 0. Jeśli nadejdą, to zwróci 1 natychmiast po otrzymaniu pierwszego bajta. Jeśli zaś w chwili wywołania dane w buforze są (choćby 1 bajt), to zostaną one natychmiast odczytane.

Czytanie z portu – funkcja *read*

Odczyt danych z bufora wejściowego odbywa się za pomocą funkcji *read*. Jej prototyp jest następujący:

```
int read(int fd, void *buffer, int
    NumberOfBytesToRead);
```

Funkcja ta przyjmuje następujące argumenty:
- fd – deskryptor pliku portu;
- buffer – wskaźnik na początek obszaru pamięci, gdzie mają być umieszczone dane odczytane z bufora wejściowego portu;
- NumberOfBytesToRead – żądana liczba bajtów do odczytu.

Funkcja *read* zwraca liczbę faktycznie odczytanych bajtów.

Ciekawe jest jej działanie, gdy w buforze wejściowym nie ma żadnych danych i ustawiono wejście bez przetwarzania (*raw input*). Domyślne zachowanie jest takie, że funkcja ta blokuje aktualny wątek i czeka na nadejście danych lub na opisane wyżej przeterminowanie oczekiwania. Można wyłączyć takie zachowanie i sprawić, że funkcja *read* będzie w takiej sytuacji natychmiast zwracać 0. Dokonuje się tego w sposób następujący:

```
fcntl(fd, F_SETFL, FNDELAY);
```

Przywrócenie domyślnego, blokującego działania uzyskamy następująco:

```
fcntl(fd, F_SETFL, 0);
```

Pisząc prostą aplikację jednowątkową skorzystamy z nieblokującego działania funkcji *read*, zaś zanim ją wywołamy sprawdzimy, ile nieodczytanych bajtów oczekuje w kolejce wejściowej.

Wysyłanie danych – funkcja *write*

W przeciwieństwie do czytania, pisanie do portu pozbawione jest „sztuczek" (jest z resztą ogólną prawidłowością, że łatwiej implementuje się nadajniki niż odbiorniki). Zajmuje się tym funkcja *write* określona następująco:

```
int write(int fd, void *buffer, int
    NumberOfBytesToWrite);
```

gdzie:
- fd – deskryptor pliku portu;
- buffer – wskaźnik na początek obszaru pamięci, gdzie znajdują się dane przeznaczone do wysłania;
- NumberOfBytesToWrite – liczba bajtów do wysłania.

Funkcja zwraca liczbę faktycznie wysłanych bajtów.

Pobranie liczby nieodczytanych bajtów znajdujących się w buforze wejściowym – wywołanie funkcji *ioctl*

Jest to niezwykle cenna właściwość sterownika portu – umożliwia poznanie, ile bajtów zostało odebranych przez UART komputera bez konieczności ich odczytywania. Pozwala to na przykład na dokonanie odczytu dopiero wtedy, gdy nadeszła taka ilość informacji, jaka nas interesuje. W Linuksie możemy to zrobić za pomocą funkcji *ioctl*, na przykład tak, jak to przedstawiłem na **list. 3**.

Funkcja *ioctl*

W poprzednim punkcie użyta została funkcja systemowa *ioctl*. Funkcja ta pozwala na wykonanie przeróżnych operacji na plikach, a więc w szczególności na porcie szeregowym. Posiada ona następujący prototyp:

```
int ioctl(int fd, int cmd, ...)
```

gdzie:
- fd – deskryptor pliku (np. pliku portu);
- cmd – stała określająca operację, jaką chcemy wykonać na pliku wskazywanym przez fd;

Trzy kropki (...) występujące w prototypie funkcji *ioctl* mogą sugerować, że przyjmuje ona zmienną liczbę argumentów. Tak jednak nie jest – funkcja przyjmuje co najwyżej 3 argumenty, zaś użycie kropek sprawia, że kompilatory C/C++ nie kontrolują typu trzeciego argumentu. Jego konkretne znaczenie zależy od wartości argumentu *cmd*.

Tab. 8 przedstawia niektóre wartości argumentu *cmd* przydatne przy pracy z interfejsem szeregowym. Pokazane są tam także funkcje standardu POSIX, których działanie odpowiada działaniu funkcji *ioctl* wywołanej z określonym argumentem. Jak widać są wśród nich znane nam już funkcje *tcgetattr* i *tcsetattr* (w systemach unixowych używają one *ioctl*).

Odczyt i modyfikacja stanu linii sterujących portu

Przydać się mogą wywołania *ioctl* z argumentem *cmd* równym TIOCMGET i TIOCMSET – pozwalają one odczytywać i zmieniać stan linii sterujących portu szeregowego. **Tab. 9** przedstawia stałe określające różne stany tych linii, zaś na **list. 4** zamieszczony jest przykład ich użycia. Jak widać zmiana stanu linii sterujących polega na pobraniu kopii ich stanu do tymczasowej zmiennej, modyfikacji tej zmiennej i w końcu na skopiowaniu jej wartości z powrotem do systemu.

Zamykanie portu

Port zamykamy za pomocą funkcji *close* określonej następująco:

```
close(int fd)
```

Podsumowanie

Mamy już wszystkie niezbędne informacje pozwalające wykorzystać port szeregowy w systemie Linux. Pozostało zebrać je do przysłowiowej kupy i zastosować.

W drugiej części artykułu wykorzystamy zdobyte informacje w praktyce. Pokażę przykład prostej aplikacji konsolowej komunikującej się z dołączonym do PC urządzeniem za pośrednictwem RS232C, zaś w następnych częściach kursu weźmiemy na warsztat aplikacje GUI. W części drugiej znajdzie się także opis prostego zestawu laboratoryjnego zbudowanego na mikrokontrolerze ATmega8, który w dalszych częściach cyklu posłuży do testowania prezentowanych przykładów.

Arkadiusz Antoniak, EP
arkadiusz.antoniak@ep.com.pl