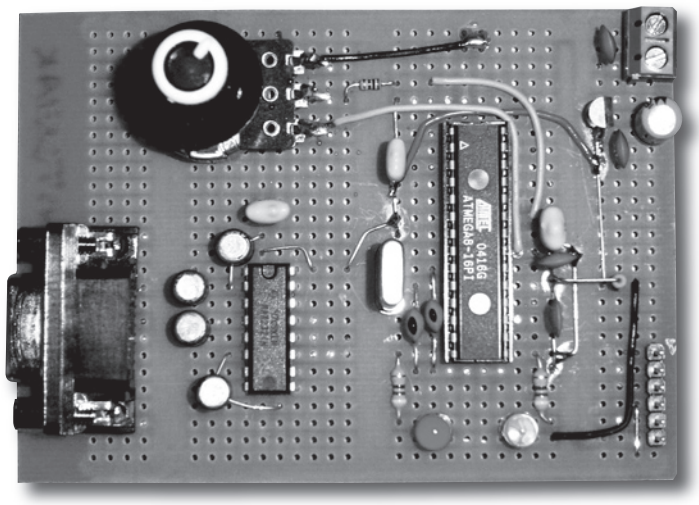
Programowanie portu szeregowego w systemach operacyjnych Linux i Windows, część 1

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.

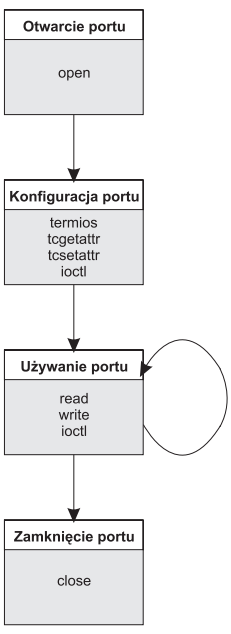


Popularność Linuksa w ostatnich latach wyraźnie wzrosła i wiele wskazuje na to, że zjawisko to będzie się z roku na rok nasilać. Choć dla nas – elektroników – podstawową platformą pozostaje Windows, to powstaje coraz więcej programów narzędziowych, kompilatorów dla mikrokontrolerów i innych programów, 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)
```

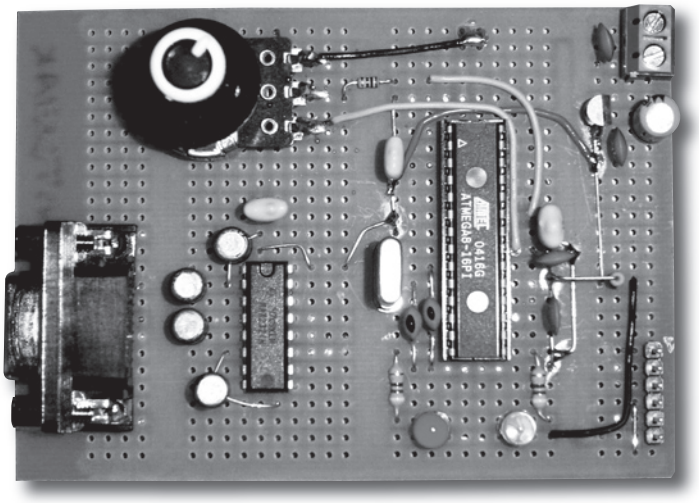
Podsumowanie

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

Programowanie portu szeregowego w systemach operacyjnych Linux i Windows, część 2



Umiejętność programowej obsługi interfejsu RS232 od strony komputera PC jest dziś istotnym elementem elektronicznego rzemiosła. W niniejszym kursie 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 poparto szczegółowo opisanymi praktycznymi przykładami.

W poprzedniej części kursu omówiłem funkcje służące do realizacji operacji na porcie szeregowym w Linuksie. Przyszedł czas na wykorzystanie zdobytej wiedzy w praktyce. W części drugiej przedstawię budowę platformy sprzętowej, która pozwoli nam testować tworzone oprogramowanie.

Zestaw testowy

Schemat zestawu testowego, który posłuży nam do testowania wszystkich przykładów w dalszych częściach kursu, znajduje się na **rys. 2**. Jego sercem jest dobrze wszystkim znany mikrokontroler ATmega8 firmy Atmel. Wybór tego właśnie mikrokontrolera podyktowany był faktem, że jest to jeden z najmniejszych mikrokontrolerów serii ATmega, który ma „na pokładzie" przetwornik analogowo–cyfrowy. Przetwornik ten wykorzystamy w ostatniej części cyklu do realizacji woltomierza sterowanego przez port szeregowy. Sam wybór rodziny AVR Atmela był oczywisty – są to wszak najpopularniejsze (jeszcze!) w Polsce mikrokontrolery i większość Czytelników zna je co najmniej dobrze. Dzięki temu będą oni mogli skupić swoją uwagę na zagadnieniu programowania RS232C na PC, zaś szczegóły oprogramowania części sprzętowej będą dla nich od początku do końca jasne.

Połączenie procesora U1 z komputerem PC zrealizowane jest typowo, to jest za pośrednictwem układu MAX232. Model łączony jest z komputerem za pośrednictwem kabla „prostego" (bez przeplotu), co wymusiło zastosowanie złącza DB9F (żeńskiego). Czytelnicy, którzy zechcą użyć typowego przewodu z przeplotem muszą wykorzystać złącze DB9M oraz zamienić w nim miejscami końcówki 2 i 3. Rezonator kwarcowy X1 ma „okrągłą" wartość 1,8432 MHz, co pozwala uzyskać większość typowych prędkości transmisji z zerowym (obliczanym) błędem baudrate. We wszystkich przykładach w tym kursie wykorzystana będzie prędkość 19200 bodów i ramka w formacie 8N1 (8 bitów danych, brak kontroli parzystości, 1 bit STOP). Dodatkową zaletą dosyć niskiej częstotliwości pracy mikrokontrolera jest redukcja pobieranego przez zestaw prądu.

Diody LED D1 i D2 stanowią najprostszy interfejs wyjściowy – rola przez nie pełniona będzie zależna od funkcjonalności testowanego przykładu. Elementy U3, C7... C9 tworzą typowy zasilacz stabilizowany zasilający część cyfrową napięciem 5 V. Para R1C3 wytwarza napięcie zasilające część analogową zestawu. Masy: analogowa i cyfrowa są rozdzielone i łączą się elektrycznie niedaleko punktu dołączenia napięcia zasilającego. Nierozdzielenie mas analogowej i cyfrowej mogłoby doprowadzić do podawania na wejście przetwornika A/C impulsów szpilkowych i – w konsekwencji – do jego błędnej pracy. Impulsy te pochodziłyby ze spadków napięcia na ścieżkach masy powodowanych dużymi prądami impulsowymi występującymi w części cyfrowej.

We wspomnianym przykładzie woltomierza cyfrowego przetwornik A/C będzie pracował z wewnętrznym napięciem odniesienia równym (typowo) 2,56 V – napięcie to występuje na końcówce AREF. Jest ono filtrowane przez kondensator C2 i stanowi napięcie wejściowe dla potencjometru PR1. Napięcie z suwaka PR1 jest podawane poprzez filtr R2C1 na wejście ADC0 przetwornika. Napięcie to będzie mierzone przez woltomierz, który zaprojektujemy w dalszej części kursu.

Zestaw testowy można zmontować w dowolny sposób pamiętając jedynie o rozdzieleniu mas i ogólnych zasadach montażu układów elektronicznych. Zamiast opisanego układu można oczywiście użyć dowolnego zestawu laboratoryjnego dostępnego w handlu, byleby tylko miał on podobne możliwości funkcjonalne jak opisany powyżej (przetwornik A/C i interfejs RS232C). Nie musi to być

nawet zestaw z mikrokontrolerem AVR. Oprogramowanie jest napisane w języku C i może być przeniesione na dowolny mikrokontroler, do którego istnieje kompilator tego języka. Oczywiście w takim przypadku trzeba będzie zmienić te fragmenty programu, które są specyficzne dla danego układu (np. konfiguracja modułu USART).

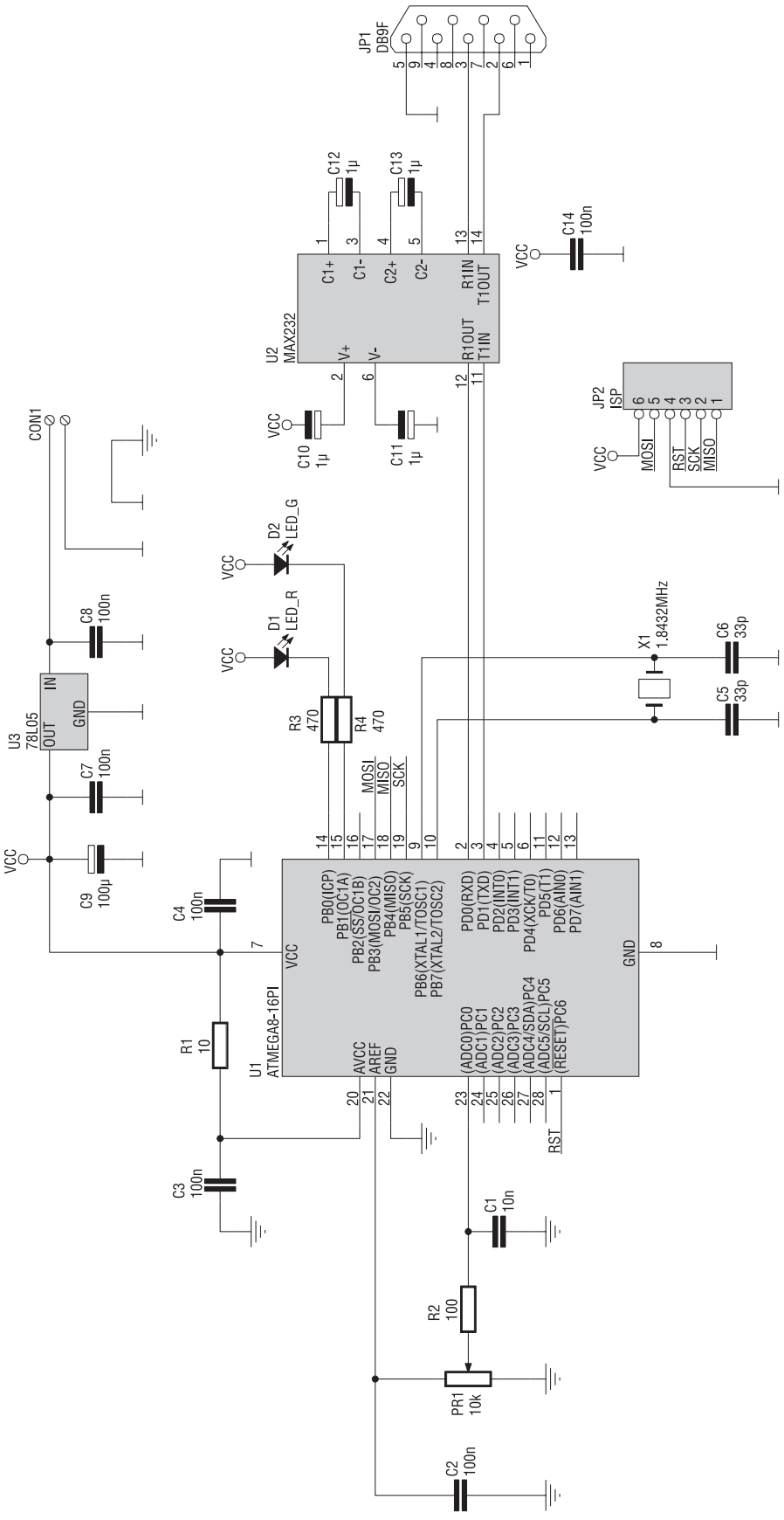
Oprogramowanie mikrokontrolera

Oprogramowanie mikrokontrolera U1 ATmega8 napisałem w języku C używając popularnego kompilatora GCC w wersji 2002–06–25 zintegrowanego z pakietem AVRStudio3.53. Kod źródłowy jest na tyle krótki, prosty i pozbawiony „wodotrysków", że można go z powodzeniem skompilować za pomocą dowolnego kompilatora C dla mikrokontrolerów AVR – oczywiście po dokonaniu stosownych (niewielkich) poprawek.

List. 5. Program testowy Example1.c działający na mikrokontrolerze

```
//*******************************
// MAIN
//*******************************
int main(void)
{
  unsigned char data,cou;
  DDRB=0xFF;
  DDRC=0x00;
  DDRD=0xFE;
  PORTC=0x00;
  PORTD=0xFE;

  Hello();

  Init_UART();

  while(1)
  {
     //Waiting for 1 byte
     UCSRA&=~(1<<RXC);
     while((UCSRA&(1<<RXC))==0);
     data=UDR;

     if(data==0)
     {
        RED_OFF;
        GREEN_OFF;
        SendString(PSTR(„ZERO"));
        SendByte(13); SendByte(10);
     }
     else if(data==1)
     {
        RED_ON;
        GREEN_OFF;
        SendString(PSTR(„ONE"));
        SendByte(13); SendByte(10);
     }
     else if(data==2)
     {
        RED_OFF;
        GREEN_ON;
        SendString(PSTR(„TWO"));
        SendByte(13); SendByte(10);
     }
     else if(data==3)
     {
        RED_ON;
        GREEN_ON;
        SendString(PSTR(„THREE"));
        SendByte(13); SendByte(10);
     }
     else if(data==255)
     {
        cou=0;
        while(1)
        {
           SendByte(cou);
           if(++cou==0) break;
        }
     }
     else
     {
     SendString(PSTR(„FOUR OR MORE"));
     SendByte(13); SendByte(10);
     }
  }
}
```

W **tab. 10** zamieściłem konfigurację *fusebit*–ów niezbędną do poprawnej pracy procesora ATmega8 w opisywanym zestawie. Podczas programowania mikrokontrolera należy zwrócić baczną uwagę na poprawność tych ustawień, gdyż ich zmiana może uniemożliwić pracę syste-



Rys. 2. Schemat zestawu testowego

mu! Podczas kursu będziemy wykorzystywać dwa programy działające na mikrokontrolerze U1. Pierwszym z nich (opisanym niżej) jest prosty system odpowiadający na pytania. Program ten będzie wykorzystywany w dalszych częściach niniejszego kursu. Drugą aplikacją jest oprogramowanie cyfrowego woltomierza, którą weźmiemy na warsztat w ostatniej części kursu.

Wszystkie przykłady oprogramowania na PC (poza woltomierzem) będą współpracować z prostym programem *Example1.c* działającym na mikrokontrolerze. Główna część tego programu (funkcja *main*) została przedstawiona na **list. 5** (pominąłem inicjalizację modułu USART i inne funkcje pomocnicze). Idea tej aplikacji polega na stworzeniu prostego systemu zdolnego odpowiadać na zadawane mu proste pytania. Taki system pozwoli w łatwy sposób testować różne aplikacje działające na komputerze PC.

Sposób działania programu z list. 5 jest następujący:

1. Mikrokontroler czeka na 1–bajtowe zapytanie, które wysyła komputer PC;
2. Po otrzymaniu jednego bajta mikrokontroler odpowiednio do jego wartości zapala diody D1 i D2 oraz wysyła via RS232 stosowną odpowiedź. Odbywa się to według następującej zasady:
   - Jeśli wartość bajta zapytania mieści się w granicach 0...3, to stan diod odzwierciedla tę liczbę (D2 – bit nr 1, D1 – bit nr 0), zaś wysyłana do PC–ta odpowiedź stanowi nazwę tej wartości zapisaną wielkimi literami w języku angielskim zakończoną sekwencją bajtów 0x0D i 0x0A. Na przykład otrzymanie bajta o wartości 1 skutkuje wygaszeniem diody D2, zapaleniem D1 i wysłaniem odpowiedzi „ONE”+0x0D+0x0A. Dla bajta o wartości 3 zostaną zapalone obie diody, a odpowiedzią będzie „THREE”+0x0D+0x0A.
   - Jeśli wartość bajta zapytania mieści się w granicach 4...254, to stan diod nie zmienia się, zaś wysyłana odpowiedź ma postać „FOUR OR MORE”+0x0D+0x0A.
   - Jeśli wartość bajta zapytania wynosi 255, to stan diod nie zmienia się, zaś w odpowiedzi wysyłana jest sekwencja 256 bajtów o wartościach 0...255.

Wywołanie funkcji powitania *Hello* powoduje dwukrotne mignięcie obiema diodami D1 i D2 tuż po włączeniu zasilania zestawu lub zerowaniu mikrokontrolera. Rzecz banalna, ale podczas pierwszego uruchomienia systemu pozwala zorientować się, czy zestaw daje jakiekolwiek „znaki życia”.

WYKAZ ELEMENTÓW

Rezystory

R1: 10 V
R2: 100 Ω
R3, R4: 470 Ω

Kondensatory

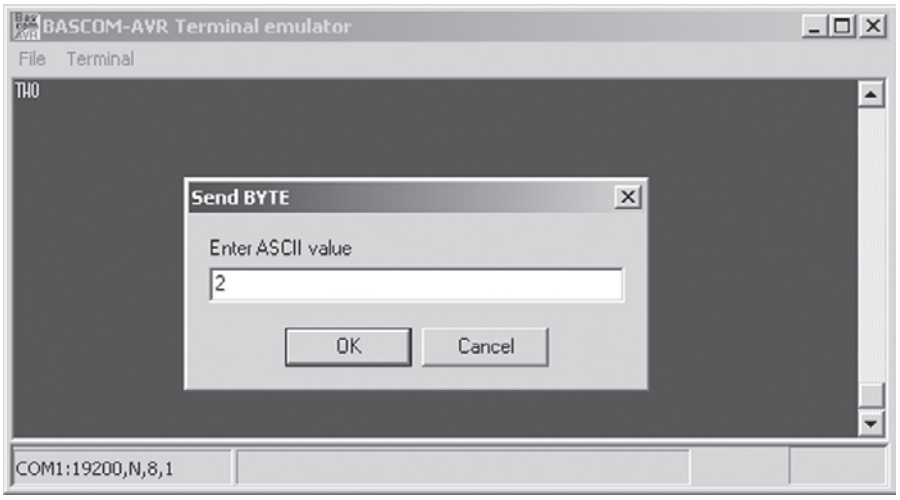
C1: 10nF
C2...C4, C7, C8, C14: 100 nF
C5, C6: 33 pF
C9: 100 μF/16 V
C10...C13: 1...10 μF/16 V

Półprzewodniki

U1: ATmega8
U2: MAX232
U3: 78L05
D1: LED czerwona
D2: LED zielona

Inne

Kwarc: 1,8432 MHz
JP1: złącze DB9F (kabel prosty) lub DB9M (kabel z przeplotem)
JP2: goldpiny (złącze ISP)
PR1: potencjometr 10 kΩ A z pokrętłem



Rys. 3. Okno terminala podczas testowania zestawu

Testowanie zestawu

Poprawnie zmontowany zestaw działa od pierwszego włączenia i nie wymaga żadnych czynności uruchomieniowych. Warto jednak przetestować go przed podjęciem prób współpracy z oprogramowaniem pracującym na PC. Do testów można użyć dowolnego terminala RS232C. Należy w tym celu ustawić odpowiednie parametry transmisji (19200, 8N1), a następnie wysyłać 1–bajtowe zapytania i sprawdzać czy w odpowiedzi wysyłane są poprawne ciągi znaków. Na **rys. 3** przedstawiłem przykładowy test zestawu za pomocą terminala wchodzącego w skład pakietu BascomAVR Demo, pracującego w systemie Windows.

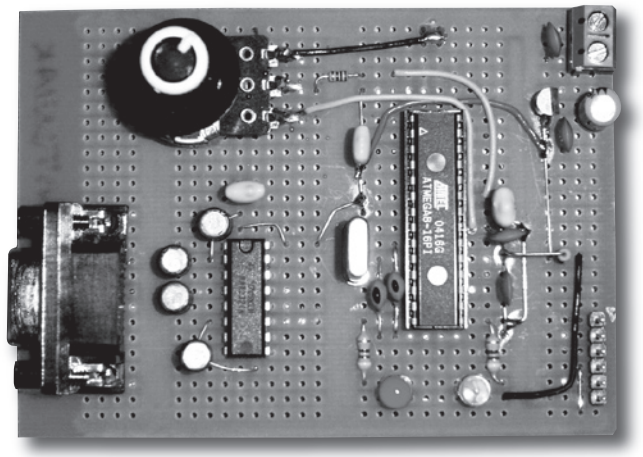
Arkadiusz Antoniak, EP
arkadiusz.antoniak@ep.com.pl
www.antoniak.ep.com.pl

Tab. 10. Ustawienia bezpieczników (fusebit) mikrokontrolera

	Fusebit	Wartość
Fusebit LOW	BODLEVEL	1: BODLEVEL 2,7V
	BODEN	0: BODEN enabled
	SUT1..0	01: fast rising power
	CKSEL3..0	1101: Crystal oscillator
Fusebit HIGH	S8515C	1: MEGA8515 mode
	WDTON	1: Watchdog timer enabled by software
	SPIEN	0: Serial programming enabled
	CKOPT	1: Oscillator option
	EESAVE	1: Erase EEPROM when chip erase
	BOOTSZ1..0	00: 1024 words boot size
	BOOTRST	1: Reset vector is $0000

Programowanie portu szeregowego w systemach operacyjnych Linux i Windows, część 3

Umiejętność programowej obsługi interfejsu RS232 od strony komputera PC jest dziś istotnym elementem elektronicznego rzemiosła. W niniejszym kursie 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.



Aplikacja konsolowa używająca portu szeregowego

Aplikacja konsolowa została napisana w języku C i skompilowana za pomocą kompilatora GCC, który spotkamy w każdej dystrybucji Linuksa. Składa się ona z trzech plików:

- *main.c* – główny plik aplikacji odpowiedzialny za jej funkcjonalność (**list. 6**);
- *LinuxRS232.h* – plik nagłówkowy zawierający deklaracje funkcji obsługi interfejsu RS232 (**list. 7)**;
- *LinuxRS232.c* – plik źródłowy zawierający definicje (ciała) funkcji obsługi interfejsu RS232 (**list. 8...11**).

Na list. 6 zamieściłem zawartość pliku *main.c*. Działanie programu polega na pobraniu jednej cyfry (jeden znak+Enter) z klawiatury i wysłaniu przez port szeregowy /dev/ttyS0 (COM1) bajta o wartości równej podanej cyfrze (0...9). Jeśli wpiszemy znak inny niż 0...9 to przez port zostanie wysłany bajt o wartości równej kodowi ASCII tego znaku pomniejszonemu o 0x30. Następnie program zasypia na czas ok. 1 sekundy (*break time*), po czym sprawdza czy w buforze wejściowym portu znajdują się jakiekolwiek dane (odpowiedź zestawu). Jeśli tak jest są one odczytywane i w konsoli pojawia się zarówno ich interpretacja ASCII jak i wartości kolejnych bajtów zapisane dziesiętnie. Pokazuje to **rys. 4** zawierający zrzut z ekranu zrobiony podczas pracy aplikacji, gdy użytkownik jako daną do wysłania wpisał liczbę 3. Zmienna *InQue* zawiera odczytaną za pomocą funkcji *CheckCommInput* liczbę bajtów oczekujących w buforze wejściowym, zaś zmienna *BytesRead* zawiera liczbę bajtów faktycznie wyjętych z tego bufora za pomocą funkcji *Read*.

List. 8 przedstawia implementację funkcji *Open* i *Close*. Funkcja *Open* otwiera port, po czym ustawia żądaną przez użytkownika prędkość transmisji oraz wyłącza co tylko można z funkcjonalności obróbki danych wejściowych i wyjściowych. Innymi słowy, wybieramy wejście *raw input* i wyjście *raw output* wyłączając wszelkie funkcje mapowania jednych znaków na drugie, sprzętową i programową kontrolę przepływu, itp..

List. 9 zawiera ciało funkcji *Write* służącej do wysłania do portu *NumberOfBytesToWrite* bajtów, począwszy od miejsca w pamięci określonego wskaźnikiem *Buffer*. Funkcja ta zwraca liczbę faktycznie wysłanych bajtów. W przypadku błędu zwraca 0.

List. 10 zawiera ciało funkcji *Read*, za pomocą której możemy wybrać z bufora wejściowego portu *NumberOfBytesToRead* bajtów. Specyfikujemy przy tym pewne ograniczenie określone stałą *cbInQueue* zdefiniowaną w pliku *LinuxRS232.h*. Ograniczenie to zapobiega próbie odczytu wielu bajtów z portu. Wartość *cbInQueue* powinna być dobrana tak, aby podczas normalnej pracy aplikacji w żadnym momencie nie zachodziła potrzeba odczytu z portu liczby bajtów większej od *cbInQueue*. Funkcja *Read* zwraca 0 w przypadku błędu i 1, gdy odczyt zakończył się sukcesem. Za pośrednictwem wskaźnika *lpNumberOfBytesRead* zwracana jest liczba faktycznie odczytanych bajtów.

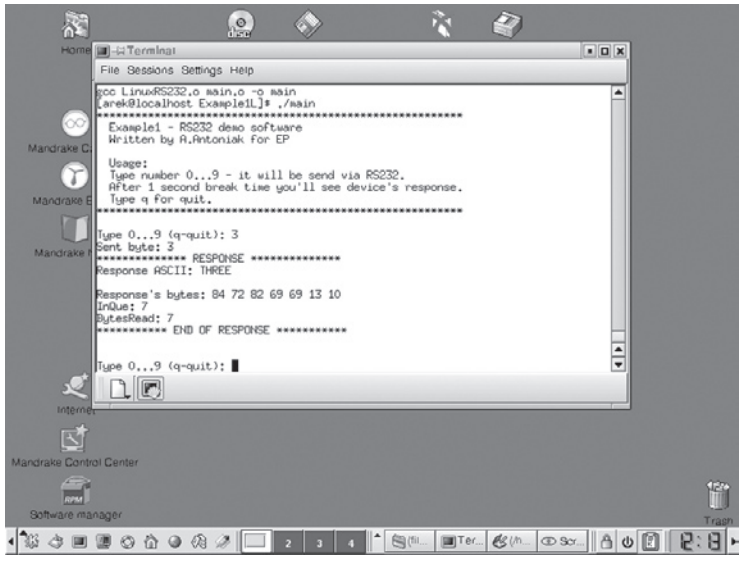
List. 7. Plik nagłówkowy LinuxRS232.h

```
//------------------------------------------------------------------------------
//  File: LinuxRS232.h
//  Description: Linux RS232 header file
//  Compiler: gcc
//  Author: Arkadiusz Antoniak, 2005
//------------------------------------------------------------------------------
#ifndef LinuxRS232H
#define LinuxRS232H
#define cbInQueue 1024
#define cbOutQueue 16
int Open(char *Port, unsigned long Baud);
void Close(void);
unsigned long Write(const void* Buffer, unsigned long NumberOfBytesToWrite);
int Read(void * Buffer, unsigned long * lpNumberOfBytesRead, unsigned long NumberO-
fBytesToRead);
void CheckCommInput(unsigned long * lpInQue);
//------------------------------------------------------------------------------
#endif
```

Uwaga!
Ze wzgledu na objętość tekstu listingów 6 i 8, publikujemy je wyłącznie na CD-EP5/2006B oraz na stronie internetowej w dziale Download.

List. 9. Funkcja Write
```
unsigned long Write(const void* Buffer, unsigned long NumberOfBytesToWrite)
{
     int iOutWrite;
     if (fd<1) return 0;
     iOutWrite = write(fd, Buffer, NumberOfBytesToWrite);
     if (iOutWrite<0) return 0;
     return iOutWrite;
}
```

List. 10. Funkcja Read
```
int Read(void * Buffer, unsigned long * lpNumberOfBytesRead, unsigned long NumberOfBytesToRead)
{
        unsigned long nNumberOfBytesToRead;
  int iInRead;
        if (NumberOfBytesToRead>cbInQueue)
                nNumberOfBytesToRead=cbInQueue;
        else
                nNumberOfBytesToRead=NumberOfBytesToRead;
        if(fd<1) return 0;
        iInRead=read(fd, Buffer, nNumberOfBytesToRead);
        if(iInRead<0)
                return 0;
        else
                *lpNumberOfBytesRead=iInRead;
        return 1;
}
```



Rys. 4. Okno aplikacji konsolowej

Wskaźnik *Buffer* określa miejsce w pamięci, gdzie zostaną zapisane odczytane bajty.

List. 11 przedstawia funkcję *CheckCommInput*, która pozwala zapytać sterownik portu szeregowego ile bajtów aktualnie znajduje się w buforze wejściowym. Ich liczba zwracana jest przez wskaźnik *lpInQue*.

Oczywiście, funkcje przedstawione na list. 8...11 (plik *LinuxRS232.c*) to tylko moja propozycja wykorzystania funkcji systemowych, a Czytelnicy zechcą zapewne napisać własne funkcje obsługi portu szeregowego (byłoby to nawet wskazane ze względów edukacyjnych).

List. 11. Funkcja CheckCommInput
```
void CheckCommInput(unsigned long * lpInQue)
{
        ioctl(fd, FIONREAD, lpInQue);
}
```

List. 12. Plik makefile
```
main: main.c LinuxRS232.c
  gcc -g -c -Wall main.c -o main.o
  gcc -g -c -Wall LinuxRS232.c -o LinuxRS232.o
  gcc LinuxRS232.o main.o -o main
```

Całość kompilujemy za pomocą GCC używając standardowo programu *make*. Przykładowy plik *makefile* zamieściłem na **list. 12.** Jest to bardzo prosty plik, mówiący kompilatorowi jedynie co kompilować, zaś linkerowi – z jakich plików *.o stworzyć plik wykonywalny o nazwie *main*. Aby otrzymać ten plik wystarczy wpisać „make” będąc w katalogu zawierającym pliki źródłowe aplikacji i plik *makefile*. Program uruchamiamy wpisując w konsoli „./main”.

Jeśli do portu COM1 komputera dołączony jest nasz zestaw, to będziemy mogli obserwować działanie całości. Polega ono na zadaniu pytania (wpisana przez użytkownika wartość) i obserwacji odpowiedzi. Pomiędzy zadaniem pytania a sprawdzeniem czy część sprzętowa na to pytanie odpowiedziała, program odczekuje pewien czas – w naszym przykładzie jest to 1 sekunda. Czas taki nazywany jest *break time*. Należy w tym miejscu zwrócić uwagę na różnicę w znaczeniu pojęć *break time* i *timeout* używanych często w opisach aplikacji komunikacyjnych.

Otóż *break time* jest minimalnym czasem jaki aplikacja musi odczekać na wystąpienie pewnego zdarzenia. Jeśli zdarzeniem tym jest odpowiedź układu pomiarowego dołączonego do komputera przez RS232, to *break time* jest wyznaczony przez czas dokonania pomiaru i czas potrzebny na przesłanie przez port szeregowy wyniku tego pomiaru. W naszym przypadku *break time* jest wyznaczony jedynie przez czas odpowiedzi – oczywiście 1 s jest czasem znacznie dłuższym niż minimalny czas jaki należałoby tutaj odczekać.

Czas *timeout* jest czasem przeterminowania (przedawnienia) pewnej operacji. Jeśli po upływie tego czasu chcemy ponownie wywołać tą operację, to musimy powtórzyć całą procedurę związaną z jej wywołaniem.

Podsumowanie

W tej części kursu pragnąłem zaprezentować wykorzystanie omówionych poprzednio funkcji na prostym przykładzie, tak aby bez wgłębiania się w szczegóły pokazać że „to działa”. Proponuję Czytelnikom poeksperymentować z opisanym zestawem i oprogramowaniem – ludzie wszak uczą się najlepiej i najszybciej na przykładach. W następnej części zmienimy nieco tematykę i zobaczymy jak oprogramowanie portu szeregowego wygląda w systemie Windows. Skupimy się przy tym na tych cechach, które są takie same lub podobne do ich odpowiedników występujących w Linuksie. Wszystko to ma na celu stworzenie klasy obsługi RS232 w języku C++ identycznej w obu systemach co do interfejsu, więc przenośnej. Klasę tą – w wersji dla Windows – stworzymy w następnej części cyklu. Oprócz tego, opiszemy też prostą aplikację GUI dla Windows, której funkcjonalność będzie taka sama jak funkcjonalność opisanego dziś prostego programu konsolowego. W dalszych częściach kursu aplikację tą przeniesiemy na Linuksa.

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

Programowanie portu szeregowego w systemach operacyjnych Linux i Windows, część 4

Umiejętność programowej obsługi interfejsu RS232 od strony komputera PC jest dziś istotnym elementem elektronicznego rzemiosła. W niniejszym kursie 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.

Po dłuższej przerwie wznawiamy kurs poświęcony programowaniu portu szeregowego w systemach operacyjnych Linux i Windows. W kolejnej części poznamy sposób obsługi łącza RS232 w systemie Windows, oraz stworzymy przenośną klasę *CCommInterface*, służącą do obsługi tego interfejsu w systemach Linux i Windows.

Obsługa łącza RS232 w systemie Windows

Sposoby obsługi portu RS232 są w dużej mierze podobne zarówno w systemie Windows, jak i w systemie Linux. Pewną różnicę stanowi brak plików urządzeń systemie Windows. Otwarcie portu polega na tymczasowym stworzeniu odpowiedniego pliku, zaś późniejsze jego używanie sprowadza się do zapisywania i odczytywania danych do/z utworzonego pliku. Podobnie jest realizowana komunikacja z innymi urządzeniami zewnętrznymi.

Otwieranie portu

Otwieranie portu szeregowego odbywa się za pomocą funkcji Win32 API *CreateFile()*, która tworzy plik związany z tym portem. Funkcja ta przyjmuje 7 parametrów i zapewnia różnorakie opcje związane z otwieraniem plików. Dla zagadnień związanych z obsługą portu szeregowego istotne jest jej typowe wykorzystanie, pozwalające na otworzenie portu do zapisu i odczytu, które przedstawiono na **list. 13**.

Konfiguracja portu

Po otwarciu, port musi zostać skonfigurowany. Jego konfiguracja odbywa się podobnie do modyfikacji struktury *termios* w systemie Linux. W systemie Windows modyfikowana jest specjalna struktura DCB. Schemat konfiguracji polega na pobraniu aktualnej wartości tej struktury za pomocą funkcji *GetCommState()*, jej zmodyfikowaniu oraz zapisaniu nowej wartości za pomocą funkcji *SetCommState()*. Prototypy tych funkcji są określone następująco:

```
BOOL GetCommState(
  HANDLE hCommDev,
  LPDCB lpDCB
);

BOOL SetCommState(
  HANDLE hCommDev,
  LPDCB lpDCB
);
```

Uchwyt *hCommDev* wskazuje na otwarty port szeregowy, zaś wskaźnik *lpDCB* wskazuje na strukturę DCB, do której mają zostać skopiowane dane o ustawieniach portu, lub z której dane mają być pobrane w celu modyfikacji tych ustawień.

Ważną rolę pełni funkcja *SetupComm()* służąca do ustalenia rozmiaru buforów: wejściowego i wyjściowego. Jest ona określona następująco:

```
BOOL SetupComm(
  HANDLE hCommDev,
  DWORD dwInQueue,
  DWORD dwOutQueue
);
```

Argumenty *dwInQueue* i *dwOutQueue* zawierają rozmiary buforów wejściowego i wyjściowego.

Przykład konfiguracji portu szeregowego w systemie Windows przedstawiono na **list. 14.** Taka konfiguracja pozwala na pracę z ramką typu 8N1 przy wyłączonej sprzętowej kontroli przepływu i szybkości transmisji 19200 bodów.

Czytanie z portu

Czytanie z bufora wejściowego portu szeregowego odbywa się za pomocą funkcji Win32 API określonej następująco:

```
BOOL ReadFile(
  HANDLE hCommDev,
  LPVOID lpBuffer,
  DWORD nNumberOfBytesToRead,
  LPDWORD lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped
);
```

List. 13. Otwarcie portu do zapisu i odczytu

```
hCommDev = CreateFile(lpFileName, GENERIC_READ |
                      GENERIC_WRITE, 0, NULL,
                      OPEN_EXISTING, 0, NULL);
```

List. 14. Przykładowa konfiguracja portu szeregowego (ramka 8N1, 19200 bodów)

```
if (hCommDev != INVALID_HANDLE_VALUE)
{
        dcb.DCBlength = sizeof(dcb);
        if(!GetCommState(hCommDev, &dcb))
                return false;
        dcb.BaudRate = 19200;
        //transmission parameters
        dcb.Parity = NOPARITY;
        dcb.StopBits = ONESTOPBIT;
        dcb.ByteSize = 8;
        //DCB flags
        dcb.fParity = FALSE;
        dcb.fDtrControl = DTR_CONTROL_DISABLE;
        //DTR line dislabled
        dcb.fRtsControl = RTS_CONTROL_DISABLE;
        //RTS line disabled
        dcb.fOutxCtsFlow = FALSE;
        dcb.fOutxDsrFlow = FALSE;
        dcb.fDsrSensitivity = FALSE;
        dcb.fAbortOnError = FALSE;
        dcb.fOutX = FALSE;
        dcb.fInX = FALSE;
        dcb.fErrorChar = FALSE;
        dcb.fNull = FALSE;
        if(!SetCommState(hCommDev,&dcb))
                return false;
        if(!SetupComm(hCommDev, cbInQueue, cbOutQueue))
                return false;
}
```

List. 15. Dyrektywy kompilacji warunkowej dla biblioteki Qt
```
#include <QtCore/QtGlobal>
//...
#ifdef Q_WS_WIN
    //Code compiled on Windows OS
#endif
#ifdef Q_WS_X11
    //Code compiled on Linux OS
#endif
```

Uchwyt *hCommDev* wskazuje na używany port szeregowy – jest to uchwyt zwracany podczas otwarcia portu przez funkcję *CreateFile()*. Argument *lpBuffer* jest wskaźnikiem na początek obszaru pamięci, w którym zostaną umieszczone dane odczytane z portu. Argument *nNumberOfBytesToRead* specyfikuje ile bajtów ma być odczytanych z portu, zaś argument *lpNumberOfBytesRead* wskazuje na zmienną, w której jest umieszczana liczba faktycznie odczytanych bajtów. Liczby te mogą się różnić wtedy, gdy w buforze wejściowym znajduje się mniej bajtów niż podano w zmiennej *nNumberOfBytesToRead*, lub gdy nastąpił błąd odczytu. Ostatni argument funkcji *ReadFile()* – *lpOverlapped* – nie jest w tym przypadku wykorzystywany i jego wartość podczas jej wywołania wynosi *NULL*. Funkcja *ReadFile()* zwraca wartość logiczną *true*, gdy odczyt przebiegł prawidłowo, zaś *false*, gdy wystąpił błąd.

Pisanie do portu

Wysyłanie danych do bufora wyjściowego portu szeregowego odbywa się za pomocą funkcji Win32 API określonej następująco:

```
BOOL WriteFile(
  HANDLE hCommDev,
  LPCVOID lpBuffer,
  DWORD nNumberOfBytesToWrite,
  LPDWORD lpNumberOfBytesWritten,
  LPOVERLAPPED lpOverlapped
);
```

Uchwyt *hCommDev* wskazuje na używany port szeregowy. Argument *lpBuffer* jest wskaźnikiem na początek obszaru pamięci, w którym są umieszczone dane przeznaczone do wysłania do portu. Argument *nNumberOfBytesToWrite* specyfikuje ile bajtów ma być wysłanych do portu, zaś argument *lpNumberOfBytesWritten* wskazuje na zmienną, w której umieszczana jest liczba faktycznie wysłanych bajtów. Liczby te mogą się różnić, gdy nastąpił błąd zapisu. Ostatni argument funkcji *WriteFile()* – *lpOverlapped* – nie jest w tym przypadku wykorzystywany i jego wartość podczas jej wywołania wynosi *NULL*. Funkcja *WriteFile()* zwraca wartość logiczną *true*, gdy zapis przebiegł prawidłowo, zaś *false*, gdy wystąpił błąd.

Sprawdzenie liczby bajtów oczekujących na odczyt w buforze wejściowym

Sprawdzenie ile nie odczytanych bajtów oczekuje w buforze wejściowym odbywa się za pomocą funkcji Win32 API określonej następująco:

```
BOOL ClearCommError(
  HANDLE hCommDev,
  LPDWORD lpErrors,
  LPCOMSTAT lpStat
);
```

Wywołanie tej funkcji pozwala poznać kod ostatniego błędu (wskazywany przez argument *lpErrors*), jaki pojawił się podczas pracy portu szeregowego (funkcja jednocześnie zeruje znacznik błędu) oraz odczytać stan portu, który jest umieszczany w specjalnej strukturze *COMSTAT*, dostępnej poprzez wskaźnik *lpStat*. Jednym z pól tej struktury jest pole *cbInQue* zawierające liczbę nie odczytanych bajtów, jakie oczekują w buforze wejściowym portu szeregowego. Argument *hCommDev* jest uchwytem do używanego portu.

Zamykanie portu

Zamykanie portu szeregowego odbywa się za pomocą funkcji Win32 API określonej następująco:

```
BOOL CloseHandle(HANDLE *hCommDev);
```

Funkcja zamyka port, a ściślej – związany z nim plik utworzony za pomocą funkcji *CreateFile()*, wskazywany przez uchwyt *hCommDev*.

Klasa CCommInterface

Klasa *CCommInterface* posiada uniwersalne metody implementujące podstawowe operacje na porcie szeregowym, a więc: otwarcie portu, zamknięcie portu, zapis ciągu bajtów, odczyt ciągu bajtów oraz sprawdzenie liczby nie odczytanych bajtów znajdujących się w buforze wejściowym. Oto lista prototypów publicznych metod tej klasy:

```
bool Open(string CommID, unsigned
long Baud);

bool Close(void);

unsigned long Write(const
void *Buffer, unsigned long
NumberOfBytesToWrite);

bool Read(void *Buffer, unsigned
long *lpNumberOfBytesRead, unsigned
long NumberOfBytesToRead);
```

List. 16. Plik nagłówkowy klasy CCommInterface
```
#ifndef CommInterfaceH
#define CommInterfaceH
#include <QtCore/QtGlobal>
#include <string>
using namespace std;
//---------------------------------------------------------------------------
class CCommInterface
{
private:
        const char *lpFileName;
        unsigned long BaudRate;
        unsigned long cbInQueue;
        unsigned long cbOutQueue;
#ifdef Q_WS_WIN
        void *hCommDev;
#endif
#ifdef Q_WS_X11
        int fd;
#endif
protected:
public:
        CCommInterface();
        bool Open(string CommID, unsigned long Baud);
        void Close(void);
        unsigned long Write(const void* Buffer, unsigned long NumberOfBytesToWrite);
        bool Read(void * Buffer, unsigned long * lpNumberOfBytesRead, unsigned long NumberOfBytesToRead);
        void CheckCommInput(unsigned long * lpErrors, unsigned long * lpInQue);
};
//---------------------------------------------------------------------------
#endif
```

```
bool CheckCommInput(unsigned long
*lpErrors, unsigned long *lpInQue);
```

Nazwy poszczególnych argumentów jednoznacznie wskazują na ich znaczenie, nie ma więc potrzeby szczegółowego wyjaśniania roli każdego z nich. Funkcja *Write()* zwraca liczbę faktycznie zapisanych bajtów (w przypadku błędu zwraca 0). Funkcja *Open()* jako jedyne argumenty przyjmuje identyfikator portu i szybkość transmisji – port jest otwierany do pracy z ramką 8N1 przy wyłączonej sprzętowej kontroli przepływu. Klasę tę można dowolnie zmodyfikować według potrzeb, w celu zapewnienia innych ustawień portu, jeśli takie będą potrzebne.

Implementacja metod klasy *CCommInterface* polegała na zebraniu wiadomości dotyczących obsługi portu w obu systemach operacyjnych i połączeniu ich poprzez dyrektywy kompilacji warunkowej. Dzięki temu, fragmenty właściwe dla systemu Windows będą kompilowane, gdy kompilacja będzie dokonywana w tym systemie, zaś fragmenty właściwe dla systemu Linux, gdy kompilacja będzie dokonywana w systemie Linux. Dyrektywy kompilacji warunkowej opierają się na predefiniowanych stałych, charakterystycznych dla używanego kompilatora czy biblioteki. Już teraz zdradzę, że przykłady aplikacji okienkowych omówione w kolejnych częściach kursu będą stworzone za pomocą rewelacyjnej i rozwojowej biblioteki Qt firmy Trolltech. Dyrektywy kompilacji warunkowej charakterystyczne dla tej biblioteki przedstawiono na **list. 15**. Stałe *Q_WS_WIN* i *Q_WS_X11* są zdefiniowane w pliku *QtGlobal*. Plik ten musi być wcześniej włączony do kodu aplikacji (oczywiście, niekoniecznie w tym samym pliku źródłowym, w którym wykorzystuje się zdefiniowane w nim stałe). Dla innych bibliotek i kompilatorów istnieją inne stałe określające system operacyjny. Adaptacja klasy *CCommInterface* do pracy z nimi polega tylko i wyłącznie na zamianie tych stałych. Reszta kodu źródłowego pozostaje bez zmian.

W celu przekazania nazwy portu („COM1”, „COM2”) do funkcji *Open()* wykorzystano uniwersalny typ *string* z biblioteki STL (*Standard Template Library*). Dzięki temu istnieje pewność co do przekazywanych danych tekstowych niezależnie od platformy, na której dokonywana jest kompilacja. Biblioteka ta jest przy tym tak bardzo popularna, że jej implementacja dostarczana jest wraz z każdym szanującym się kompilatorem. Plik nagłówkowy klasy *CCommInterface* przedstawiono na **list. 16**.

Podsumowanie

Wiemy już jak obsłużyć port RS232 w systemach Linux i Windows, dysponujemy także uniwersalną klasą C++, która to umożliwia. Następne części kursu będą poświęcone tworzeniu prostych aplikacji GUI wykorzystujących klasę *CCommInterface*.

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

1/8L

180x93 mm

Programowanie portu szeregowego w systemach operacyjnych Linux i Windows, część 5

Umiejętność programowej obsługi interfejsu RS232 od strony komputera PC jest dziś istotnym elementem elektronicznego rzemiosła. W niniejszym kursie 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.

W poprzednich częściach kursu poznaliśmy, w jaki sposób można oprogramować port szeregowy w systemach Linux i Windows (32–bitowych). Została przedstawiona uniwersalna klasa obsługi tego interfejsu, której wykorzystanie jest możliwe z każdym kompilatorem języka C++ i biblioteką STL. W następnych odcinkach pokażę przykład prostej aplikacji okienkowej działającej w obu systemach operacyjnych i współpracującej z programem Example1.c pracującym na mikrokontrolerze ATmega8. Funkcjonalność tej aplikacji polega na wysłaniu jednobajtowego zapytania i prezentacji odpowiedzi przesłanej przez mikrokontroler, lub informacji o braku takiej odpowiedzi. Jako narzędzie do stworzenia programu przenośnego wybrano bibliotekę Qt 4 (wersja Qt 4.0.1 *Open Source*) firmy Trolltech.

Dlaczego Qt?

Na rynku istnieje wiele narzędzi służących do tworzenia przenośnych aplikacji z graficznym interfejsem użytkownika. Do niedawna obiecująco zapowiadał się rozwój pakietu IDE (*Integrated Development Environment*) Kylix firmy Borland, który wraz ze środowiskami Delphi i C++ Builder, stanowił zapowiedź dobrego zestawu narzędzi RAD (*Rapid Application Development*) dla systemów Linux i Windows. Niestety, w 2005 roku firma Borland zrezygnowała z dalszego rozwoju środowiska Kylix, traktując je na równi z tak leciwymi produktami jak Borland Turbo C++. Na szczęście na przenośnych IDE świat się nie kończy i stworzono wiele przenośnych bibliotek, a więc programiści mają w czym wybierać. Określenie „przenośnych" dotyczy tu kodu źródłowego, a nie – jak to ma miejsce w przypadku języków Java, C#, czy platformy .NET – interpretowanych na podstawie tzw. kodu bajtowego, gdzie konieczna jest wcześniejsza instalacja Wirtualnej Maszyny Javy lub jej odpowiednika. Wykorzystanie bibliotek przenośnych odbywa się zgodnie z maksymą „napisz raz, skompiluj (i uruchom) wszędzie". Ten sam kod źródłowy kompilowany jest na różnych platformach, w wyniku czego dla każdej z nich powstaje natywny kod wykonywalny.

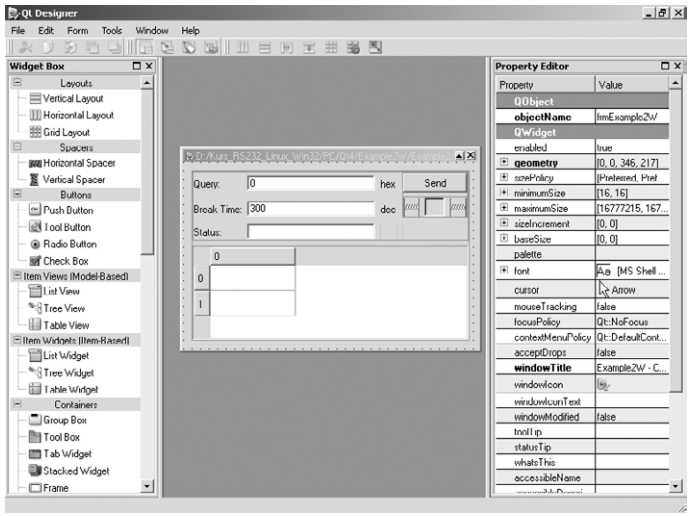
Najbardziej popularnymi bibliotekami służącymi do tworzenia aplikacji przenośnych są: wxWidgets (dawniej wxWindows), Qt, Ultimate++, FOX, V. Różnią się one przede wszystkim możliwościami, elegancją dostarczanego przez nie API, łatwością tworzenia interfejsu graficznego w sposób wizualny, funkcjonalnością wersji dostępnych bezpłatnie, jakością dokumentacji, czy wreszcie popularnością i perspektywami rozwoju. Biorąc pod uwagę wymienione cechy, toolkit Qt (czytaj *ang. cute* – miły, sympatyczny) wyróżnia się eleganckim API (w przeciwieństwie np. do API wxWidgets, przypominającego bibliotekę MFC), dużymi możliwościami, dobrą dokumentacją (Qt Assistant) i świetnym narzędziem do tworzenia interfejsu użytkownika w sposób wizualny – Qt Designer. Jest przy tym dostępny bezpłatnie w formie *Open Source* na licencji GPL, zarówno dla systemu Linux jak i Windows (począwszy od wersji 4). Jest to biblioteka stale rozwijana przez firmę Trolltech. Co ciekawe, w parze z jej rozwojem idzie rozwój popularnego pulpitu dla systemów linuksowych – KDE, który jest tworzony właśnie z użyciem Qt.

Kolejną zaletą Qt jest możliwość współpracy z wieloma kompilatorami C++. Na potrzeby tego kursu użyto rozwiązań *Open Source*, co zaowocowało wykorzystaniem kompilatora GCC, który w wersji dla systemu Windows dostępny jest m.in. w postaci pakietu MinGW.

Instalacja biblioteki Qt i narzędzi wspomagających w systemie Windows

Wersję *Open Source* pakietu Qt w wersji 4.0.1, zarówno dla systemu Windows jak i Linux, można pobrać z *ftp://ftp.trolltech.com/qt/source/*. Dla Windows, biblioteka jest dostarczana w postaci pliku *qt–win–opensource–src–4.0.1.zip*. Archiwum należy rozpakować w miejscu, w którym chcemy mieć na dysku twardym zainstalowaną bibliotekę Qt. W moim przypadku był to katalog C:\qt–win–opensource–src–4.0.1. Ze względu na to, że wersja darmowa, jako pakiet *Open Source*, rozpowszechniana jest w wersji źródłowej, instalacja polega na jej skompilowaniu. W wyniku kompilacji powstają niezbędne narzędzia i biblioteki. Jednak zanim przystąpimy do kompilacji, musimy zaopatrzyć się w kompilator, na przykład MinGW. Kompilator ten można ściągnąć osobno (*http://www.mingw.org/*) lub wraz z jednym z wielu darmowych edytorów w stylu IDE. W niniejszym kursie wykorzystano IDE *Code Blocks* (w skrócie C::B), dostępne wraz z kompilatorem MinGW na stronie internetowej *http://www.codeblocks.org/*. Jedną z zalet tego edytora jest to, że posiada on wzorzec (*template*) projektu Qt, co bardzo ułatwia jego integrację z tą biblioteką. Co ciekawe, sam edytor jest tworzony z użyciem konkurencyjnego pakietu wxWidgets. Edytor C::B wraz z kompilatorem MinGW należy zainstalować w typowy dla systemu Windows sposób.

Po zainstalowaniu edytora i kompilatora przystępujemy do zainstalowania biblioteki Qt. Sama instalacja tego pakietu jest szczegółowo opisana



Rys. 4. Tworzenie graficznego interfejsu aplikacji za pomocą programu Qt Designer

w pliku INSTALL, dostarczanym wraz z nim. Niestety, jedną z największych wad biblioteki Qt jest bardzo długi czas instalowania, mogący dochodzić nawet do kilku godzin (i to nawet na całkiem szybkich komputerach). Dlatego też warto zrezygnować zarówno z tworzenia plików Makefile, przykładów, jak i ich kompilacji (kod źródłowy przykładów jest umieszczony w katalogu \examples). W razie potrzeby można je skompilować później. Podczas opisu instalacji i użytkowania biblioteki Qt odniesienia do katalogu w jakim ma być ona zainstalowana będą dotyczyć przykładowego katalogu C:\qt–win–opensource–src–4.0.1. Procedura instalacji jest następująca:

1. Nadać poniższym zmiennym środowiskowym następujące wartości:
   PATH – dodać ścieżkę

```
C:\qt-win-opensource-src-4.0.1\bin
QTDIR=C:\qt-win-opensource-src-4.0.1
QMAKESPEC=C:\qt-win-opensource-src-
4.0.1\mkspecs\win32-g++
```

   Dostęp do zmiennych środowiskowych w nowych systemach Windows NT odbywa się poprzez *Panel Sterowania–>System–>Zaawansowane–>Zmienne środowiskowe*. Można też w linii poleceń wpisać:

```
set NAZWA_ZMIENNEJ=ŚCIEŻKA
```

2. W linii poleceń wpisać:

```
configure
```

   Program *configure* służy do przygotowania programu *qmake*, który z kolei służy do tworzenia plików *Makefile* na podstawie plików źródłowych. Zagadnienia te zostaną opisane dalej. Działanie programu *configure* można przerwać po stworzeniu programu *qmake* (katalog QTDIR\bin), w czasie, gdy tworzone są pliki *Makefile* projektów przykładowych (w tym czasie na ekranie monitora wyświetlane są napisy „reading *.pro" odnoszące się do plików z katalogu QTDIR\examples);
3. Po skonfigurowaniu należy sprawdzić, czy w katalogu QTDIR znajduje się plik Makefile. Wersje *Open Source* są dostarczane bez jakiejkolwiek gwarancji i zdarza się, że plik ten nie zostanie stworzony. Jeśli go nie ma, należy utworzyć go ręcznie, wpisując w linii poleceń:

```
qmake
```

4. W linii poleceń wpisać:

```
mingw32-make
```

   Polecenie to jest wywołaniem programu *make* właściwego dla kompilatora MinGW. Po jego wywołaniu nastąpi długotrwały proces kompilacji. Wraz z jego postępem, w katalogu QTDIR\lib będą się pojawiać stosowne biblioteki, a w katalogu QTDIR\bin będą to biblioteki **.dll* oraz programy narzędziowe, między innymi:
   - *moc.exe* (*Meta–Object Compiler*). Kompilator meta obiektów biblioteki Qt;
   - *uic.exe* (*User Interface Compiler*). Konwerter plików XML *.ui, będących wynikiem działania programu Qt Designer, na kod źródłowy w języku C++;
   - *designer.exe* – narzędzie do tworzenia GUI;
   - *assistant.exe* – pomoc biblioteki Qt;
   - *linguist.exe* – narzędzie do internacjonalizacji aplikacji tworzonych za pomocą Qt.
5. Kompilację można przerwać w czasie, gdy kompilowane są przykładowe projekty. Każdy z nich można, wedle potrzeb, skompilować później.

Po wykonaniu opisanych czynności biblioteka Qt 4.0.1 jest gotowa do użycia.

Tworzenie aplikacji testowej z Graficznym Interfejsem Użytkownika

Wykorzystanie biblioteki Qt jest zagadnieniem na tyle szerokim, że z powodzeniem mogłoby stanowić tematykę niemałej wielkości książki. Dlatego też opisano tylko te jego elementy, które są najistotniejsze z punktu widzenia tematyki niniejszego kursu.

W celu zademonstrowania obsługi łącza RS232 przez aplikację okienkową, stworzono program Example2W, którego działanie jest analogiczne do działania programu Example1L, opisanego w jednej z poprzednich części kursu. Aplikacja umożliwia wysłanie jednobajtowego zapytania, ustalenie czasu *break time* oraz prezentację odpowiedzi udzielonej przez część sprzętową.

Tworzenie aplikacji rozpoczynamy od uruchomienia programu *Qt Designer*, znajdującego się w katalogu QTDIR\bin. Rozmieszczając widgety (linuksowe odpowiedniki okien, przy czym w nomenklaturze WinAPI za okno uważa się także takie obiekty, jak np. przyciski) na formie aplikacji należy posłużyć się zestawem odpowiednich rozkładów (*layout*), dzięki czemu finalny program będzie się zachowywał sensownie, na przykład podczas prób rozciągania formy. Zachowanie formy można w każdej chwili wypróbować za pomocą opcji *preview* (*Ctrl+R*). Dostępne są różne style podglądu, między innymi styl Windows czy Motif. Zrzut ekranowy programu *Qt Designer* zrobiony podczas tworzenia formy aplikacji testowej, przedstawiono na **rys. 4**.

Po zaprojektowaniu wyglądu formy, należy ją zapisać w wybranym katalogu, najlepiej w tym, w którym będzie umieszczony projekt aplikacji. Forma jest zapisywana w postaci specjalnego pliku XML z rozszerzeniem *.ui. W przypadku aplikacji testowej *Example2W*, plik z danymi formy nazwano *Example2WFrm.ui*. Fragment tego pliku przedstawiono na **list. 17**. Aby z niego skorzystać w tworzonej aplikacji, należy dokonać jego konwersji na plik nagłówkowy języka C++. Można to zrobić za pomocą programu narzędziowego *uic.exe*, znajdującego się w katalogu QTDIR\bin. W tym celu, będąc w katalogu projektu, należy wpisać w linii poleceń:

```
uic –o ui_Example2WFrm.h Example2WFrm.
ui
```

Powyższa komenda spowoduje stworzenie pliku *ui_Example2WFrm.h*,

List. 17. Fragment pliku z danymi formy Example2WFrm.ui
```
<ui version="4.0" >
 <author></author>
 <comment></comment>
 <exportmacro></exportmacro>
 <class>frmExample2W</class>
 <widget class="QDialog"
name="frmExample2W" >
  <property name="geometry" >
   <rect>
    <x>0</x>
    <y>0</y>
    <width>346</width>
    <height>213</height>
   </rect>
  </property>
  <property name="windowTitle" >
   <string>Example2W - COM1</string>
  </property>
```

którego szkielet przedstawiono na **list. 18**. Plik ten zawiera definicję klasy pomocniczej o nazwie Ui_frmExample2W, której pola stanowią deklaracje poszczególnych widgetów i innych elementów interfejsu graficznego. Klasa ta jest dziedziczona przez klasę frmExample2W, umieszczoną w przestrzeni nazw Ui. Sposób jej wykorzystania we właściwym pliku nagłówkowym formy zdradza **list. 19**. Klasą formy aplikacji Example2W jest klasa Example2WFrm, będąca potomkiem klasy okna dialogowego *QDialog*. Klasa Example2WFrm posiada prywatne pole o następującej definicji:

```
Ui::frmExample2W ui;
```

Dzięki temu, wszystkie elementy Graficznego Interfejsu Użytkownika, stworzonego za pomocą programu *QT Designer*, są po prostu jednym z pól klasy formy. Można się do nich odwoływać w ciele tej klasy za pomocą formuły:

```
ui.nazwa_elementu_gui
```

List. 18. Szkielet pliku źródłowego ui_Example2WFrm.h, określającego interfejs graficzny formy
```
class Ui_frmExample2W
{
  //Deklaracje pol i metod klasy
};
namespace Ui {
    class frmExample2W: public
Ui_frmExample2W {};
} // namespace Ui
```

List. 19. Właściwy plik nagłówkowy formy Example2WFrm.h
```
#include „ui_Example2WFrm.h"
#include „CommInterface.h"
#include <QtCore/QTimer>
class Example2WFrm : public QDialog
{
  Q_OBJECT
public:
  Example2WFrm(QDialog *parent = 0);
  ~Example2WFrm();
private slots:
  void timerBreakTimeOverflow(void);
  void clickedSendQuery(void);
private:
  QTimer *timerBreakTime;
  CCommInterface *pComm;
  Ui::frmExample2W ui;
};
```

Stworzenie obiektu formy i jego uwidocznienie odbywa się w funkcji *main()*, zaimplementowanej w pliku *main.cpp*, który pokazano na **list. 20**.

List. 20. Funkcja main()
```
#include <QtGui/QApplication>
#include „Example2WFrm.h"
int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
  Example2WFrm frmEx;
  frmEx.show();
  return app.exec();
}
```

Sygnały i sloty

Wiemy już, jak zbudować formę aplikacji opartej na oknie dialogowym i jak ją pokazać użytkownikowi. Pozostało odpowiedzieć na pytanie, w jaki sposób można tę aplikację ożywić, czyli dodać funkcje obsługi zdarzeń, takich jak na przykład kliknięcie klawiszem myszki na przycisku. Otóż, w przypadku biblioteki Qt, odbywa się to za pośrednictwem mechanizmu **sygnałów i slotów**. Mechanizm sygnałów i slotów polega na tym, że określone zdarzenie dotyczące danego widgeta (np. kliknięcie myszką) powoduje wyemitowanie przezeń sygnału, zależnego od tego, jakie zdarzenie miało miejsce. Każdy widget lub forma (ogólnie: każda klasa wywodząca się od klasy QObject) może emitować sygnały oraz posiadać specjalne metody zwane slotami. Sloty to po prostu funkcje wywoływane w odpowiedzi na określone sygnały, które to sygnały są z kolei funkcjami wywoływanymi w przypadku wystąpienia odpowiadających im zdarzeń. Aby dany slot był wywoływany w odpowiedzi na dany sygnał, należy go z tym sygnałem połączyć za pomocą specjalnej funkcji *connect()*. Zasadę tę ilustruje **rys. 5**. Nie każdy slot da się połączyć z danym sygnałem. Połączenie jest możliwe, gdy lista argumentów slotu jest podzbiorem listy argumentów sygnału. Slot, połączony z danym sygnałem, wywoływany jest z tymi samymi argumentami, z którymi wywoływany jest sygnał (lub ich podzbiorem właściwym dla listy argumentów slotu). Jeden slot może być połączony z dowolną liczbą sygnałów, a jeden sygnał może być połączony z dowolną liczbą slotów. Biblioteka Qt zapewnia dużą liczbę predefiniowanych sygnałów i slotów, oprócz tego każdy jej użytkownik może tworzyć własne.

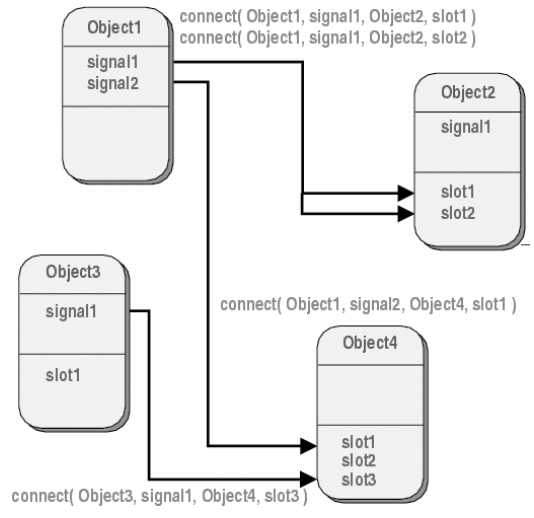
List. 21. Konstruktor klasy Example2WFrm
```
Example2WFrm::Example2WFrm(QDialog
*parent)
    : QDialog(parent)
{
  ui.setupUi(this);
  //Create members
  pComm = new CCommInterface();
  timerBreakTime = new
QTimer(this);
  //Set text codec for tr()
function
  QTextCodec::setCodecForTr(QTextCo
dec::codecForName(„ISO8859-2"));
  //Connect signals and slots
  connect(timerBreakTime,
SIGNAL(timeout()), this, SLOT(timer
BreakTimeOverflow()));
  connect(ui.pbtnSendQuery,
SIGNAL(clicked()), this, SLOT(click
edSendQuery()));
}
```

Spójrzmy ponownie na list. 19. Klasa Example2WFrm posiada dwa sloty, wymienione w sekcji *private slots*. Są nimi funkcje, jakie mają być wywoływane w odpowiedzi na kliknięcie przycisku *Send* oraz przepełnienie timera *timerBreakTime*, będącego polem tej klasy. Na **list. 21** przedstawiono konstruktor klasy Example2WFrm. Jak widać, w konstruktorze sloty łączone są z odpowiednimi sygnałami. Ta prosta operacja wystarcza, aby zrealizować mechanizm reagowania aplikacji na wybrane zdarzenia.

Ciąg dalszy opisu tworzenia aplikacji testowej działającej w systemie Windows przedstawimy w następnej części kursu.

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

Linki internetowe:
ftp://ftp.trolltech.com/qt/source/
http://www.mingw.org/
http://www.codeblocks.org/



Rys. 5. Mechanizm sygnałów i slotów

Programowanie portu szeregowego w systemach operacyjnych Linux i Windows, część 6

Umiejętność programowej obsługi interfejsu RS232 od strony komputera PC jest dziś istotnym elementem elektronicznego rzemiosła. W niniejszym kursie 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.

Wykorzystanie łącza RS232 w aplikacji testowej

Funkcje obsługi interfejsu RS232 dostarczane są za pomocą klasy *CCommInterface*. Plik nagłówkowy tej kasy jest włączany do pliku nagłówkowego głównej formy aplikacji (list. 19). Klasa tej formy posiada pole będące wskaźnikiem do obiektu interfejsu komunikacyjnego:

CCommInterface *pComm;

Obiekt tego interfejsu tworzony jest dynamicznie w konstruktorze klasy *Example2WFrm* wraz z obiektem licznika odmierzającego czas *break time* (list. 21).

Jak wspomniano, działanie aplikacji polega na wysłaniu zapytania i prezentacji odpowiedzi. Zapytanie jest wysyłane w chwili kliknięcia przycisku *Send,* zaś odpowiedź, będąca ciągiem bajtów, prezentowana jest w formie szesnastkowej i ASCII za pomocą obiektu klasy *QTableWidget*. Klasa ta implementuje tabelkę (siatkę) mogącą zawierać dowolne widgety. Odpowiedź jest sprawdzana po upłynięciu czasu *break time* liczonego od chwili wysłania zapytania. Czas ten jest ustalany przez użytkownika aplikacji i odmierzany za pomocą licznika *timerBreakTime* typu *QTimer*.

Ciało slotu obsługującego kliknięcie przycisku *Send* przedstawiono na **list. 22**. W pierwszej kolejności dokonywane jest otwarcie portu COM1 dla szybkości transmisji 19200 bodów. Niepowodzenie skutkuje wyświetleniem odpowiedniego komunikatu i zamknięciem aplikacji. Jeśli port został poprawnie otwarty, to z pól tekstowych odczytywane jest zapytanie oraz wartość czasu *break time* wyrażonego w milisekundach. Następnie czyszczone jest pole odpowiedzi, po czym zapytanie jest wysyłane przez łącze RS232 i włączany jest licznik *timerBreakTime*. Na czas trwania oczekiwania (*break time*) przycisk *Send* jest wyłączany. Aplikacja jest wyposażona w kontrolkę, która „świeci się" na jasno niebiesko (kolor *cyan*) podczas trwania czasu *break time*.

Po upływie tego czasu aplikacja sprawdza, jaka (i czy w ogóle) odpowiedź nadeszła. Zadanie to jest realizowane w slocie obsługującym przepełnienie licznika *timerBreakTime*, pokazanym na **list. 23**. Po wyłączeniu kontrolki i zatrzymaniu licznika, program sprawdza za pomocą

List. 22. Wysyłanie jednobajtowego zapytania

```
void Example2WFrm::clickedSendQuery()
{
  //Open port
  string port="COM1";
  if(pComm->Open(port,19200)==false)
  {
    QString portname=port.c_str();
    QMessageBox::critical(this,tr(„Error"),tr(„Port „)+portname+tr(„ opening
error. Application will terminate."));
    this->reject();
  }

  //Read query
  bool ok;
  unsigned char qry=ui.leQuery->text().toInt(&ok,16);

  if(false==ok)
  {
    QMessageBox::critical(this,tr(„Error"),tr(„Invalid query value."));
    pComm->Close();
    return;
  }

  //Update displayed query value
  ui.leQuery->setText(QString::number(qry,16));


  //Read break time
  int btime=ui.leBreakTime->text().toInt(&ok);

  if((false==ok) || (btime<0))
  {
    QMessageBox::critical(this,tr(„Error"),tr(„Invalid break time value."));
    pComm->Close();
    return;
  }

  //Clear response window
  ui.twResponse->clear();
  ui.twResponse->setRowCount(2);
  ui.twResponse->setColumnCount(1);
  ui.leStatus->setText(tr(„"));

  //Send query
  ui.pbtnSendQuery->setEnabled(false);

  if(pComm->Write(&qry,1)!=1)
  {
    QMessageBox::critical(this,tr(„Error"),tr(„Write error."));
    pComm->Close();
    return;
  }

  QPalette pal;
  pal.setColor(QPalette::Base,QColor(tr(„cyan")));
  ui.leLamp->setPalette(pal);

  this->timerBreakTime->start(btime);
}
```

List. 23. Odbiór odpowiedzi

```
void Example2WFrm::timerBreakTimeOverflow()
{
  QPalette pal;
  pal.setColor(QPalette::Base,QColor(tr(„black”)));
  ui.leLamp->setPalette(pal);

  this->timerBreakTime->stop();

  //Read response
  unsigned long Errors,InQue,BytesRead;
  unsigned char *Input;

  pComm->CheckCommInput(&Errors,&InQue);
  if(Errors!=0)
    ui.leStatus->setText(tr(„*** ERROR „)+QString::number(Errors)+tr(„
***”));
  else
   {
     if(InQue==0)
       ui.leStatus->setText(tr(„*** NO RESPONSE ***”));
     else
       {
          ui.leStatus->setText(tr(„*** OK ***”));

          //Read response
          Input=new unsigned char[InQue];
          if(false==pComm->Read(Input,&BytesRead,InQue))
            {
              QMessageBox::critical(this,tr(„Error”),tr(„Read error.”));
              delete [] Input;
              pComm->Close();
              ui.pbtnSendQuery->setEnabled(true);
              return;
            }

          //Show response
          ui.twResponse->setColumnCount(BytesRead);
          for(unsigned long j=0;j<BytesRead;j++)
          {
            QTableWidgetItem *hexItem = new QTableWidgetItem(QString::
number(Input[j],16));
            QTableWidgetItem *charItem = new QTableWidgetItem(QString((char)
(Input[j])));

            ui.twResponse->setItem(0, j, hexItem);
            ui.twResponse->setItem(1, j, charItem);
          }

          delete [] Input;
       }
   }

  pComm->Close();

  ui.pbtnSendQuery->setEnabled(true);
}
```

funkcji *CheckCommInput()* ile bajtów oczekuje w buforze wejściowym oraz czy nie wystąpiły błędy transmisji na poziomie kontrolera UART i systemu operacyjnego. W przypadku błędu wyświetlany jest odpowiedni komunikat. Brak odpowiedzi (zero bajtów w buforze) jest sygnalizowany poprzez wyświetlenie komunikatu „*** NO RESPONSE ***”. Jeśli zaś odpowiedź nadeszła, wyświetlany jest komunikat „*** OK ***”, a odpowiedź jest odczytywana z bufora wejściowego za pomocą funkcji *Read()*. Odczytany ciąg bajtów trafia do tabelki prezentującej odpowiedź (w tym przypadku każdy z jej elementów przechowuje obiekt klasy *QString*). Na koniec port szeregowy jest zamykany, a przycisk *Send* ponownie włączany.

Kompilacja i uruchomienie programu z linii poleceń

Przed kompilacją projektu, należy przygotować odpowiednie pliki *Makefile*. Pliki te są tworzone na podstawie plików źródłowych za pomocą wspomnianego już programu narzędziowego *qmake.exe*. Będąc w katalogu projektu wpisujemy w linii poleceń:

qmake – project

a następnie:

qmake

Pierwsze z wymienionych poleceń spowoduje stworzenie pliku *Example2W.pro*, który pokazano na **list. 24**. Plik ten stanowi informację, które pliki (zarówno źródłowe, jak i pliki **.ui*) należy brać pod uwagę przy tworzenia plików *Makefile*, a także dodatkowe opcje. Linia:

DEFINES –= UNICODE

została dodana ręcznie po to, aby biblioteka Qt traktowała znaki *char* jako znaki ASCII, a nie jak znaki zapisane w *Unicode* (*Wide Char*). Jest to potrzebne, bo prototypy funkcji WinAPI wymagają typu ASCII. Mimo to, biblioteka Qt w pełni wspiera *Unicode* i możliwe jest używanie 16–bitowych znaków *Unicode* – są one implementowane przez typ *QChar*.

Drugie z wymienionych poleceń powoduje stworzenie trzech plików: *Makefile, Makefile.Debug i Makefile.Release*, które oczywiście umożliwiają tworzenie wersji *Debug* i *Release* aplikacji. Kompilacji wersji *Debug* dokonujemy wpisując:

mingw32–make –f Makefile.Debug

zaś wersji Release, wpisując:

mingw32–make –f Makefile.Release

Uruchomienie aplikacji może odbyć się bez podania parametrów wywołania (co odpowiada kliknięciu myszy na pliku *Example2W.exe*), lub przykładowo z podaniem stylu w jakim program ma zostać uruchomiony. Dostępne są następujące style (użytkownik może także definiować własne):

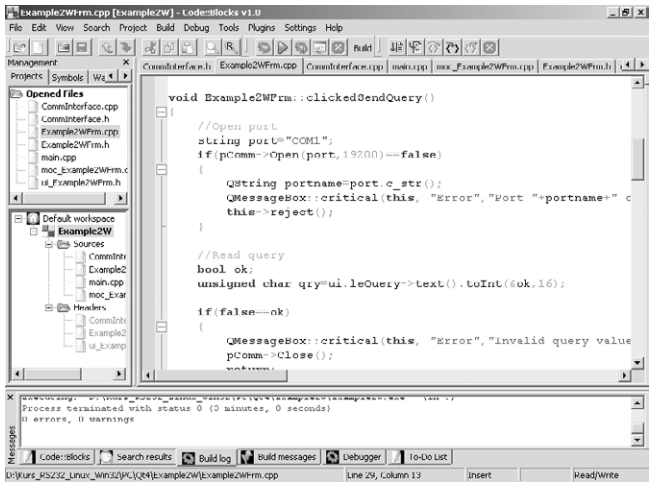
- Windows (domyślny w systemie Windows)
- WindowsXP
- Motif
- CDE
- Macintosh
- Plastique (domyślny w systemie Linux)

Uruchomienie aplikacji z użyciem danego stylu odbywa się po podaniu jego nazwy z opcją *–style*. Przykładowo, aplikację *Example2W* w stylu *Plastique* uruchamiamy za pomocą polecenia:

Example2W.exe –style=plastique

Integracja biblioteki Qt z edytorem Code Blocks

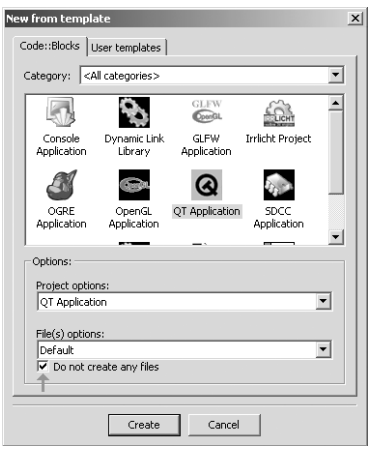
O ile możliwa jest praca z plikami źródłowymi projektu za pomocą zwykłego notatnika, to oczywiście wygodnie jest nimi operować za pomocą dowolnego edytora przeznaczonego dla programistów, wyposażonego co najmniej w możliwość kolorowania składni. Edytor *Code Blocks*, oprócz tej podstawowej cechy, posiada między innymi wzorzec projektu Qt, dzięki któremu tworzenie własnego projektu wykorzystującego tę bibliotekę jest wyjątkowo łatwe. Posiada on także możliwość współpracy z debuggerem GDB, będącym częścią środowiska MinGW, co znakomicie ułatwia (a niekiedy wręcz warunkuje) proces tworzenia aplikacji. Ponadto, integracja z GDB jest dokonywana automatycznie, a programista otrzymuje wygodne środowisko pracy z możliwością zastawiania pułapek *breakpoint* z poziomu



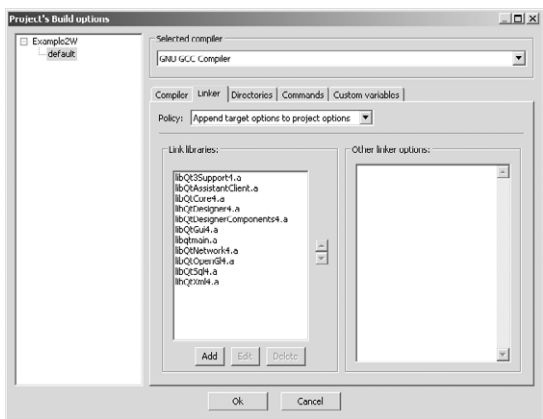
Rys. 6. Środowisko Code Blocks podczas pracy z aplikacją Example2W

interfejsu graficznego, podglądu zmiennych i różnymi rodzajami pracy krokowej. Ogólny widok edytora *Code Blocks* podczas pracy nad niniejszą aplikacją przedstawiono na **rys. 6.**

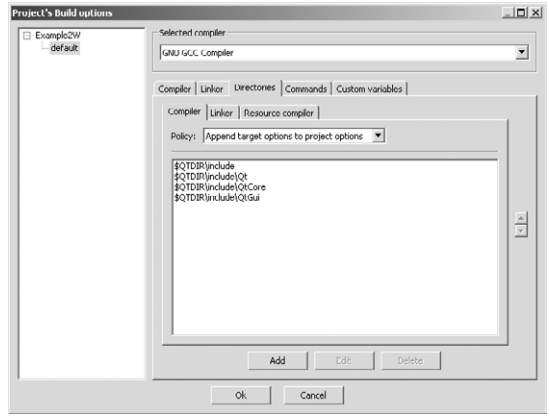
Podczas tworzenia nowego projektu (*Project–>New project...*) proponuję, zgodnie z **rys. 7**, zaznaczyć opcję „*Do not create any files*". Dzięki temu unika się tworzenia niepotrzebnych plików przykładowej aplikacji. Właściwe pliki projektu można stworzyć ręcznie. Po stworzeniu projektu należy odpowiednio skonfigurować środowisko, wybierając opcję *Project–>Build options*. Dzięki użyciu wzorca projektu Qt, w sekcji konsolidatora są już wymienione potrzebne biblioteki (**rys. 8**). Bardzo ważne jest właściwe podanie ścieżek do plików nagłówkowych biblioteki Qt. Jest to o tyle istotne, że – przynajmniej w wersji 1.0 edytora – domyślne ścieżki są błędne i nie pasują do wersji 4.0.1 biblioteki. Być może jest to relikt pozostały z wersji 3.x. Przykład właściwie podanych ścieżek przedstawiono na **rys. 9**. Można dodać ścieżki dostępu do pozostałych podkatalogów katalogu *\include*, ale można też pozostawić jedynie dwie pierwsze ścieżki i włączając pliki nagłówkowe podawać, oprócz samej nazwy pliku, ścieżkę dostępu do niego, odniesioną do jednego z katalogów wymienionych w zakładce *Directories/Compiler*. Drugie rozwiązanie jest bezpieczniejsze i takie też zostało wybrane.



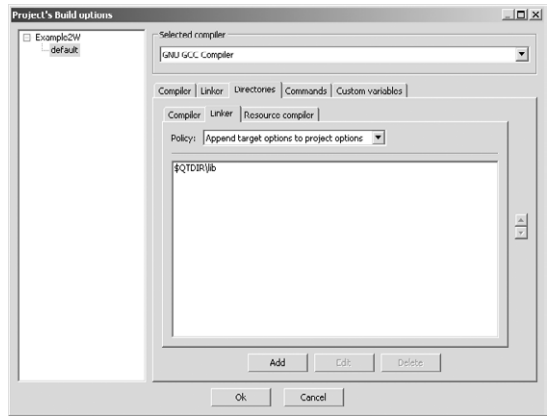
Rys. 7. Nowy projekt Code Clocks



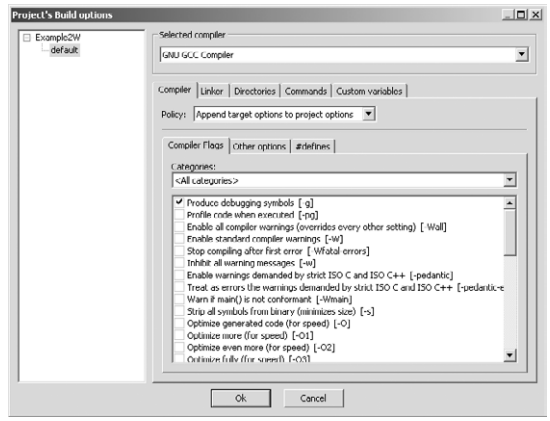
Rys. 8. Opcje projektu – konsolidator



Rys. 9. Opcje projektu – pliki nagłówkowe



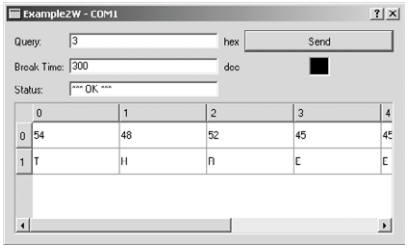
Rys. 10. Opcje projektu – biblioteki



Rys. 11. Opcje projektu – kompilator

Podobnie jak w przypadku nazw bibliotek, ścieżka do katalogu je zawierającego powinna być poprawnie ustalona, co jest zapewnione dzięki wykorzystaniu wzorca projektu Qt (**rys. 10**). Ustawienia opcji kompilatora (**rys. 11**) zależą od tego, co chcemy otrzymać w wyniku kompilacji oraz jak proces kompilacji ma przebiegać. Jedną z najbardziej użytecznych opcji jest *–g*. Jej zaznaczenie spowoduje stworzenie wersji *Debug*, odznaczenie – wersji *Release*.

Przed przystąpieniem do kompilacji projektu za pomocą IDE Code Blocks należy za pomocą programu narzędziowego *moc.exe* stworzyć, na podstawie pliku nagłówkowego formy *Example2WFrm.h*, specjalny plik *moc_Example2WFrm.cpp*. Plik ten powinien być umieszczony w katalogu projektu i dodany do projektu w środowisku *Code Blocks*. Utworzenia pliku *moc_Example2WFrm.cpp* dokujemy wydając następujące polecenie:



Rys. 12. Aplikacja testowa działająca w systemie Windows (styl Windows)

List. 24. Plik Example2W.pro

```
#############################################################################
# Automatically generated by qmake (2.00a) Cz 18. maj 23:36:02 2006
#############################################################################

TEMPLATE = app
TARGET +=
DEPENDPATH += .
INCLUDEPATH += .

# Input
HEADERS += CommInterface.h Example2WFrm.h
FORMS += Example2WFrm.ui
SOURCES += CommInterface.cpp main.cpp Example2WFrm.cpp
DEFINES -= UNICODE
```

moc –o moc_Example2WFrm.cpp Example2WFrm.h

Działanie takie nie było potrzebne podczas kompilacji programu w linii poleceń, gdyż program qmake sam umieszczał w plikach *Makefile* stosowne wywołania programu moc. Program ten zajmuje się obsługą rozszerzeń języka C++ właściwych dla biblioteki Qt, m.in. mechanizmu sygnałów i slotów. Powinien on być używany dla każdego pliku nagłówkowego zawierającego deklarację klasy, która zawiera w sobie wywołanie makra *Q_OBJECT*. Pliki wygenerowane przez program moc muszą być skompilowane i skonsolidowane razem z innymi plikami projektu.

Po wygenerowaniu pliku *moc_Example2WFrm.cpp* można skompilować i uruchomić aplikację (F9). Jeśli zaznaczono opcję *–g*, to program jest gotowy do uruchomienia w trybie debuggowania – wystarczy ustawić pułapki w wybranych miejscach programu i wcisnąć F8. Aplikację testową działającą w systemie Windows przedstawiono na **rys. 12**.

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

180x128
1/2

180x23

Programowanie portu szeregowego w systemach operacyjnych Linux i Windows, część 7

Umiejętność programowej obsługi interfejsu RS232 od strony komputera PC jest dziś istotnym elementem elektronicznego rzemiosła. W niniejszym kursie 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.

Instalacja biblioteki Qt w systemie Linux (X11)

Instalacja pakietu Qt 4.0.1 w systemie Linux przebiega bardzo podobnie do instalacji w systemie Windows, choć może być nieco bardziej skomplikowana. W celu jej przeprowadzenia należy ściągnąć plik *qt–x11–opensource–src–4.0.1.tar.gz* spod adresu [1]. Przykładowy proces instalacji przedstawiono dla systemu Mandrake 10.1, ale oczywiście jest ona taka sama dla innych dystrybucji systemu Linux. Ewentualne różnice są kosmetyczne i związane ze specyfiką konkretnej dystrybucji.

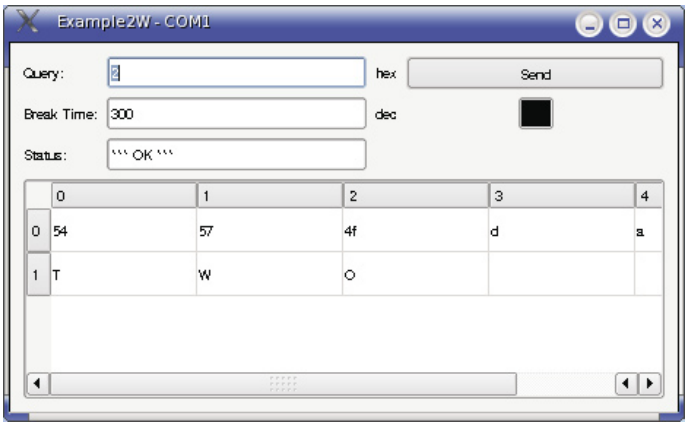
Kolejne czynności związane z instalacją są następujące:

1. Sprawdzić, czy w katalogu /usr/X11R6/include/X11 (lub analogicznym, zależnie od dystrybucji systemu Linux) znajdują się pliki o nazwach *X*.h* (np. *Xlib.h*, *Xatom.h*). Jeśli nie, to należy zainstalować którykolwiek z pakietów RPM, jaki zawiera te pliki. Ja wykorzystałem pakiet *xorg–x11–devel–6.8.2–9tr.i586.rpm*, ale nadaje się do tego dowolny pakiet, który w nazwie zawiera człon *x11–devel–*. Jeśli system, którym dysponujemy nie używa serwera *X.Org* lecz *XFree86*, należy zainstalować pakiet odpowiedni dla tego serwera.
2. Skopiować plik *qt–x11–opensource–src–4.0.1.tar.gz* do katalogu domowego lub innego, w którym chcemy mieć umieszczoną bibliotekę Qt. W moim przypadku był to katalog /home/arek i takiego katalogu dotyczy niniejszy opis przykładowej instalacji.
3. Rozpakować archiwum wpisując:
   $ tar –xvzf qt–x11–opensource–src–4.0.1.tar.gz
   W efekcie powstanie katalog o nazwie qt–x11–opensource–src–4.0.1.
4. Przejść do utworzonego katalogu.
5. Jako *root* stworzyć dowiązanie symboliczne do tego katalogu, wpisując:
   $ ln –s /home/arek/qt–x11–opensource–src–4.0.1 /usr/local/qt
   Dzięki temu możliwe jest zainstalowanie kilku wersji biblioteki Qt. Poprzez odpowiednie manipulowanie tym dowiązaniem, do każdej z nich można się odwoływać poprzez /usr/local/qt.
6. Nadać następującym zmiennym środowiskowym nazwy:
   QTDIR=/usr/local/qt
   PATH=$QTDIR/bin:$PATH
   Zapis powyższy należy umieścić w pliku konfiguracyjnym używanej powłoki. Dla powłoki *bash* jest to plik *.bash_profile*. Aby zmiana przyniosła skutek, należy się wylogować i ponownie zalogować. Można też ustalić wartości wymienionych zmiennych na czas trwania bieżącej sesji wpisując:
   $ export QTDIR=/usr/local/qt
   $ export PATH=$QTDIR/bin:$PATH
7. Warto poinformować skrypt konfiguracyjny o tym, w jakim katalogu ma się znajdować biblioteka Qt. W niniejszym przykładowym przypadku można tego dokonać wpisując:
   $ configure – prefix /home/arek/qt–x11–opensource–src–4.0.1
   Jeśli tego nie zrobimy, jako katalog w którym znajduje się biblioteka rozumiany będzie domyślny katalog /usr/local/Trolltech/Qt–4.0.1.
8. Uruchomić skrypt konfiguracyjny wpisując:
   $ ./configure
   a następnie „yes” w celu zaakceptowania licencji.
9. Po skonfigurowaniu należy skompilować bibliotekę wpisując:
   $ gmake
   lub
   $ make
   Podobnie jak w przypadku systemu Windows, proces kompilacji jest bardzo długi – może dochodzić do kilku godzin.
10. Po skompilowaniu biblioteki Qt należy dodać następujący wiersz do pliku */etc/ld.so.conf*:
    /usr/local/qt/lib
    a następnie zalogować się jako *root* i wpisać:
    $ ldconfig
    Dzięki temu konsolidator (linker) będzie miał informację o katalogu w jakim umieszczone są biblioteki pakietu Qt.
11. Jeśli jako katalog instalacji pozostawiono katalog domyślny (nie skorzystano z opcji *–prefix* skryptu konfiguracyjnego *configure*), to należy utworzyć dodatkowe dowiązanie symboliczne:
    $ ln –s /home/arek/qt–x11–opensource–src–4.0.1 /usr/local/Trolltech/Qt–4.0.1

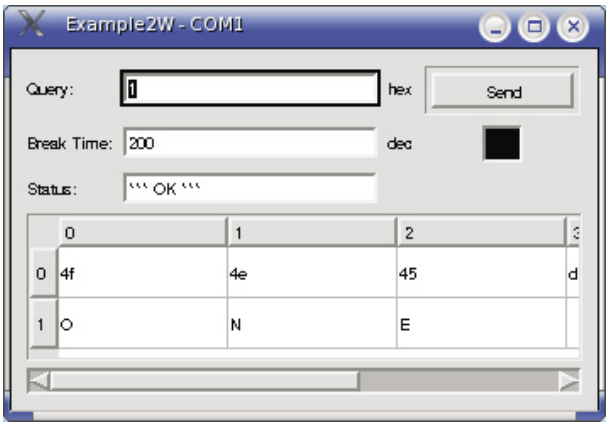
Przy wykorzystaniu opisanego sposobu instalacji biblioteki Qt nie ma potrzeby wykonywania polecenia *make install*.

Kompilacja aplikacji testowej w systemie Linux

Dzięki dobrej jakości biblioteki Qt, przeniesienie programu stworzonego w systemie Windows do systemu Linux jest niezwykle proste. Wystarczy skopiować pliki źródłowe do dowolnego katalogu (np. /home/arek/Example2W w przypadku aplikacji *Example2W*) i znajdując się w nim wpisać:



Rys. 13. Aplikacja testowa działająca w systemie Linux (styl Plastique)



Rys. 14. Aplikacja testowa działająca w systemie Linux (styl Motif)

$ qmake –project (opcjonalnie)
$ qmake
$ gmake (lub make)

Jeśli podczas instalacji nie popełniono żadnego błędu, w wyniku otrzymamy plik wykonywalny *Example2W*. Aplikację uruchamiany wpisując:

$ ./Example2W

i możemy cieszyć oczy widokiem podobnym do tego, który przedstawiono na **rys. 13** i **14**.

Podsumowanie

Wiemy już jak stworzyć prostą aplikację przenośną, wykorzystującą łącze RS232. W następnej części kursu zostanie zaprezentowany przykładowy program woltomierza cyfrowego działający w systemach Windows i Linux. Program o tyle ciekawy, że zawierający pewne elementy wielowątkowości, bez których nie może się obyć żadna nietrywialna aplikacja komunikująca się z urządzeniami zewnętrznymi przez interfejs RS232.

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

Linki internetowe:
[1] *ftp://ftp.trolltech.com/qt/source/*

R E K L A M A

180x128
1/2

Programowanie portu szeregowego w systemach operacyjnych Linux i Windows, część 8

Umiejętność programowej obsługi interfejsu RS232 od strony komputera PC jest dziś istotnym elementem elektronicznego rzemiosła. W niniejszym kursie 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.

W poprzednich częściach kursu poznaliśmy, jak stworzyć przenośną aplikację wykorzystującą interfejs RS232 z użyciem pakietu Qt firmy Trolltech. Poniżej zostanie zaprezentowany nieco bardziej złożony przykład. Jest nim woltomierz odczytywany przez łącze RS232.

Woltomierz – część sprzętowa

Do budowy woltomierza wykorzystano ten sam zestaw testowy, który posłużył do testów poprzednio opisanych aplikacji – został on opisany w części 2. kursu. Oprogramowanie części sprzętowej woltomierza stworzono wykorzystując kompilator AVR–GCC, ten sam, który wykorzystano przy implementacji opisanej wcześniej aplikacji testowej. Działanie oprogramowania mikrokontrolera polega na skonfigurowaniu modułu UART i przetwornika A/C, a następnie na oczekiwaniu na odpowiednie ramki zapytania. W odpowiedzi na poprawną ramkę przesyłana jest informacja o napięciu panującym na wejściu ADC0 przetwornika. Jak widać, układ PC – mikrokontroler jest typowym układem *Master – Slave*. W celu wymiany danych pomiędzy mikrokontrolerem a komputerem PC ustalono prosty protokół komunikacyjny. Format ramki zapytania (*Query Frame*) wysyłanej przez komputer jest następujący:

0x41 0x3F 0x80

Pierwszy bajt jest umownie wybranym bajtem nagłówkowym ramki (*QF_HEADER*). Określa on początek ramki i służy do synchronizacji odbiornika (w tym przypadku mikrokontrolera) z nadajnikiem (w tym przypadku komputerem PC). Drugi bajt (*QF_COMMAND*) to kod ASCII znaku zapytania ‚?' – łatwo odgadnąć, że symbolizuje on pytanie. Ostatni bajt jest sumą kontrolną i pozwala odbiornikowi dokonać kontroli poprawności ramki. Jego wartość jest dopełnieniem do dwóch sumy arytmetycznej pozostałych bajtów ramki zapytania, dzięki czemu najmniej znaczący bajt sumy arytmetycznej wszystkich bajtów poprawnej ramki jest równy zero:

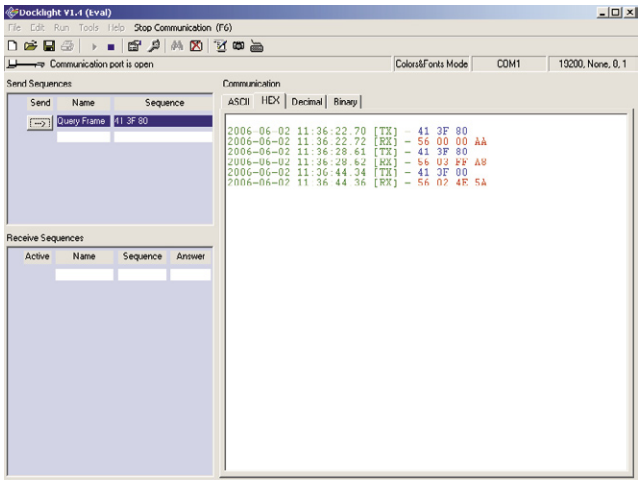
41+3F+80=100 (HEX)

Jak widać, zaproponowano format ramki nieco na wyrost w stosunku do funkcji jaką pełni. Przesyłany jest jeden rodzaj informacji, co sprawia, że suma kontrolna ma zawsze tę samą wartość. Pokazuje on jednak podstawowe cechy, jakie powinna mieć poprawna ramka zapytania: nagłówek służący synchronizacji, dane, suma kontrolna, a w przypadku ramek o zmiennych długościach – przesyłana powinna być długość ramki lub liczba bajtów danych, ewentualnie bajt terminujący (określający koniec pakietu danych). W profesjonalnych systemach stosuje się znacznie lepsze sposoby obliczania sum kontrolnych (jak choćby CRC, czy kod Reeda–Solomona), jednak w niniejszym, prostym przykładzie suma arytmetyczna jest wystarczająca.

Odpowiedzią na ramkę zapytania jest ramka odpowiedzi, której format jest następujący:

0x56 ByteH ByteL CKS

Pierwszy bajt to kod ASCII znaku ‚V', symbolizującego napięcie elektryczne (*Volt*). Jest to nagłówek ramki. Następne dwa bajty są wynikiem konwersji analogowo–cyfrowej i, ze względu na rozdzielczość przetwornika mikrokontrolera ATmega8, istotnych jest 10 ich najmniej znaczących bitów. Bajt *CKS* jest oczywiście sumą kontrolną, obliczaną tak samo jak w przypadku ramki zapytania. Pozwala on odbiornikowi (PC) stwierdzić, czy ramka jest poprawna, czy nie.



Rys. 15. Test części sprzętowej woltomierza

List. 25. Konfiguracja przetwornika A/C mikrokontrolera

```
//*******************************
// ADC Init
//*******************************
void Init_ADC(void)
{
  //ADC channel 0, internal vref=2.56V
  ADMUX=0xC0;

  //single conversion
  ADCSR=0x97;

  //start first dummy conversion -
  //- here ADC is adjusting himself
  ADCSR|=0x40;
  while((ADCSR & 0x40)!=0);

  Waitms(10);
}
```

List. 26. Konfiguracja USART i funkcje obsługi łącza RS232

```
//*******************************
// RS232 Init
//*******************************
void Init_UART(void)
{
  //enable transmitter
  UCSRB|=(1<<TXEN);

  //enable receiver
  UCSRB|=(1<<RXEN);

  //normal speed
  UCSRA&=~(1<<U2X);

  //19200 baud, err=0.0%
  UBRRH=0;
  UBRRL=5;
  UCSRB&=~(1<<UCSZ2);

  //8bit, no parity, one stop bit
  UCSRC=(1<<URSEL)|(3<<UCSZ0);
}

//*******************************
// RS232 Send one byte
//*******************************
void SendByte(unsigned char byte)
{
  UCSRA|=(1<<TXC);
  UDR=byte;
  while((UCSRA&(1<<TXC))==0);
}

//*******************************
// RS232 Send string
//*******************************
void SendString(char *pString)
{
  char a;
  unsigned char i=0;
  while(a=PRG_RDB(&pString[i++]))
     SendByte(a);
}
```

List. 27. Pętla główna programu mikrokontrolera

```
while(1)
{
    //Waiting for query frame header - QF_HEADER
    UCSRA&=~(1<<RXC);
    while((UCSRA&(1<<RXC))==0);

    QFrame[0]=UDR;

    if(QFrame[0]==QF_HEADER)
    {
        //Waiting for rest 2 bytes of query frame with timeout
        unsigned char byte_counter;
        unsigned int timeout_counter;

        timeout_counter=0;
        for(byte_counter=1;byte_counter<3;byte_counter++)
        {
            //Waiting for 1 byte
            UCSRA&=~(1<<RXC);
            while((UCSRA&(1<<RXC))==0)
                if(++timeout_counter>=TIMEOUT)
                    break;

            QFrame[byte_counter]=UDR;

            //If timeout occured - break for loop
            if(timeout_counter>=TIMEOUT)
                break;

        }

        //If timeout did not occured - check checksum and
  //if OK then check query and send response
        if(timeout_counter<TIMEOUT)
        {
            //Checking checksum
            unsigned char summa=0;
            for(byte_counter=0;byte_counter<3;byte_counter++)
                summa+=QFrame[byte_counter];

            if((summa & 0xFF)==0)
            {
                unsigned char adc_valL,adc_valH;

                //Checking query command
                if(QFrame[1]==QF_COMMAND)
                {
                    //Measuring voltage
                    RED_ON;
                    ADCSR|=0x40;     //start conversion
                    while((ADCSR & 0x40)!=0);
                    adc_valL=ADCL;     //read LSB first
                    adc_valH=(ADCH & 0x03);       //read MSB (2 bits)

                    //Sending response
                    SendByte('V');
                    SendByte(adc_valH);
                    SendByte(adc_valL);
                    SendByte(0x100-'V'-adc_valH-adc_valL);

                    Waitms(10);
                    RED_OFF;
                }
            }
        }
    }
}
```

Przykładowy test części sprzętowej z użyciem terminala RS232 *Docklight* przedstawiono na **rys. 15**. Widoczne są trzy testy. Pierwszy został dokonany w chwili, gdy napięcie na wejściu ADC0 wynosiło zero, drugi – gdy było ono maksymalne. Trzeci test przeprowadzono dla dowolnej wartości pośredniej (w tym przypadku równej 1,48 V, przy napięciu referencyjnym, odpowiadającym pełnej skali, równym 2,56 V). We wszystkich przypadkach najmniej znaczący bajt sumy arytmetycznej wszystkich bajtów ramki jest równy 0 (np. 56+02+4E+5A=100).

Na **list. 25** przedstawiono konfigurację przetwornika analogowo–cyfrowego mikrokontrolera. Został on skonfigurowany do pracy z wewnętrznym napięciem odniesienia o wartości nominalnej 2,56 V i do pracy w trybie pojedynczych konwersji. Pierwsza konwersja, jaka jest wywoływana w funkcji z list. 25, ma na celu ustawienie przetwornika – konwersja ta trwa znacznie dłużej niż późniejsze przetwarzanie. Na **list. 26** przedstawiono konfigurację oraz funkcje obsługi interfejsu RS232 (wysyłanie jednego bajtu i wysyłanie łańcucha znaków). USART pracuje z ramką 8N1 przy szybkości 19200 b/s.

Pętlę główną oprogramowania mikrokontrolera przedstawiono na **list. 27**. Na początku procesor oczekuje na bajt nagłówkowy ramki zapytania. Oczekiwanie to nie jest objęte żadnym czasem przeterminowania. Odebranie bajtu innego niż *QF_HEADER* (o wartości 0x41) powoduje jego zignorowanie i kontynuowanie oczekiwania. Jeśli odebrano właściwy bajt nagłówkowy, mikrokontroler przechodzi do pętli, w której oczekuje na pozostałe dwa bajty ramki zapytania. Oczekiwanie to jest objęte odpowiednim czasem przeterminowania (*timeout*), wynoszącym około 4–krotnemu czasowi transmisji dwóch bajtów. Jeśli w tym czasie nie odebrano 2 bajtów, ramka jest ignorowana i program przechodzi do oczekiwania na bajt nagłówkowy. Jeśli odebrano, to sprawdzana jest wartość drugiego bajtu ramki (*QF_COMMAND*). Jeśli jego wartość jest poprawna, to następnie jest obliczana suma arytmetyczna wszystkich bajtów ramki i, jeśli jest ona poprawna, mikrokontroler dokonuje konwersji analogowo–cyfrowej. Odczytana z przetwornika wartość napięcia (w formie dwóch bajtów) jest wysyłana do komputera PC w formie ramki odpowiedzi opisanej wyżej. Na czas konwersji i transmisji tej ramki oraz przez ok. 10 następnych milisekund, zapalana

Rys. 16. Aplikacja woltomierza działająca w systemie Windows

Rys. 17. Aplikacja woltomierza działająca w systemie Linux

jest dioda czerwona D1 sygnalizując aktywność woltomierza.

Woltomierz – program na komputer PC

Woltomierz działający w systemie Windows przedstawiono na **rys. 16**, zaś na **rys. 17** pokazano widok okna programu pracującego w systemie Linux. Aplikacja *VoltmeterW* posiada pole wyniku (pokazujące napięcie wejściowe z rozdzielczością 0,01 V), pole stanu (zawierające informację o poprawności otrzymanej odpowiedzi lub jej braku), pole pozwalające zatrzymać i wznowić odświeżanie wyniku oraz pole tekstowe, służące do kalibracji woltomierza. Kalibracja może okazać się konieczna, ze względu na spory rozrzut napięć wewnętrznego źródła napięcia odniesienia mikrokontrolera ATmega8. Zgodnie z dokumentacja mikrokontrolera, wewnętrzne napięcie odniesienie może zawierać się w przedziale 2,3...2,7 V. Maksymalna odchyłka od wartości nominalnej (2,56 V) wynosi 0,36 V, co oczywiście znacznie przekracza rozdzielczość niniejszego woltomierza.

Kod aplikacji woltomierza składa się z trzech klas:
- *CCommInterface*
- *SamplingThread*
- *VoltmeterWFrm*

Pierwsza z nich jest nam dobrze znana z poprzednich części kursu. Klasa *VoltmeterWFrm* jest klasą głównej (i jedynej) formy aplikacji. Została ona stworzona podobnie do klasy formy aplikacji testowej *Example2W*, opisanej w poprzednich częściach kursu. Podobnie jak wtedy, tak i w tym przypadku, do budowy interfejsu graficznego wykorzystano program *Qt Designer*. Deklarację klasy *VoltmeterWFrm* przedstawiono na **list. 28**. Jak widać, jednym z jej pól jest obiekt *stSampling* klasy *SamplingThread*. Klasa ta implementuje wątek odpytujący część sprzętową. Zastosowanie w tym celu osobnego wątku, zamiast np. licznika klasy *QTimer*, ma ogromne zalety. Tworząc nowy wątek otrzymujemy możliwość przeprowadzania przeróżnych operacji, które wykonają się w tle głównego wątku, dzięki czemu nie będą one niepotrzebnie zabierać jego czasu. Wątek *SamplingThread* zajmuje się odpytywaniem sprzętu (działa jako typowy *Worker Thread*) i analizowaniem odpowiedzi, a do wątku głównego przesyła gotowe efekty tej analizy, które są w tym wątku jedynie wizualizowane.

List. 28. Deklaracja klasy VoltmeterWFrm
```
#include „ui_VoltmeterWFrm.h”
#include „SamplingThread.h”

class VoltmeterWFrm : public QDialog
{
    Q_OBJECT

public:
    VoltmeterWFrm(QDialog *parent = 0);
    ~VoltmeterWFrm();

private slots:
    void UpdateResult(ulong data, QString
status);
    void ClickedCheckBoxSampling(int state);

private:
    SamplingThread stSampling;
    Ui::frmVoltmeterW ui;
};
```

Plik nagłówkowy klasy wątku przedstawiono na **list. 29**. Obiekt interfejsu szeregowego jest polem tej właśnie klasy. Port jest otwierany w konstruktorze klasy *VoltmeterWFrm*, w którym wątek jest uruchamiany (**list. 30**). Do komunikacji pomiędzy wątkiem próbkującym a wątkiem głównym wykorzystano mechanizm sygnałów i slotów. Klasa *VoltmeterWFrm* posiada następujący slot:

List. 29. Deklaracja klasy wątku SamplingThread
```
#ifndef SamplingThreadH
#define SamplingThreadH

#include „CommInterface.h”

#include <QtCore/QMutex>
#include <QtCore/QThread>
#include <QtCore/QWaitCondition>

class SamplingThread : public QThread
{
    Q_OBJECT

public:
    QMutex mutex;
    CCommInterface Comm;
    SamplingThread(QObject *parent = 0);
    ~SamplingThread();
    void Suspend();
    void Resume();

signals:
    void NewDataReceived(ulong data, QString status);

protected:
    void run();

private:
    bool abort;
    bool suspend;
    QWaitCondition condition;
    void CheckResponse(void);
};


#endif
```

void UpdateResult(ulong data, QString status);

Funkcji tej przekazywana jest wartość napięcia (10 najmniej znaczących bitów argumentu *data*) oraz napis, jaki pojawić się ma w polu stanu. Slot jest wywoływany wtedy, gdy wątek próbkujący wyemituje następujący sygnał:

void NewDataReceived(ulong data, QString status);

Sygnał ten jest zadeklarowany w klasie wątku (list. 29). Oczywiście slot jest wywoływany z tymi samymi argumentami, co sygnał z nim połączony. Połączenie wymienionych slotów i sygnałów jest dokonywane w konstruktorze klasy *VoltmeterWFrm* (list. 30). Implementację slotu *UpdateResult()* przedstawiono na **list. 31**. Jak widać, jego działanie polega na pobraniu wartości napięcia kalibrującego (odpowiadającego napięciu pełnej skali) i przetworzeniu 10–bitowej liczby odczytanej z przetwornika A/C na liczbę ułamkową oraz jej wyświetlenie w polu wyniku. Informacja z argumentu *status* trafia do pola stanu. Wartość napięcia jest zakodowana w 10–bitowej zmiennej *data* (zakres wartości 0...1023).

Ważnymi polami klasy *SamplingThread* są pola *abort* i *suspend*, oba typu *bool* (list. 29). Nadanie wartości *true* polu *abort* powoduje zakończenie działania wątku, zaś polu *suspend* – zawieszenie jego działania. Do realizacji zawieszania służy

List. 30. Konstruktor klasy VoltmeterWFrm

```
VoltmeterWFrm::VoltmeterWFrm(QDialog *parent)
    : QDialog(parent)
{
    ui.setupUi(this);

    //Set palette (text color) for result display and status label
    QPalette pal=ui.lbResult->palette();
    pal.setColor(QPalette::Foreground,QColor(tr(„lightgreen”)));
    ui.lbResult->setPalette(pal);

    pal=ui.lbStatus->palette();
    pal.setColor(QPalette::Foreground,QColor(tr(„white”)));
    ui.lbStatus->setPalette(pal);

    //Set text codec for tr() function
    QTextCodec::setCodecForTr(QTextCodec::codecForName(„ISO8859-2”));

    //Connect signals and slots
    connect(&stSampling, SIGNAL(NewDataReceived(ulong, QString)),
        this, SLOT(UpdateResult(ulong, QString)));
    connect(ui.cbSampling, SIGNAL(stateChanged(int)),
        this, SLOT(ClickedCheckBoxSampling(int)));

    //Initial information
    UpdateResult(0,”No response”);

    //Open port
    string port=”COM1”;
    if(stSampling.Comm.Open(port,19200)==false)
    {
        QString portname=port.c_str();
        QMessageBox::critical(this,tr(„Error”),tr(„Port „)+portname+
            tr(„ opening error. Application will terminate.”));
        this->reject();
    }

    //Start the thread
    if (!stSampling.isRunning())
        stSampling.start(QThread::TimeCriticalPriority);
}
```

List. 31. Odświeżanie wartości napięcia i stanu

```
void VoltmeterWFrm::UpdateResult(ulong data, QString status)
{
    //Format result
    double fUmax=ui.leCalibration->text().toDouble();
    if((fUmax<0) || (fUmax>5))
            fUmax=0;

    double fResult=(fUmax*data)/1024;

    //Dislpay result and status
    ui.lbResult->setText(QString::number(fResult,’f’,2)+” V”);
    ui.lbStatus->setText(status);
}
```

List. 32. Funkcja run() wątku SamplingThread

```
void SamplingThread::run()
{
    forever
    {
        //Repeat sampling every 200ms (150+50)
        msleep(150);

        //Send query frame
        mutex.lock();
        unsigned char QueryFrame[3];
        QueryFrame[0]=0x41;     //’A’
        QueryFrame[1]=0x3F;     //’?’
        QueryFrame[2]=0x80;     //checksum
        Comm.Write(QueryFrame,3);
        mutex.unlock();

        //Break Time
        msleep(50);

        //Check response
        mutex.lock();
        this->CheckResponse();
        mutex.unlock();

        if(abort)
            return;

        mutex.lock();
        if(suspend)
            condition.wait(&mutex);
        mutex.unlock();
    }
}
```

pole *condition* klasy *QWaitCondition* (szczegóły opisane zostaną dalej).

Na **list. 32** przedstawiono funkcję *run()* wątku próbkującego. Do tej właśnie funkcji przekazywane jest sterowanie po uruchomieniu wątku za pomocą metody *start()*. Działanie tej funkcji polega na cyklicznym wysyłaniu zapytania, odczekaniu czasu *break time* (ok. 50 ms), analizie odpowiedzi za pomocą funkcji *CheckResponse()* oraz sprawdzeniu warunków zakończenia i zawieszenia wątku. Implementację funkcji *CheckResponse()* przedstawiono na **list. 33**. Jej działanie jest podobne do działania fragmentu aplikacji testowej, opisanej w poprzednich częściach kursu. Funkcja sprawdza czy nie wystąpiły błędy i czy odpowiedź w ogóle nadeszła. W przypadku braku odpowiedzi lub błędu emituje ona sygnał *NewDataReceived()*, podając jako argument *data* wartość 0, a jako *status* stosowną informację. Jeśli nadeszła poprawna odpowiedź, emitowany jest sygnał zawierający 10–bitową informację o napięciu oraz stan „OK”.

Wróćmy na chwilę do funkcji *run()*. Niektóre jej fragmenty są ujęte w następującą klamrę:

```
mutex.lock();
...
mutex.unlock();
```

Pole klasy *SamplingThread* o nazwie *mutex* (obiekt klasy *QMutex*) to semafor binarny wzajemnego wykluczania (*MUTual EXclusion*). Jego działanie polega na tym, że uniemożliwia on dwóm (lub więcej) wątkom na jednoczesne operowanie na współdzielonych zasobach. Gdyby nie został zastosowany, to jeden wątek mógłby zmienić warunki pracy drugiego wątku w sposób dla niego zupełnie nieprzewidywalny. Byłoby tak dlatego, że system operacyjny może przełączyć wątki na przykład w chwili, gdy jeden z nich (*wątek A*) jest w trakcie wykonywania pewnego ciągu instrukcji na pewnych danych. Dostęp do tych danych ma także drugi wątek (*wątek B*). *Wątek B* może zmienić wartości zmiennych, które są wynikami operacji wykonanych przed przełączeniem przez *wątek A*. Po ponownym przekazaniu sterowania *wątkowi A* próbuje on skorzystać z tych zmiennych, ale ich wartości są już inne niż te, których się spodziewa. Powoduje to oczywiście nieprzewidywalne zachowanie się programu. Gdy *mutex* jest opuszczony (*lock*), to wątek próbujący dostać się do chronionej przez niego sekcji krytycznej zostaje zawieszony. Gdy wątek, który opuścił semafor, podniesie go (*unlock*), wątek zawieszony wznawia swoje działanie i wchodzi do sekcji krytycznej opuszczając jednocześnie semafor. Operacje sprawdzenia stanu semafora i jego opuszczenia (jeśli to możliwe) są operacjami atomowymi.

W przypadku aplikacji woltomierza, konkurencyjnymi wątkami są: główny wątek aplikacji i wątek

List. 33. Funkcja analizująca odpowiedź części sprzętowej

```
void SamplingThread::CheckResponse(void)
{
    unsigned long Errors,InQue,BytesRead;
    unsigned char *Input;

    Comm.CheckCommInput(&Errors,&InQue);
    if(Errors!=0)
    {
        emit NewDataReceived(0,"UART error");
    }
    else
    {
        if(InQue==0)
        {
            emit NewDataReceived(0,"No response");
        }
        else
        {
            Input=new unsigned char[InQue];
            if(Comm.Read(Input,&BytesRead,InQue)==false)
            {
                emit NewDataReceived(0,"Read error");
                delete [] Input;
                return;
            }

            //Checking response
            if((Input[0]!='V') || (BytesRead!=4))
            {
                emit NewDataReceived(0,"Response error");
                delete [] Input;
                return;
            }

            //Checking checksum
            unsigned char summa=0;
            for(unsigned long j=0;j<BytesRead;j++)
                summa+=Input[j];

            //Checksum OK
            if((summa & 0xFF)==0)
            {
                //Show result
                emit NewDataReceived(256*Input[1]+Input[2],"OK");
            }
            else
                emit NewDataReceived(0,"Checksum error");

            delete [] Input;
        }
    }
}
```

List. 34. Funkcje zawieszenia i wznowienia wątku próbkującego

```
void SamplingThread::Suspend()
{
    suspend=true;
}

void SamplingThread::Resume()
{
    mutex.lock();
    suspend=false;
    condition.wakeOne();
    mutex.unlock();
}
```

List. 35. Slot ClickedCheckBoxSampling()

```
void VoltmeterWFrm::ClickedCheckBoxSampling(int state)
{
    if(0==state)
        stSampling.Suspend();
    else
        stSampling.Resume();
}
```

List. 36. Konstruktor i destruktor klasy wątku SamplingThread

```
SamplingThread::SamplingThread(QObject *parent)
    : QThread(parent)
{
    abort = false;
    suspend=false;
}

SamplingThread::~SamplingThread()
{
    mutex.lock();
    abort=true;
    suspend=false;
    condition.wakeOne();
    mutex.unlock();

    //Wait until thread ends run()
    wait();
}
```

List. 37. Destruktor klasy VoltmeterWFr

```
VoltmeterWFrm::~VoltmeterWFrm()
{
    stSampling.mutex.lock();
    stSampling.Comm.Close();
    stSampling.mutex.unlock();
}
```

SamplingThread. Główny wątek aplikacji ma bezpośredni dostęp do publicznych pól i metod wątku *SamplingThread*, a co za tym idzie – pośredni dostęp do składowych prywatnych. Należy pamiętać, że wywołanie np. konstruktora czy destruktora klasy *SamplingThread* odbywa się w przestrzeni wątku głównego (bo to właśnie on wywołuje konstruktor i destruktor). Z tego powodu, klamrą *mutex.lock()...mutex.unlock()* należy objąć wszystkie te odwołania do składowych klasy *SamplingThread*, w których zależy nam na ciągłości wykonywanych operacji. Zasada ta dotyczy zarówno implementacji klasy *SamplingThread*, jak i klasy *VoltmeterWFrm*, znajdującej się w przestrzeni wątku głównego.

Jeśli pole *abort* ma wartość *true*, funkcja *run()* kończy swoje działanie, a co za tym idzie, kończy je także wątek próbkujący. Jeśli pole *suspend* ma wartość *true*, to wątek jest zawieszany na zmiennej warunkowej *condition*. Wywołanie funkcji

condition.wait(&mutex);

powoduje nie tylko zawieszenie działania wątku, ale także niejawne podniesienie (*unlock*) semafora *mutex*. Dzięki temu wątek główny ma dostęp do zmiennej *condition* i może wznowić działanie wątku próbkującego, gdy tylko „zechce". Jednocześnie zapewniona jest spójność operacji sprawdzenia stanu zmiennej *suspend* i zawieszenia wątku. Na **list. 34** pokazano funkcje zapewniające dostęp do pola *suspend*. Funkcje te wykorzystano w slocie obsługi zdarzenia polegającego na zmianie stanu pola typu *QCheckBox*, pozwalającego zatrzymać i wznowić odświeżanie wyniku (**list. 35**).

Na **list. 36** przedstawiono konstruktor i destruktor klasy wątku *SamplingThread*. Destruktor powoduje ustawienie znacznika *abort* oraz obudzenie wątku, który może być zawieszony na warunku *condition*. Dzięki wywołaniu funkcji *wait()*, istnieje pewność, że destruktor zakończy swoje działanie dopiero po zakończeniu pracy funkcji *run()*. Port szeregowy jest zamykany w destruktorze klasy *VoltmeterWFrm* (**list. 37**). Aby nie odbyło się to w trakcie korzystania z portu przez wątek próbkujący, operacja zamykania portu ujęta jest w klamrę *mutex.lock()...mutex.unlock()*.

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