RS-232 for Linux, FreeBSD and windows

Here you can find code to use the serial port.
It has been tested with GCC on Linux and Mingw-w64 on Windows.
It uses polling to receive characters from the serial port.
Interrupt/event-based is not supported.

It is licensed under the GPL version 3.

No serial port available on your computer? Use a USB to RS232 cable.

Download

This is free software, it is experimental and available under the GPL License version 3.
Despite this software is intend to be usefull, there is no warranty, use this software at your own risk.

May 31, 2019 new version:
- Added support for hardware flow control using RTS and CTS lines.

May 20, 2019 new version:
- added function to check for status of "RING"
- added 921600, 1500000, 2000000, 3000000 baud rates for windows
- extended comport range to COM32 for windows

November 22, 2017 new version:
- Bugfix: RS232_SendBuf() did not return the number of bytes on Linux.

August 5, 2017 new version:
- added a makefile for the demo's.

July 10, 2016 new version:
- added the function: RS232_GetPortnr().
- always unlock the device in case of an error.

December 19, 2015 new version:
- added the functions: RS232_flushRX(), RS232_flushTX() and RS232_flushRXTX().

January 10, 2015 new version:
- Fixed a bug that affected the parity settings.

December 6, 2014 new version:
- Make it compile on FreeBSD.
- Added the devices "/dev/cuau0", etc. needed for FreeBSD.

October 5, 2014 new version:
- Added the possibility to set the mode (databits, parity and stopbits).
- Changed function RS232_OpenComport(), from now on, an extra argument is required to set the mode.
- Fixed a bug that could set wrong baudrates on Linux systems (POSIX instead of BSD style).

Januari 31, 2014 new version:
- Fixed a bug that made it impossible to read from the serial port on Linux 64-bit systems.

December 26, 2013 new version:
- added the function RS232_IsDCDEnabled()

February 1, 2013 new version:
- added the prefix "RS232_" to all functions in order to prevent clashes with other libraries
- set the DTR pin and RTS pin active when opening a serial port (some RS-422/485 converters need this to enable the outputbuffers)
- added the baudrates 500000 and 1000000 for windows, this can be usefull when using an FTDI-chip or USB-converter
- added the devices "/dev/ttyAMA0" and "/dev/ttyAMA1" for use with the Raspberry Pi
- added the devices "/dev/ttyACM0" and "/dev/ttyACM1" for use with the Atmel (USB-)microcontrollers
- added the devices "/dev/rfcomm0" and "/dev/rfcomm1" for use with Bluetooth
- added the devices "/dev/ircomm0" and "/dev/ircomm1" for Infrared communication
- added the following functions: RS232_enableDTR(), RS232_disableDTR(), RS232_enableRTS(), RS232_disableRTS() and RS232_IsDSREnabled()
- changed function "cprintf()" to "RS232_cputs()"

The sourcecode

- https://gitlab.com/Teuniz/RS-232

Usage:

git clone https://gitlab.com/Teuniz/RS-232.git

Functions

int RS232_OpenComport(*int comport_number, int baudrate, const char * mode, int flowctrl*)

Opens the comport, comportnumber starts with 0 (see the list of numbers).
Baudrate is expressed in baud per second i.e 115200 (see the list of possible baudrates).
Mode is a string in the form of "8N1", "7E2", etc.
8N1 means eight databits, no parity, one stopbit. If in doubt, use 8N1 (see the list of possible modes).
If flowctrl is set to 0, no flow control is used.
If flowctrl is set to 1, hardware flow control is enabled using the RTS/CTS lines.
Returns 1 in case of an error.

int RS232_PollComport(*int comport_number, unsigned char *buf, int size*)

Gets characters from the serial port (if any). Buf is a pointer to a buffer
and size the size of the buffer in bytes.
Returns the amount of received characters into the buffer. This can be less than size or zero!
It does not block or wait, it returns immediately, no matter if any characters have been received or not.
After succesfully opening the COM-port, connect this function to a timer.
The timer should have an interval of approx. 100 milliSeconds.
Do not forget to stop the timer before closing the COM-port.

int RS232_SendByte(*int comport_number, unsigned char byte*)
Sends a byte via the serial port. Returns 1 in case of an error.

int RS232_SendBuf(*int comport_number, unsigned char *buf, int size*)

Sends multiple bytes via the serial port. Buf is a pointer to a buffer
and size the size of the buffer in bytes.
Returns -1 in case of an error, otherwise it returns the amount of bytes sent.
This function blocks (it returns after all the bytes have been processed).

void RS232_CloseComport(*int comport_number*)

Closes the serial port.

void RS232_cputs(*int comport_number, const char *text*)

int RS232_GetPortnr(*const char *device*)

Returns the comport number based on the device name e.g. "ttyS0" or "COM1". (Doesn't mean the device actually exists!)
Returns -1 when not found.

The following functions are normally not needed but can be used to set or check the status of the control-lines:

void RS232_enableDTR(*int comport_number*)

Sets the DTR line high (active state).

void RS232_disableDTR(*int comport_number*)

Sets the DTR line low (non active state).

void RS232_enableRTS(*int comport_number*)

Sets the RTS line high (active state). Do not use this function when hardware flow control is enabled!

void RS232_disableRTS(*int comport_number*)

Sets the RTS line low (non active state). Do not use this function when hardware flow control is enabled!

int RS232_IsDSREnabled(*int comport_number*)

Checks the status of the DSR-pin. Returns 1 when the the DSR line is high (active state), otherwise 0.

int RS232_IsCTSEnabled(*int comport_number*)

Checks the status of the CTS-pin. Returns 1 when the the CTS line is high (active state), otherwise 0.

int RS232_IsDCDEnabled(*int comport_number*)

Checks the status of the DCD-pin. Returns 1 when the the DCD line is high (active state), otherwise 0.

Checks the status of the RING-pin. Returns 1 when the the RING line is high (active state), otherwise 0.

The following functions are normally not needed but can be used to empty the rx/tx buffers: ("discards data written to the serial port but not transmitted, or data received but not read")

void RS232_flushRX(*int comport_number*)

Flushes data received but not read.

void RS232_flushTX(*int comport_number*)

Flushes data written but not transmitted.

void RS232_flushRXTX(*int comport_number*)

Flushes both data received but not read, and data written but not transmitted.

Notes:

You don't need to call RS232_PollComport() when you only want to send characters.
Sending and receiving do not influence eachother.

The os (kernel) has an internal buffer of 4096 bytes.
If this buffer is full and a new character arrives on the serial port,
the oldest character in the buffer will be overwritten and thus will be lost.

After a successfull call to RS232_OpenComport(), the os will start to buffer incoming characters.

Do not use microsoft tools to compile this library.
The microsoft C-compiler is an old and retarded compiler that does not even support ANSI C99.

Demo:

Example code that demonstrates how to use the library to receive characters and print them to the screen:

```
/**************************************************
```

```
exit the program by pressing ctrl-c

compile with the command: gcc demo_rx.c rs232.c -Wall -Wextra -o2 -o test_rx

**********************************************/

#include <stdlib.h>
#include <stdio.h>

#ifdef _WIN32
#include <Windows.h>
#else
#include <unistd.h>
#endif

#include "rs232.h"



int main()
{
  int i, n,
      cport_nr=0,        /* /dev/ttyS0 (COM1 on windows) */
      bdrate=9600;       /* 9600 baud */

  unsigned char buf[4096];

  char mode[]={'8','N','1',0};


  if(RS232_OpenComport(cport_nr, bdrate, mode))
  {
    printf("Can not open comport\n");

    return(0);
  }

  while(1)
  {
    n = RS232_PollComport(cport_nr, buf, 4095);

    if(n > 0)
    {
      buf[n] = 0;   /* always put a "null" at the end of a string! */

      for(i=0; i < n; i++)
      {
        if(buf[i] < 32)  /* replace unreadable control-codes by dots */
        {
          buf[i] = '.';
        }
      }

      printf("received %i bytes: %s\n", n, (char *)buf);
    }

#ifdef _WIN32
    Sleep(100);
#else
    usleep(100000);  /* sleep for 100 milliSeconds */
#endif
  }

  return(0);
}
```

Example code that demonstrates how to use the library to transmit characters and print them to

```
/**************************************************

file: demo_tx.c
purpose: simple demo that transmits characters to
the serial port and print them on the screen,
exit the program by pressing Ctrl-C

compile with the command: gcc demo_tx.c rs232.c -Wall -Wextra -o2 -o test_tx

**************************************************/

#include <stdlib.h>
#include <stdio.h>

#ifdef _WIN32
#include <Windows.h>
#else
#include <unistd.h>
#endif

#include "rs232.h"



int main()
{
  int i=0,
      cport_nr=0,        /* /dev/ttyS0 (COM1 on windows) */
      bdrate=9600;       /* 9600 baud */

  char mode[]={'8','N','1',0},
       str[2][512];


  strcpy(str[0], "The quick brown fox jumped over the lazy grey dog.\n");

  strcpy(str[1], "Happy serial programming!.\n");


  if(RS232_OpenComport(cport_nr, bdrate, mode))
  {
    printf("Can not open comport\n");

    return(0);
  }

  while(1)
  {
    RS232_cputs(cport_nr, str[i]);

    printf("sent: %s\n", str[i]);

#ifdef _WIN32
    Sleep(1000);
#else
    usleep(1000000);  /* sleep for 1 Second */
#endif

    i++;

    i %= 2;
  }

  return(0);
}
```

Look here for a timer library.

Look here for a serial communication tester/debugger.

List of comport numbers, possible baudrates and modes:

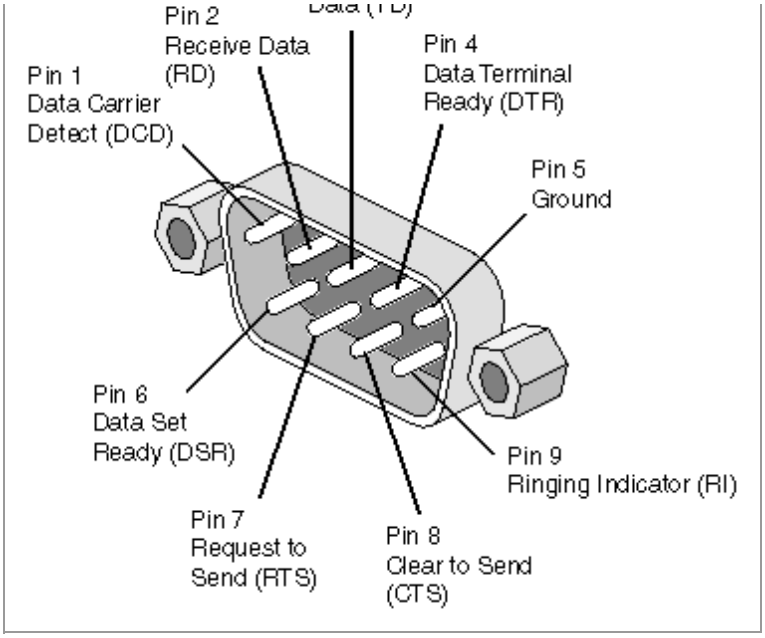
1	ttyS1	COM2
2	ttyS2	COM3
3	ttyS3	COM4
4	ttyS4	COM5
5	ttyS5	COM6
6	ttyS6	COM7
7	ttyS7	COM8
8	ttyS8	COM9
9	ttyS9	COM10
10	ttyS10	COM11
11	ttyS11	COM12
12	ttyS12	COM13
13	ttyS13	COM14
14	ttyS14	COM15
15	ttyS15	COM16
16	ttyUSB0	COM17
17	ttyUSB1	COM18
18	ttyUSB2	COM19
19	ttyUSB3	COM20
20	ttyUSB4	COM21
21	ttyUSB5	COM22
22	ttyAMA0	COM23
23	ttyAMA1	COM24
24	ttyACM0	COM25
25	ttyACM1	COM26
26	rfcomm0	COM27
27	rfcomm1	COM28
28	ircomm0	COM29
29	ircomm1	COM30
30	cuau0	COM31
31	cuau1	COM32
32	cuau2	n.a.

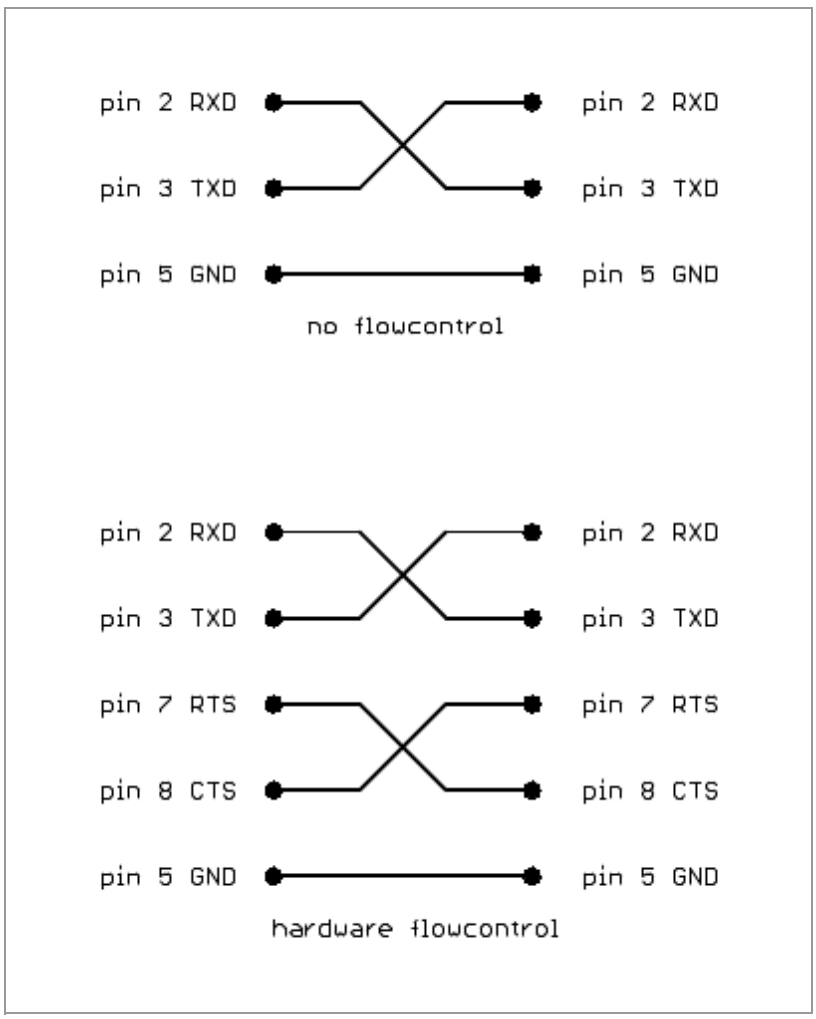
75	n.a.
110	110
134	n.a.
150	n.a.
200	n.a.
300	300
600	600
1200	1200
1800	n.a.
2400	2400
4800	4800
9600	9600
19200	19200
38400	38400
57600	57600
115200	115200
230400	128000
460800	256000
500000	500000
576000	n.a.
921600	921600
1000000	1000000
1152000	n.a.
1500000	1500000
2000000	2000000
2500000	n.a.
3000000	3000000
3500000	n.a.
4000000	n.a.

8O1
8E1
8N2
8O2
8E2
7N1
7O1
7E1
7N2
7O2
7E2
6N1
6O1
6E1
6N2
6O2
6E2
5N1
5O1
5E1
5N2
5O2
5E2

Note: Traditional (on-board) UART's usually have a speed limit of max. 115200 baud. Special cards and USB to Serial converters can usually be set to higher baudrates.

Connector pinlayout


Pin 2
Receive Data
(RD)
Pin 1
Data Carrier
Detect (DCD)
Pin 4
Data Terminal
Ready (DTR)
Pin 5
Ground
Pin 6
Data Set
Ready (DSR)
Pin 9
Ringing Indicator (RI)
Pin 7
Request to
Send (RTS)
Pin 8
Clear to Send
(CTS)


pin 2 RXD
pin 3 TXD
pin 5 GND
pin 2 RXD
pin 3 TXD
pin 5 GND
no flowcontrol
pin 2 RXD
pin 3 TXD
pin 7 RTS
pin 8 CTS
pin 5 GND
pin 2 RXD
pin 3 TXD
pin 7 RTS
pin 8 CTS
pin 5 GND
hardware flowcontrol