init, read and write for linux serial device with C

At the moment i'm working on a new project where i want to make a connection with an FTDI which is connected to my debian machine. I am intending to write the code with C, not C++. Here lies my problem. All the examples i find are incomplete or are made for a c++ compliler in stead of the GCC compiler.

The goal is to talk to my microcontroller which is connected to the FTDI. For debugging i want to start building a linux application which is able to:

- initialize a serial connection on startup with ttyUSB1
- send a character string
- display character strings when they are received by the pc
- save the communication to a .txt file

Is there any example code or tutorial to make this?

If i succeed i will defenetly place the code here so that new viewers can use it to!

Edit:

Like i said i would post the code if i had it, and this is what worked for me:

```
    stdio.c_oflag=0;
    stdio.c_cflag=0;
    stdio.c_lflag=0;
    stdio.c_cc[VMIN]=1;
    stdio.c_cc[VTIME]=0;
    tcsetattr(STDOUT_FILENO,TCSANOW,&stdio);
    tcsetattr(STDOUT_FILENO,TCSAFLUSH,&stdio);
    fcntl(STDIN_FILENO, F_SETFL, O_NONBLOCK);       // make the reads non-
blocking
    memset(&tio,0,sizeof(tio));
    tio.c_iflag=0;
    tio.c_oflag=0;
    tio.c_cflag=CS8|CREAD|CLOCAL;           // 8n1, see termios.h for more
information
    tio.c_lflag=0;
    tio.c_cc[VMIN]=1;
    tio.c_cc[VTIME]=5;
    if((tty_fd = open(MODEM , O_RDWR | O_NONBLOCK)) == -1){
        printf("Error while opening\n"); // Just if you want user interface
error control
        return -1;
    }
    cfsetospeed(&tio,BAUDRATE);
    cfsetispeed(&tio,BAUDRATE);            // baudrate is declarated above
    tcsetattr(tty_fd,TCSANOW,&tio);
    while (c!='q'){
        if (read(tty_fd,&c,1)>0){
            write(STDOUT_FILENO,&c,1); // if new data is available on the
serial port, print it out
            printf("\n");
        }
        if (read(STDIN_FILENO,&c,1)>0){
            write(tty_fd,&c,1);//if new data is available on the console, send
it to serial port
            printf("\n");
        }
```

Most of the code came from http://en.wikibooks.org/wiki/Serial_Programming/Serial_Linux but i also used a bit from the code posted below. **Thank you all for helping me out!**

Feel free to copy the code.

c linux serial-port tty

edited Oct 21 '13 at 19:42
dsolimano
5,095 3 20 31

asked Oct 17 '13 at 13:01
Embed
56 1 6

1 There's even a nice howto dedicated to that subject. – fvu Oct 17 '13 at 13:07

1 did you check this question stackoverflow.com/questions/2982552/… – 999k Oct 17 '13 at 13:08

What's wrong with minicom? – KBart Oct 17 '13 at 13:09

Good question, later i want to rebuild the program to log and execute commands without having me behind the pc so this concept is just the beginning. – Embed Oct 17 '13 at 13:11

1 I see the point. Well, you can at least look at minicom source code (alioth.debian.org/frs/…). It's open source and written in C, also heavily used for many years, so I'm sure there is a lot to reuse/learn from. – KBart Oct 17 '13 at 13:19

show **7** more comments

2 Answers

Handling with serial ports (for linux OS) :
- To open communication, you will need a descriptor which will be the handle for your serial port.
- Set the flags to control how the comunication will be.
- Write the command to this Handle (make sure you're formatting the input correctly).
- Get the answer. (make sure you're to read the amount of information you want)
- Close the handle. It will seem like this:

```
int fd; // file descriptor
int flags; // communication flags
int rsl_len; // result size
char message; // message to send
char result; // result to read

flags = O_RDWR | O_NOCTTY; // Read and write, and make the job control for
portability
if ( (fd = open("/dev/ttyUSB1", flags)) == -1 ){
  printf("Error while opening\n"); // Just if you want user interface error
control
  return -1;
}
// In this point your communication is already estabilished, lets send out
something
strcpy(message, "Hello");
if ( (rsl_len = write(fd, message, strlen(message)) < 0 ){
  printf("Error while sending message\n"); // Again just in case
  return -2;
}
if ( (rsl_len = read(fd, &result, sizeof(result)) < 0 ){
  printf("Error while reading return\n");
  return -3;
}
close(fd);
```

Note that you have to care about what to write and what to read. Some more flags can be used in case of parity control, stop bits, baud rate and more.

answered Oct 17 '13 at 14:12
rfermi
114 3

Of course this code does not handle `E_INTR` , doesn't open the file handles using `O_CLOEXEC` , doesn't handle signals... This is not how you want to tell a newbie to write good Unix code. Once you get this to a point where it's good Unix code, it will be so long for the little it does, there's a question of whether it's even worth doing any longer in plain C without losing your sanity... – Kuba Ober Oct 17 '13 at 21:46

@KubaOber At the other end of the spectrum one can say that Qt is so large and all-encompassing in what it does that there's a question of whether it's even worth using. I have seen Qt brought in as a dependency on projects which only used it for a very small portion of the functionality (for example: basic data structures,

which are far more sensibly implemented using STL) and basically the price you pay is bloat. I figure it's easier to hold on to your sanity doing it this way... It's building your car from scratch rather than being at the mercy of the manufacturer for replacement parts – Steven Lu May 14 at 2:18

I may have chosen a terrible analogy as without extraordinary resources, the car you end up with will be not fit for use. POSIX, though, is a bit less challenging, mostly because it's well documented. – Steven Lu May 14 at 2:19

@StevenLu Qt is quite usable even if you don't bring in the entire modules, but just selected parts of them. This is, generally, done automatically if you statically link the Qt library to your executable. As an extra bonus, you can certainly include only certain Qt *source* files in your project, just like Qt does when it bootstraps itself. On Windows, a completely statically compiled hello world that uses QString, QTextStream on a `stdout` from cstdio, without threading support, with no dependencies on any runtimes, is around 800kb without trying super hard. – Kuba Ober May 14 at 9:53

Yeah, but 800kB is still enormous for how little actual functionality that binary provides! – Steven Lu May 14 at 13:17

show **3** more comments

Since gcc is a C/C++ compiler, you don't need to limit yourself to pure C.

Sticking to pure C is OK if you enjoy writing lots of boilerplate code, and if you really know what you're doing. Many people use Unix APIs incorrectly, and a lot of example code out there is much too simplistic. Writing correct Unix C code is somewhat annoying, to say the least.

Personally, I'd suggest using not only C++, but also a higher-level application development framework like Qt. Qt 5 comes bundled with a QtSerialPort module that makes it easy to enumerate the serial ports, configure them, and get data into and out of them. Qt does not force you to use the gui modules, it can be a command line application, or a non-interactive server/daemon.

QtSerialPort is also usable from Qt 4, but it doesn't come bundled with Qt 4, you have to add it to your project. I suggest starting out with Qt 5, it's nicer to use with its C++11 support.

The code using Qt can be pretty simple, not much longer than your plain-English description. The below is a Qt console application using Qt 5 and C++11. It uses the `core` and `serialport` modules. It also handles the `SIGINT` signal so that the output file gets flushed before the process would terminate due to a `^C`. I'm using `QLocalSocket` in place of raw Unix API to communicate from the Unix signal handler, the functionality is the same.

Only the code within `main` is strictly required, the rest is just to make it properly wrap things up when you hit `^C`.

```
    QLocalSocket * receive = srv.nextPendingConnection();
    receive->setParent(qApp);
    qApp->connect(receive, &QLocalSocket::readyRead, &QCoreApplication::quit);
    struct sigaction sig;
    sig.sa_handler = signalHandler;
    sigemptyset(&sig.sa_mask);
    sig.sa_flags = SA_RESTART;
    return ! sigaction(SIGINT, &sig, NULL);
}

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    setupSignalHandler();

    QSerialPort port("ttyUSB1");
    QFile file("file.txt");
    QTextStream err(stderr, QIODevice::WriteOnly);
    QTextStream out(stdout, QIODevice::WriteOnly);
    if (!file.open(QIODevice::WriteOnly)) {
        err << "Couldn't open the output file" << endl;
        return 1;
    }
    if (!port.open(QIODevice::ReadWrite)) {
        err << "Couldn't open the port" << endl;
        return 2;
    }
    port.setBaudRate(9600);
    QObject::connect(&port, &QSerialPort::readyRead, [&](){
        QByteArray data = port.readAll();
        out << data;
        file.write(data);
    });
    out << "Use ^C to quit" << endl;
    return a.exec();
}
```

edited Oct 17 '13 at 22:21

answered Oct 17 '13 at 21:29

Kuba Ober
19.7k 6 17 53

Sounds promising but how do i add the Qt librarys to GCC? I just installed QT5 but still comes up with "gcc -o qtSerial qtSerial.cpp qtSerial.cpp:1:28: fatal error: QCoreApplication: No such file or directory" when compiling. – Embed Oct 18 '13 at 9:17

Well, you can't do it manually without understanding what goes on. Use QT creator to create a project, add your files to the project, add the `serialport` module to the .pro file (`QT += serialport`), and it should work. – Kuba Ober Oct 18 '13 at 12:36

I would vote your answer up if i had 15 reputation – Embed Oct 18 '13 at 14:55

required language was C and OS was unix. Downvoted as this two requirement wasn't respected. – lesto yesterday

@lesto I offered an alternative solution, and this most certainly works on any Unix system where you can compile Qt (anything worthwhile, essentially) :) – Kuba Ober yesterday

add a comment

Not the answer you're looking for? Browse other questions tagged c | linux | serial-port | tty **or ask your own question.**