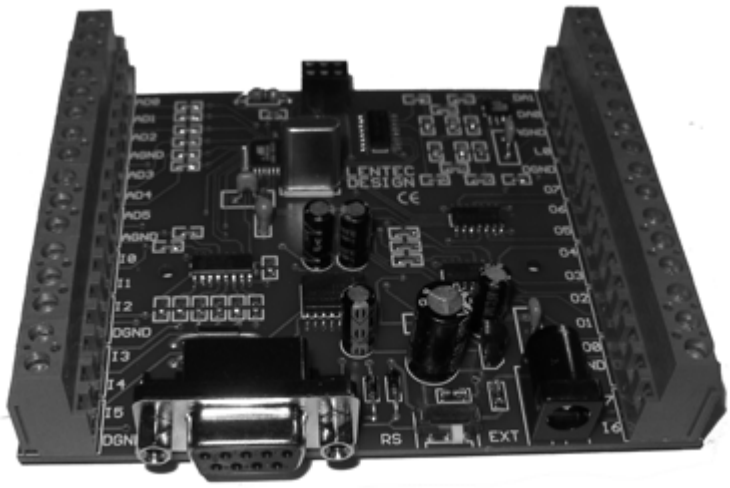
MANUAL FOR DEVICE

LENDEVICE80RS232

LENTEC DESIGN



INTRODUCTION

The device LENDEVICE80RS232 can be used to measure and generate voltage signal. The device is controlled by computer program (when computer communicates with the device using COM port – RS232).

The device has ability:

- **measure analog signals with resolution 10 bits in range 0 V – 5 V (6 inputs)**
- **generate analog signals with resolution 10 bits in range 0 V – 5 V (2 outputs)**
- **measure digital signals in standard TTL in range 0 V – 5 V (8 inputs)**
- **generate digital signals in standard TTL in range 0 V – 5 V (8 outputs)**
- **count pulses for maximum frequency 900 kHz (1 input)**

Additionally the program in the device can be upgraded and in future it will be increased scope of activity these devices. It exist ability to create special programs for customers, who want to use the device in own applications.

Power supply to the module can be from the COM port of the computer or from an external voltage source 9 V ÷ 12 V.

We thank you for choosing the product of Lentec Design. Our aim is to make your projects successful.

The team of Lentec Design
www.lentecdesign.com
office@lentecdesign.com

COPYRIGHTS

TABLE OF CONTENTS

1. USING THE DEVICE

Layout of inputs and outputs of device LENDEVICE80R232 shown in Fig. 1 below.

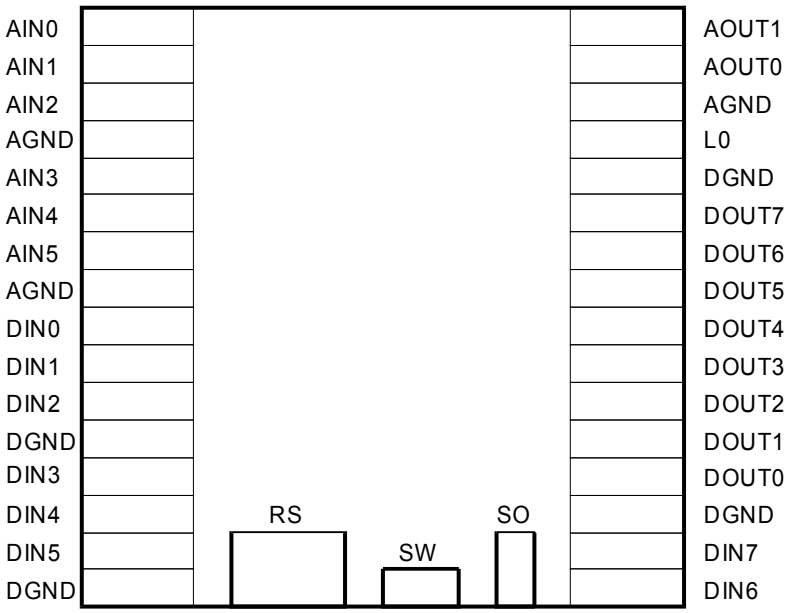


Fig. 1. *Inputs and Outputs for LENDEVICE80RS232 device (top view).*

AGND and DGND are connected with the ground of the device 0 V.

All units are in International System of Units (SI).

1.1. ANALOG INPUTS

The device has 6 analog inputs: AIN0, AIN1, ..., AIN5, which serve to measure voltage in the range 0 V to 5 V. If the input voltage exceeds this range, the device may fail. In order to decrease noise on the inputs, you should use ground connection (AGND) located in the vicinity of analog inputs.

Analog input(s) typically take 10 μA (see Appendix C: Characteristics of analog inputs, p. 34).

In order to measure voltage on the analog input, you can use the supplied program 'LenOscilloscope' or if you are writing your own program you can use one of the functions [methods] below described in Chap. 2.14: “Measuring of analog voltages” on p. 21 (see Chap. 2.2: “Interface in C and C++”, p. 11):

- `analog_input_len(...)`
  `[analog_input(...)]`
- `AnalogInputLen(...)`
  `[AnalogInput(...)]`

1.2. ANALOG OUTPUTS

The device has 2 analog outputs, AOUT0 and AOUT1. They have a closed – loop output impedance of 50 Ω (typical). These outputs can generate voltage in the range of 0 V to 5 V.

Maximum noise between peaks of generated voltage is 10 mV.

Connecting an external power supply to analog outputs may cause the device to fail.

Maximal current which can flow through the output is 10 mA (see Appendix D: Characteristics of analog outputs, p. 35).

In order to set the value of analog voltage on the output, you have to use either the supplied program LenOscilloscope, or one of the functions [method] described in Chap. 2.15: "Controlling of analog voltages" on p. 21 (see Chap. "Interface in C and C++", p. 11):

- `analog_output_len(...)`
  `[analog_output(...)]`
- `AnalogOutputLen(...)`
  `[AnalogOutput(...)]`

1.3. DIGITAL INPUTS

The device has 8 digital inputs; DIN0, DIN1, ..., DIN7. Characteristics of the digital inputs is given in the Table "Characteristics of digital inputs" (see Appendix E: Characteristics of digital and counter inputs on p. 35).

- `digital_input_len(...)`
  `[digital_input(...)]`

In order to measure states on the digital inputs, you have to use either the supplied program 'LenOscilloscope', or one of the functions [methods] described int Chap. 2.16: "Getting logical states on digital inputs" on p. 22 (see Chap. 2.2: "Interface in C and C++", p. 11):

- `is_hi_digital_input(...)`
  `[is_hi_digital_input(...)]`
- `is_lo_digital_input(...)`
  `[is_lo_digital_input(...)]`
- `DigitalInputLen(...)`
  `[DigitalInput(...)]`
- `IsHiDigitalInputLen(...)`
  `[IsHiDigitalInput(...)]`
- `IsLoDigitalInputLen(...)`
  `[IsLoDigitalInput(...)]`

1.4. DIGITAL OUTPUTS

The device has 8 digital outputs: DOUT0, DOUT1, ..., DOUT7. Characteristics of the digital outputs is given in the Table "Characteristics of digital outputs" (see Appendix F: Characteristics of digital outputs on p. 35).

Connecting an external power source to digital outputs may fail the device.

Maximum current which can flow through the outputs is 5 mA.

In order to set digital states on the outputs, you have to use one of the functions [methods] described in Chap. 2.17: “Controlling of digital outputs” on p. 24 (see Chap. 2.2: “Interface in C and C++”, p. 11):

- `digital_output_len(...)`
  `[digital_output(...)]`
- `hi_digital_output_len(...)`
  `[hi_digital_output(...)]`
- `lo_digital_output_len(...)`
  `[lo_digital_output(...)]`
- `DigitalOutputLen(...)`
  `[DigitalOutput(...)]`
- `HiDigitalOutputLen(...)`
  `[HiDigitalOutput(...)]`
- `LoDigitalOutputLen(...)`
  `[LoDigitalOutput(...)]`

1.5. COUNTER'S INPUT

The device has 1 counter input, L0, which can count pulses. Counted pulses have to have a high state (above 3.7 V) for a minimum time of 0.56 µs, and a low state (below 1.3 V) for a minimum time of 0.56 µs (see Fig. 2). Maximum frequency of measured pulses is 900 kHz (see Appendix E: Characteristics of digital and counter inputs on p. 35).

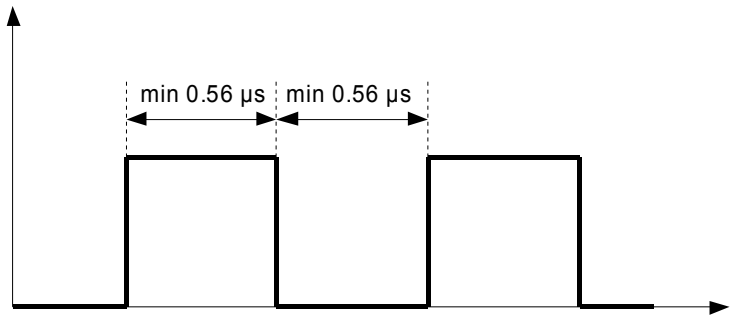


Fig. 2. *Pulse characteristic, which can be measured by device on the input LO.*

In order to read counted pulses, you have to use the function [method] described in Chap. 2.18: “Controlling the counter” on p. 26 (see Chap. 2.2: “Interface in C and C++”, p. 11):

- `DWORD get_counter_len(const LENDEVICE hdevice, const INT number)`
  `[unsigned long get_counter(const int number)]`
- `BOOL GetCounterLen(const LENDEVICE hdevice, const INT number,`
  `DWORD* result_ptr)`
  `[bool GetCounter(const int number, unsigned long* result_ptr)]`

In order to clear counter, call the function [method] described in Chap. 2.18: “Controlling the counter” on p. 26 (see Chap. 2.2: “Interface in C and C++“, p. 11):

- `VOID clear_counter_len(const LENDEVICE hdevice, const INT number)`
  `[void clear_counter(const int number)]`
- `BOOL ClearCounterLen(const LENDEVICE hdevice, const INT number)`
  `[bool ClearCounter(const int number)]`

1.6. POWER SUPPLY OF THE DEVICE

The LENDEVICE80R232 can be provided with power in two ways:

- From the COM port of the computer: The device may not supply more than 5 mA from all outputs. If the power consumption from the outputs is higher, the device may work incorrectly.

  Before connecting other devices and microchips, you have to ensure, that power to the device is supplied. This is achieved by running the OpenLenDevice program (the shortcut to the program should be on the desktop or available from the START menu, see also Chap. 2.21: “OpenLenDevice program” on p. 29) or calling the functions [methods] below described in Chap. 2.3: “Connecting with the device” on p. 13, (see Chap. 2.2: “Interface in C and C++”, p. 11):

  - `open_len(...)`
    `[open(...)]`
  - `open_with_baudrate_len(...)`
    `[open_with_baudrate(...)]`

  If these functions [methods] are used to open the device, then the device is supplied from the COM port during whole session of the system, if it was not disconnected from the computer.

- From an external (stable) power supply with an output voltage in the range of 9 V to 12 V. The power source should be connected to the plug–in socket, SO. The device uses a maximal 60 mA from all outputs.

  The shape of the plug–in socket, SO is shown in Fig. 3. A suitable plug-in connector is included with the device.

  Incorrect connection of the power supply may destroy the device.

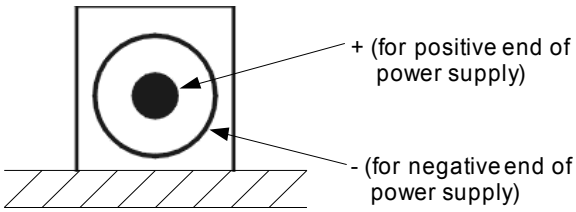


Fig. 3. *Shape of plug–in socket, SO.*

The method of power supply must be set using the toggle switch, SW. The switch positions are shown below in Fig. 4.

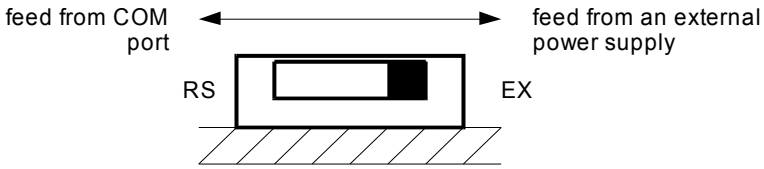


Fig. 4. *Shape of toggle switch, SW.*

If the power consumption from all outputs of the device exceed the 60 mA limit, the device may fail.

1.7. COMMUNICATION WITH A COMPUTER

The device communicates with a computer by cable connected to the COM port of the computer and the plug–in socket of the device, denoted by symbol RS. This connection is compatible with DB9 COM port. The configuration of the RS232 connection is given in Fig. 5 and Table 1 below:

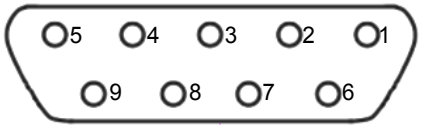


Fig. 5. *RS232 pin layout.*

Table 1 shows the function of the particular transmission lines of the plug-in socket, RS.

Tab. 1. *The functions of the transmission lines of RS232 connection.*

No of line	Symbol of line	Function of line
1	DCD	Data Carrier Detected
2	RD	Received Data
3	TD	Transmitted Data
4	DTR	Data Terminal Ready
5	SG	Signal Ground
6	DSR	Data Set Ready
7	RTS	Request To Send
8	CTS	Clear To Send
9	RI	Ring Indicator, Not Used

Connecting the device requires a typical one-to-one connection cable. If there is another spacing of transmission lines in the computer (or there is a DB25 connector), you have to

use a module which changes the electrical wiring. Other spacings of the transmission lines, which can be in COM port, are presented in Appendix B: Electrical characteristics on the RS plug-in socket on p. 33. If the system UART of the computer services only transmission in handshaking mode, than you have to use an external power supply (see Chap. 1.6: “Power supply of the device” on p. 8).

The device will communicate at the following baud rates (bits / second):

- 115200
- 57600
- 38400
- 19200
- 14400
- 9600
- 4800
- 2400
- 1200
- 600
- 300

The default baud rate is 9600 b/s. You can change this baud rate using the supplied software LenOscilloscope or OpenLenDevice, or use the following function [method] described in Chap. 2.8: “Change the startup baud rate of the transmission” on p. 17 (see Chap. 2.2: “Interface in C and C++”, p. 11):

- `set_startup_baudrate_len(...)`
  `[set_startup_baudrate(...)]`

Note: If using this function [method], because of the EEPROM memory, the device will only store this information for a limited time (approximately 100,000 start-up cycles).

The actual baud rate of the transmission can be also changed (for example from 9600 b/s to faster 115200 b/s), you can use the function [method] described in Chap. 2.7: “Change the actual baud rate of the transmission” on p. 16 (see Chap. 2.2: “Interface in C and C++”, p. 11):

- `change_baudrate_len(...)`
  `[change_baudrate(...)]`

You can use this function [method] an arbitrary number times.

2. CONTROLLING THE DEVICE

You can control the device by computer with one of the following Microsoft Windows Operating System installed: Windows 98 / ME / NT / 2000 / XP, using the delivered software.

In order to use the device in your own programs, you have to add files:

- "lendev.lib" − static library, this file are mandatory;
- "lendev.h" − header file, this file are mandatory;
- "lendev.hpp", this file is optional.

"lendev.lib" file serves to use functions exported from lendev.dll library. Appendix H: Adding the static library (*.LIB) to the project on p. 36 shows, how to add static library (*.lib) to a project in some integrated development environments (IDE).

Appendix G: Adding a new path to the directory, which are scanned in Integrated Development Environments (IDE) on p. 36 shows, how to add a shortcuts to folders, which are searched in IDE during looking for files. In Appendix I: Information for developers on p. 37 there is information for developers about "lendev.dll" library and the driver "lenR232.exe". After including the files, you can start programming.

2.1. INFORMATION ABOUT INCLUDED EXAMPLES

All programs in this documentation serve to show the rules of using the software to control the device. These are simple programs, which run in a console of the Windows system. You can use the software in arbitrary applications written in C / C++. The programs have been tested on compilers: Visual C++ 8.0, Visual C++ 7.0, Visual C++ 6.0, MinGW 3.4.

2.2. INTERFACE IN C AND C++

The device can be controlled by either functions declared in file "lendev.h", or by methods of the `len::DeviceRS232` class defined in file "lendev.hpp". Before starting your programming, you will need to decide which way you want to program: using functions taking `LENDEVICE` handle, or object of `len::DeviceRS232` class. In both interfaces there are functions and methods, which control the device identically and differ only in name.

For example, you can measure a voltage on an analog input calling the function `analog_input_len(...)`, or the method `analog_input(...)` (more precisely `len::DeviceRS232::analog_input(...)`, but the name of class `len::DeviceRS232` is omitted in this document).

The text method of the `len::DeviceRS232` class corresponds to function from "lendev.h" file is given in square bracket, for example:

- `FLOAT analog_input_len(const LENDEVICE hdevice, const INT channel)`
  `[float analog_input(const int channel)]`

Most applications, which control the device are duplicated in order to make their use in

programs easier. For example, in file “lendev.h” there are two functions declared, which returns the value of measured voltage on analog inputs:

- `FLOAT analog_input_len(const LENDEVICE hdevice, const INT channel)`
- `BOOL AnalogInputLen(const LENDEVICE hdevice, const INT channel, FLOAT* result_ptr)`

The first function, above, returns the value of voltage.

Note: Error Information, on whether the operation was finished successfully can be obtained by calling the following function (see Chap. 2.6: “Getting information about errors” on p. 16):

- `INT get_last_error_len(const LENDEVICE hdevice)`

The second listed function returns a result of success of operation:

- `TRUE` – Operation Successful;
- `FALSE` – Operation Failed.

To get extended error information, call `get_last_error_len(...)` function. It is handy to use `analog_input_len(...)` function, when you are sure, the device is correctly connected to the computer. For example:

Example 1.

```
#include "lendev.h"
#include "stdio.h"

int main()
{
  LENDEVICE hdev = open_len(1); // see chap. 2.3
  if ( hdev != NO_LENDEVICE )   // a while ago was opened the device,
  {                             // so now it is too
    printf("Value of voltage on channel 0: %.3f V\n",
      analog_input_len(hdev, AIN0));
    close_len(hdev);            // see chap. 2.4
  }
  else
    printf("Unable to establish connection.\n");
  system("PAUSE");
  return 0;
}
```

It is handy to use `AnalogInputLen(...)` function, when you are using the device for long time, to ensure input source is still connected. i.e.; damage of cable, disconnect of device, etc.

Example 2 shows program, in which `AnalogInputLen(...)` function is used:

Example 2.

```
#include "lendev.h"
#include "stdio.h"
```

Example 2.

```
int main()
{
  LENDEVICE hdev = open_len(1); // see chap. 2.3
  if ( hdev != NO_LENDEVICE )
  {
    FLOAT value_of_voltage;
    for (int no_test = 0; no_test < 10; ++no_test )
    {
      if ( AnalogInputLen(hdev, 0, &value_of_voltage) == TRUE )
      {
        // operation was successful
        printf("Value of voltage on channel 0: %.3f V.\n",
          value_of_voltage);
      }
      else
        printf("It is impossible to measure voltage.\n");
      Sleep(2000);  // waiting 2 seconds
    }
    close_len(hdev); // see chap. 2.4
  }
  else
    printf("Unable to establish connection .\n");
  system("PAUSE");
  return 0;
}
```

Similar interface was created in `len::DeviceRS232` class. For example, the method:

- `float analog_input(const int channel)`

returns value of measured voltage, and method:

- `bool AnalogInput(const int channel, float* result_ptr)`

returns the result of the success of operation.

2.3. CONNECTING WITH THE DEVICE

Before the first use of the device, you have to open the device, and obtain a handle of `LENDEVICE` type. In order to do this you have to call function:

- `LENDEVICE open_len(const INT port)`

  `port` = 1 for COM1, `port` = 2 for COM2, ..., it has to be within range: 1 ⩽ `port` ⩽ 256. The function returns a handle of `LENDEVICE` type: if returned value is `NO_LENDEVICE`, it means, that the connection with the device failed.

  Function opens the device beginning a communication at 9600 b/s. If it doesn't succeed at this baud rate, it tries to communicate with the device at other transmission baud rates only if system UART of computer enables transmit and receive data, in listed sequence: 115200, 57600, 38400, 19200, 14400, 9600, 4800, 2400, 1200, 600, 300 b/s.

  The new devices are set to startup at baud rate equal to 9600 b/s, but it can be changed with the aid of function `set_startup_baudrate_len(...)` (see Chap. 2.8: “Change the startup baud rate of the transmission” on p. 17), or the baud rate can be changed by

calling function `change_baudrate_len(...)` (see Chap. 2.7: “Change the actual baud rate of the transmission” on p. 16).

To speed up the the start-up, for slower connection speeds (at end of list above), e.g. 300 b/s, then you can use the following function:

- `LENDEVICE open_with_baudrate_len(const INT port, const BaudRate baudrate)`

`port` = 1 for COM1, `port` = 2 for COM2, it has to be within range: 1 ⩽ `port` ⩽ 256 The function returns a handle of `LENDEVICE` type: if returned value is `NO_LENDEVICE`, it means, that connection with the device failed. This function opens the device starting the communication with baud rate `baudrate` given as a parameter.

Every opened device has to be closed (see Chap. 2.4: “Shutting down the device” on p. 15).

In `len::DeviceRS232` class the following methods are equal to the functions above:

- `bool open(const int port)` – equivalent `open_len(...)` function;
- `bool open_with_baudrate(const int port, const BaudRate baudrate)` – equivalent `open_with_baudrate_len(...)` function.

Both methods return the result of the success of opening the device.

The device can be opened with the aid of constructors, too:

- `DeviceRS232(const int port)` – calls `open(...)` method;
- `DeviceRS232(const int port, const BaudRate baudrate)` – calls `open_with_baudrate(...)` method.

Examples 3 and 4 below, show directions of opening and closing the device.

Example 3.

```
#include "lendev.h"
#include "stdio.h"

int main()
{
   LENDEVICE hdev = open_with_baudrate_len(1, BR_9600); // opening
   if ( hdev != NO_LENDEVICE )  // checking, if it was opened
   {
      printf("The device was opened\n");
      // we do a job
      // we call functions declared in "lendev.h" file
      // and always use hdev parameter
      close_len(hdev);           // obligatory closing the device
      // the hdev handle is out-of-date and you can't use its
   }
   else
      printf("Unable to establish connection .\n");
   system("PAUSE");
   return 0;
}
```

Example 4.

```
#include "lendev.hpp"
#include <iostream>

int main()
{
   len::DeviceRS232 dev;  // an object,
   // which enables controlling the device
   if ( dev.open_with_baudrate(1, BR_115200) ) // startup baud rate of
                                               // transmission = 115200 b/s
   {
      std::cout << "The device was opened\n";
      // we do a job
      // we call a methods len::DeviceRS232 class defined
      // in "lendev.hpp" file

      dev.close();  // optional closing the device
      // if we don't do this, then the destructor
      // len::DeviceRS232::~DeviceRS232() of class does this
   }
   else
      std::cout << "Unable to establish connection .\n";
   system("PAUSE");
   return 0;
}
```

2.4. SHUTTING DOWN THE DEVICE

Before work is finished with the device, you have to close the handle of `LENDEVICE` type. In order to do this call function [method]:

- `BOOL close_len(const LENDEVICE hdevice)`
  `[bool close()]`

Both the function and method return the result of the success of the operation.

Method `close()` class does not have to be called explicitly, because it done by the class destructor.

2.5. TESTING OF CONNECTED DEVICE

To test, if the device is connected to the computer, call the function:

- `BOOL is_open_len(const LENDEVICE hdevice,`
  `const BOOL check_obligatorily)`
  `[bool is_open(const bool check_obligatorily = false)]`

If parameter `check_obligatorily` is `TRUE [true]`, it calls the function [method] `check_device_len(...) [check_device(...)]`. If parameter `check_obligatorily` is `FALSE [false]`, then it calls the function [method] `check_device_len(...) [check_device(...)]` only if last instruction was ended unsuccessfully. The function [method] `is_open_len(...) [is_open(...)]` returns `FALSE [false]` value, if the function [method] `check_device_len(...) [check_device(...)]` returns

DEVICE_OFF value (see Chap. 2.10: “Checking the device” on p. 19). In other cases, this function returns TRUE [true].

If you can not connect with the device, you can call the function [method] reset_device_len(...) [reset_device()] (see Chap. 2.11: “Resetting the device” on p. 20).

2.6. GETTING INFORMATION ABOUT ERRORS

You can obtain the information about last error code by calling the function [method]:

◆ INT get_last_error_len(const LENDEVICE hdevice)
[int get_last_error()]

This function [method] returns one of the following value defined in file “lendev.h”:

- ERROR_SUCCESS_LEN – no errors;
- ERROR_SYSTEM_PROBLEM_LEN – the system doesn't make available the last operation. This problem occurs e.g. when user wants to change baud rate, which is not made available by UART chip on the computer board (see Chap. 2.20: “Getting information about COM port of computer” on p. 27).
- ERROR_INVALID_PARAMETERS_LEN – wrong value of some or all of the parameters in the called function, e. g. user wants to change voltage on channel number 100 of the analog outputs, but this output doesn't exist.
- ERROR_COMMUNICATION_LEN – communication error between the device and the computer. It may occur if:
  - the device is connected to wrong COM port of the computer;
  - the device is not supplied with power because the setting of power source switch is incorrect (see Chap. 1.6: “Power supply of the device” on p. 8).
  - the device and COM port of the computer are set at different baud rates.
- ERROR_INVALID_INSTRUCTION_LEN – error which comes from calling an instruction which is not possible to do on the device (see Chap. 2.12: “Checking of the device's type” on p. 20).
- ERROR_BAD_HANDLE_LEN – wrong handle of LENDEVICE type to the device.

2.7. CHANGE THE ACTUAL BAUD RATE OF THE TRANSMISSION

In order to change the actual baud rate of the transmission, call the function [method]:

◆ BOOL change_baudrate_len(const LENDEVICE hdevice,
const BaudRate new_baudrate)
[bool change_baudrate(const BaudRate new_baudrate)]

The device enables transmission of data at the following baud rate (b/s):

300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200

The parameter new_baudrate can have one of the following values:

- BR_300,

- BR_600,
- BR_1200,
- BR_2400,
- BR_4800,
- BR_9600,
- BR_14400,
- BR_19200,
- BR_38400,
- BR_57600
- BR_115200

The function [method] returns the result of the success of operation.

After opening the device, the actual baud rate is equal to the startup baud rate.

Before changing the baud rate, you have to be sure that UART chip of the computer enables transmission at the chosen baud rate (see Chap. 2.20: "Getting information about COM port of computer" on p. 27).

In order to change baud rate, it is recommended to use function [method] `change_baudrate_len(...)` `[change_baudrate(...)]`. If any problem occurs when changing the actual baud rate, you should change by using function [method] `set_startup_baudrate_len(...)` `[set_startup_baudrate(...)]` (see Chap. 2.8: "Change the startup baud rate of the transmission" on p. 17), then turn the device off and back on by disconnecting the power supply to the device and reconnecting the power supply.

2.8. CHANGE THE STARTUP BAUD RATE OF THE TRANSMISSION

The startup baud rate is the baud rate, which the device starts working at. The default startup baud rate of the transmission is 9600 b/s, but it can be changed using the function:

◆
```
BOOL set_startup_baudrate_len(const LENDEVICE hdevice,
    const BaudRate new_baudrate)
[bool set_startup_baudrate(const BaudRate new_baudrate)]
```

Parameter `new_baudrate` can have the same values as the function `change_baudrate_len(...)` (see Chap. 2.7: "Change the actual baud rate of the transmission" on p. 16). The function [method] returns the result of the success of operation.

Another startup baud rate than 9600 b/s can be useful, e.g. if a long wire is used to communicate between the computer and the device. For example, you can use wire 15 m long and transmit data with baud rate 19200 b/s (typical). You can use also current loop and transmit data with baud rate 9600 b/s through 4000 m long wire or 38400 b/s on distance 500 m.

It is not recommended to call often the function [method] `set_baudrate_len(...)` `[set_baudrate(...)]`, because the EEPROM memory of the device has limited number cycles for storing data (approximately 100000 times).

2.9. MODE OF COMMUNICATION WITH THE DEVICE

A computer can communicate with the device using one of the following communication's mode:

- `COMM_FAST_MODE` – fast mode of communication: consist in transmitting data to the device without checking of their correctness; you can control the device faster, but you haven't a guarantee, that an error doesn't occur during data transmission.
- `COMM_CONTROLLED_MODE` – mode of communication with control of the transmitted data: consist in transmitting every instruction from transmitter, then the receiver retransmitting the instruction to the transmitter. The transmitter checks if correct and sends a positive or negative acknowledgement to the receiver. The transmitter can be computer or a LENDEVICE80RS232 device (identical receiver). If an error occurs during data transmission, the data is retransmitted once more.

Figure 6 shows comparison of transmitting data between transmitter and receiver in the two modes.

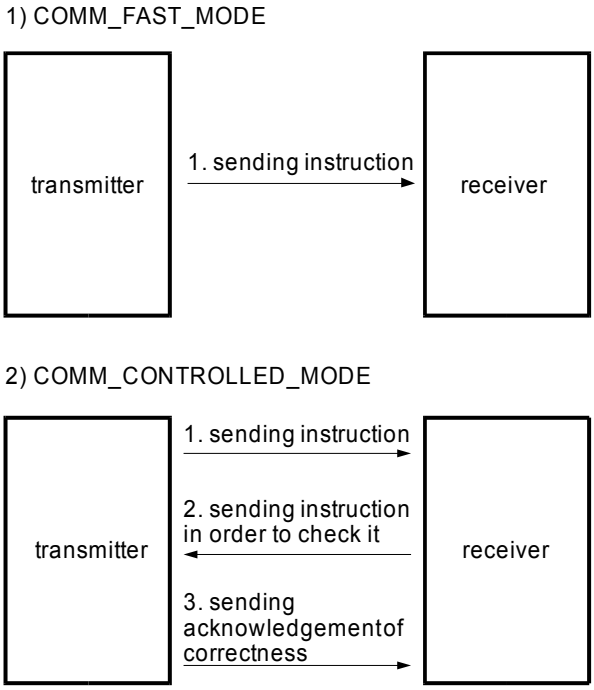


Fig. 6. *Algorithm of sending an instruction in two modes of communication.*

In order to change the mode of communication, call the function [method]:

- `BOOL change_communication_mode_len(const LENDEVICE hdevice, const CommunicationMode new_mode)`
  `[bool change_communication_mode(const CommunicationMode new_mode)]`

Parameter `new_mode` is a new mode of communication and it can have one of the following value:

- `COMM_FAST_MODE`

- COMM_CONTROLLED_MODE

The function [method] returns the result of the success of operation. The COMM_CONTROLLED_MODE mode is the default mode after turning on the LENDEVICE80RS232 device. Example 5 shows how to use the change_communication_mode(...) method.

Example 5.

```
#include "lendev.hpp"
#include <iostream>

int main()
{
  len::DeviceRS232 dev;
  if ( dev.open(1) )
  {
    std::cout << "The device was opened\n";
    std::cout << "Name of device: " << dev.info() << '\n';
      // see Chap. 2.12
    std::cout << "Change the actual baud rate to 115200 b/s:"
      << " success = " << std::boolalpha
      << dev.change_baudrate(BR_115200) << '\n';
    std::cout << "Change of mode of communication on COMM_FAST_MODE: "
      << "success = " << std::boolalpha
      << dev.change_communication_mode(COMM_FAST_MODE) << '\n';
    std::cout << "You can do next operation...\n";
  }
  else
    std::cout << Unable to establish connection\n";

  system("PAUSE");
  return 0;
}
```

2.10. CHECKING THE DEVICE

In order to check the device, call the function [method]:

◆ DeviceMode check_device_len(const LENDEVICE hdevice,
  const INT how_many)
[DeviceMode check_device(const int how_many = 10)]

Parameters how_many decides how many times (maximal) the device connection will be checked.

The function [method] returns one of the following values:

- DEVICE_OFF – there is not connection between computer and the device;
- DEVICE_ON – the device works properly;
- DEVICE_BAD_TRANSMISSION – the device is connected but some error occurs during transmission data;
- DEVICE_WITHOUT_PROGRAM – the device is connected but it is without a main program

(see Chap. 2.13: “Programming the device” on p. 20).

2.11. RESETTING THE DEVICE

The device can be reset. The resetting of the device clears all digital outputs, sets the analog outputs to 0 V and set the communication mode to `COMM_CONTROLLED_MODE`. Additionally, if an error with communication between a computer and the device occurs, the computer will try to connect with the device using every available baud rate. In this situation, the resetting may go on for some seconds (typical).

In order to reset the device, call the function [method]:

- `BOOL reset_device_len(const LENDEVICE hdevice)`
  `[bool reset_device()]`

The function [method] returns `TRUE [true]`, if there is correct communication with the device or else returns `FALSE [false]`.

2.12. CHECKING OF THE DEVICE'S TYPE

The software enables simultaneous use of many different devices of Lentec Design company, which use the COM port of computer. In order to check the type of device, call the function [method]:

- `INT type_of_device_len(const LENDEVICE hdevice)`
  `[int type_of_device()]`
- `VOID info_len(const LENDEVICE hdevice, BYTE information[40])`
  `[std::string info()]`

The function [method] `type_of_device_len(...) [type_of_device()]` returns one of the following values:

- `LENTECDEVICE_NOOPENED` = 0 – there is no connection with the device;
- `LENTECDEVICE80RS232` = 10 – the device is a LENDEVICE80RS232.

The manufacturer reserves the right to add a new returned value by this function [method] in new versions of the software, but the actual numerical values won't be changed.

The second function [method] `info_len(...) [info()]` returns information about the device in the text format. The parameter information is a pointer to the buffer on the ANSI sign. Returned string in the buffer is finished `'\0'` sign (like a C string).

2.13. PROGRAMMING THE DEVICE

The manufacturer can supply new firmware to LENDEVICE80RS232 device in the future in order to increase available functions. Every new installed firmware will require new software to be installed on the computer. If the programming of the device fails, this operation should be repeated. More information and new firmware will be available on the web page www.lentecdesign.com.

In order to upgrade the device, call the function [method]:

- `BOOL program_device_len(const LENDEVICE hdevice, const BYTE* file_ansi)`
  `[bool program_device(const std::string& file_ansi)]`

Parameter `file_ansi` is a pointer to the ANSI signs with stored path to the file of type "*.lhx" provided by manufacturer. The `'\0'` sign must be on the end of a string. The function [method] returns the result of the success of operation.

2.14. MEASURING OF ANALOG VOLTAGES

In order to measure analog voltage on the input channels of the device, you can use the function [method]:

- `FLOAT analog_input_len(const LENDEVICE hdevice, const INT channel)`
  `[float analog_input(const int channel)]`

or

- `BOOL AnalogInputLen(const LENDEVICE hdevice, const INT channel, FLOAT* result_ptr)`
  `[bool AnalogInput(const int channel, float* result_ptr)]`

The function [method] `analog_input_len(...)` `[analog_input(...)]` returns the value of the measured analog voltage or -1 if an error occurs. The function [method] `AnalogInputLen(...)` `[AnalogInput(...)]` returns the result of the success of operation and the value of the measured analog voltage is stored in variable with address `result_ptr`. In order to choose input channel, use `AIN0`, `AIN1`, ..., `AIN5` constants.

2.15. CONTROLLING OF ANALOG VOLTAGES

In order to set the analog voltage on an output of the device, you can use the function [method]:

- `VOID analog_output_len(const LENDEVICE hdevice, const INT channel, const FLOAT value)`
  `[void analog_output(const int channel, const float value)]`

or

- `BOOL AnalogOutputLen(const LENDEVICE hdevice, const INT channel, const FLOAT value)`
  `[bool AnalogOutput(const int channel, const float value)]`

These functions set value `value` of analog voltage on output `channel` of the device. In order to choose output channel, use `AOUT0` or `AOUT1` constants. The function [method] `AnalogOutputLen(...)` `[AnalogOutput(...)]` returns the result of the success of operation.

Example 6.

```
#include "lendev.hpp"
#include <iostream>

int main()
{
```

Example 6.

```
  len::DeviceRS232 dev;
  if ( dev.open(1) )
  {
    std::cout << "The device is opened\n";
    std::cout << "Name of the device: " << dev.info() << '\n';
    std::cout << "Change of actual baud rate to 115200 b/s:"
      << " success = " << std::boolalpha
      << dev.change_baudrate(BR_115200) << '\n';
    std::cout << "Change of mode of communication on COMM_FAST_MODE: "
      << "success = " << std::boolalpha
      << dev.change_communication_mode(COMM_FAST_MODE) << '\n';
    std::cout << "Measurements\n";
    for (int i = 0; i < 100; ++i)
    {
      printf("Measurement %3d:  %.3f  %.3f  %.3f [V]\n", i+1,
        dev.analog_input(AIN0), dev.analog_input(AIN1),
        dev.analog_input(AIN2));
    }
    std::cout << "Analog outputs\n";
    dev.analog_output(AOUT0, 2.5f);
    dev.analog_output(AOUT1, 1.0f);
  }
  else
    std::cout << "Unable to establish connection.\n";
  system("PAUSE");
  return 0;
}
```

2.16. GETTING LOGICAL STATES ON DIGITAL INPUTS

In order to read the digital states on inputs, call the function [method]:

- `DWORD digital_input_len(const LENDEVICE hdevice)`
- `[unsigned long digital_input()]`

This function [method] returns the states on digital inputs. There is a high state on DIN0 input, if the operation of binary conjunction of the returned value `bits` by `digital_input_len(...)` `[digital_input(...)]` (`bits = digital_input_len(...)`] and `DIN0` value is different than `0`:

```
if ( (bits & DIN0) != 0 )
{
  // there is high state on DIN0 input
}
else
{
  // there is low state on DIN0 input
}
```

You can check many digital inputs simultaneously, e.g.:

```
if ( (bits & (DIN0 | DIN1 | DIN2) ) != 0 )
{
```

```
  // there are high states on DIN0, DIN1 and DIN2 inputs
}
else
{
  // there are low state at least on one of the following digital input:
  // DIN0, DIN1 or/and DIN2
}
```

The second way to do above operation is use the function [method]:

- `BOOL is_hi_digital_input_len(const LENDEVICE hdevice, const DWORD inputs)`
  `[bool is_hi_digital_input(const unsigned long inputs)]`
  `[template<int N> bool is_hi_digital_input(const std::bitset<N>& inputs)]`

e.g.:

```
if ( is_hi_digital_input_len(hdev, DIN0 | DIN1 | DIN2) != FALSE )
{
  // there are high states on DIN0, DIN1 and DIN2 inputs
}
else
{
  // there are low state at least on one of the following digital input:
  // DIN0, DIN1 or/and DIN2
}
```

This function [method] returns value different than `FALSE [false]`, if there are high states on all listed inputs in parameter `inputs`, or else returns `FALSE [false]`.

The next function [methods] operates similarly:

- `BOOL is_lo_digital_input_len(const LENDEVICE hdevice, const DWORD inputs)`
  `[bool is_lo_digital_input(const unsigned long inputs)]`
  `[template<int N> bool is_lo_digital_input(const std::bitset<N>& inputs)]`

This function [method] returns value different than `FALSE [false]`, if there are low states on digital inputs on all listed inputs in parameter `inputs`, or else returns `FALSE [false]`.

The next function [methods] listed below operates likewise:

- `BOOL DigitalInputLen(const LENDEVICE hdevice, DWORD* result_ptr)`
  `[bool DigitalInput(unsigned long* result_ptr)]`
- `BOOL IsHiDigitalInputLen(const LENDEVICE hdevice, const DWORD inputs, BOOL* result_ptr)`
  `[bool IsHiDigitalInput(const unsigned long inputs, bool* result_ptr)]`
  `[template<int N> bool IsHiDigitalInput(const std::bitset<N>& inputs, bool* result_ptr)]`

```
◆ BOOL IsLoDigitalInputLen(const LENDEVICE hdevice,
    const DWORD inputs, BOOL* result_ptr)
  [bool IsLoDigitalInput(const unsigned long  inputs,
    bool* result_ptr)]
  [template<int N>
  bool IsLoDigitalInput(const std::bitset<N>& inputs,
    bool* result_ptr)]
```

They returns the result of the success of operation. The information about the states on digital inputs is stored in variable with address `inputs_ptr`.

2.17. CONTROLLING OF DIGITAL OUTPUTS

In order to control the states on digital outputs, call the function [method]:

```
◆ VOID digital_output_len(const LENDEVICE hdevice,
    const DWORD outputs)
  [void digital_output(const unsigned long outputs)]
  [template<int N>
  void digital_output(const std::bitset<N>& outputs)]
```

The parameter `outputs` controls the digital outputs. The output value of `DOUT0` sets a high state on the DOUT0 output, likewise a `DOUT1` value sets a high state on the DOUT1 output, etc. There are low states on the digital outputs not listed. In order to set DOUT0 and DOUT1 outputs and clear the other outputs, call the function with values:

```
digital_output_len(hdev, DOUT0 | DOUT1)
```

In order to set a few outputs, call the function [method]:

```
◆ VOID hi_digital_output_len(const LENDEVICE hdevice,
    const DWORD outputs)
  [void hi_digital_output(const unsigned long outputs)]
  [template<int N>
  void hi_digital_output(const std::bitset<N>& outputs)]
```

In order to clear a few outputs, call the function [method]:

```
◆ VOID lo_digital_output_len(const LENDEVICE hdevice,
    const DWORD outputs)
  [void lo_digital_output(const unsigned long outputs)]
  [template<int N>
  void lo_digital_output(const std::bitset<N>& outputs)]
```

There are listed constant of digital outputs to clear in the parameter outputs in file “lendev.h”.

There are added a methods in the class `len::DeviceRS232`, which have parameters of type `std::bitset<>`, where the number of bit is the number of digital input / output of the device.

In “lendev.h” and “lendev.hpp” files were declared [defined] the functions [methods], which return the result of the success of operation:

```
◆ BOOL DigitalOutputLen(const LENDEVICE hdevice, const DWORD outputs)
  [bool DigitalOutput(const unsigned long outputs)]
  [template<int N>
  bool DigitalOutput(const std::bitset<N>& outputs)]
```

- BOOL HiDigitalOutputLen(const LENDEVICE hdevice,
  const DWORD outputs)
  [bool HiDigitalOutput(const unsigned long outputs)]
  [template<int N>
  bool HiDigitalOutput(const std::bitset<N>& outputs)]
- BOOL LoDigitalOutputLen(const LENDEVICE hdevice,
  const DWORD outputs)
  [bool LoDigitalOutput(const unsigned long outputs)]
  [template<int N>
  bool LoDigitalOutput(const std::bitset<N>& outputs)]

Example 7

```
#include "lendev.hpp"
#include <iostream>
#include <iomanip>
#include <sstream>

len::DeviceRS232 dev;
using namespace std;

int main()
{
  if ( dev.open(1) )
  {
    cout << "The device is opened\n";
    // test of digital inputs
    unsigned long inputs = dev.digital_input();
    cout << "Digital inputs: returned value = " << inputs << '\n';
    cout << "The states on particular inputs: "
      << bitset<8>(inputs) << '\n';
    cout << "The states on particular inputs: \n";
    for (int no_of_input = 0; no_of_input < 8;
      ++no_of_input )
    {
      cout << '\t' << "DIN" << no_of_input << " = "
        << ( (inputs & (1ul << no_of_input)) ? "HI" : "LO")
        << '\n';
    }
    // checking, if there is low state on DIN3 i DIN7 inputs
    if ( dev.is_lo_digital_input(DIN3 | DIN7) )
    {
      cout << "There is low state on DIN3 and DIN7 inputs\n";
    }

    // controlling the digital outputs
    // set DOUT7, DOUT3, DOUT1 outputs
    // and clear the others (that is DOUT6, DOUT5, DOUT4, DOUT2, DOUT0)
    dev.digital_output(bitset<8>(string("10001010")));

    // clear DOUT7 and DOUT1 outputs of the device
    // and others are without changing
    dev.lo_digital_output(DOUT7 | DOUT1);

    // set DOUT7 and DOUT3 outputs of the device
    // and others are without changing
    dev.hi_digital_output(bitset<8>(string("10001000")));
```

Example 7

```
    // use of methods, which return the result of success of operation
    for (int test = 0; test < 20; ++test )
    {
      bool result;
      if( dev.IsHiDigitalInput(bitset<8>(string("00100000")),&result) )
      {
        ostringstream s;
        s << "Test no " << test + 1 << ": ";
        cout << setiosflags(ios_base::left) << setw(13)
          << s.str() << resetiosflags(ios_base::adjustfield)
          << "On DOUT5 digital output is "
          << (result ? "HI" : "LO") << " state\n";
      }
      else
        cout << "The operation wasn't correct. ”
          << "Error code " << dev.get_last_error() << '\n';
      Sleep(1000);
    }
  }
  else
    cout << "Unable to establish connection\n";
  system("PAUSE");
  return 0;
}
```

2.18. CONTROLLING THE COUNTER

In order to read number of counted pulses by the counter on the device, call the function [method]:

- `DWORD get_counter_len(const LENDEVICE hdevice, const INT number)`
  `[unsigned long get_counter(const int number)]`

The parameter `number` is the number of the counter. The function [method] returns number of counted pulses.

In order to clear of the counter, call the function [method]:

- `VOID clear_counter_len(const LENDEVICE hdevice, const INT number)`
  `[void clear_counter(const int number)]`

where the parameter `number` is the number of counter.

The function [method]:

- `BOOL GetCounterLen(const LENDEVICE hdevice, const INT number,`
  `DWORD* result_ptr)`
  `[bool GetCounter(const int number, unsigned long* result_ptr)]`

returns the result of the success of operation, and the number of counted pulses is stored in variable with address `result_ptr`.

The function [method]:

- `BOOL ClearCounterLen(const LENDEVICE hdevice, const INT number)`
  `[bool ClearCounter(const int number)]`

clears the counter and returns the result of the success of operation.

2.19. GETTING THE SERIAL NUMBER

Every device of the Lentec Design company has an individual serial number, which can be read as a `DWORD` value (4 bytes). In order to read the serial number of the device, call the function [method]:

- `DWORD get_serial_number_len(const LENDEVICE hdevice)`
  `[unsigned long get_serial_number()]`

The function [method] returns the serial number of the device. You can call also the function [method]:

- `BOOL GetSerialNumberLen(const LENDEVICE hdevice, DWORD* result_ptr)`
  `[bool GetSerialNumber(unsigned long* result_ptr)]`

which returns the result of the success of operation, and the value if the serial number is stored in variable with address `result_ptr`.

2.20. GETTING INFORMATION ABOUT COM PORT OF COMPUTER

You should check the hardware parameters of COM port before its use. In order to get the information about the COM port of the computer, call the function declared in "lendev.h" file:

- ```
  BOOL check_accessibility_com_len(const INT port, INT* problems_ptr,
    BaudRate* max_baudrate_ptr,INT* number_of_available_baudrate_ptr,
    BOOL table_of_available_baudrate[11])
  ```

or in "lendev.hpp" file:

- ```
  bool len::check_accessibility_com(const int port,
    int *const problems_ptr, BaudRate *const max_baud_rate_ptr,
    int *const number_of_available_baud_rate_ptr,
    bool table_of_available_baud_rate[11])
  ```

The parameter `port` is the number of the COM port: COM1 = `1`, COM2 = `2`, etc. It must be: $1 \leq port \leq 256$. In variable with address `problems_ptr`, is stored the information about identified problems:

- `HARDWARE_NO_OPERATED_9600_BAUDRATE`: the UART chip of the computer does not provide default startup baud rate 9600 b/s and in order to communicate with the device you should change the startup baud rate (e.g. using another computer, which provides this baud rate).
- `HARDWARE_ONLY_WITH_EXTERNAL_SOURCE`: the UART chip of the computer enables only communication in Handshaking mode and the RTS and DTR lines can not be used to supply the LENDEVICE80RS232 device through COM port of the computer. The device can be supplied only from external power source (see Chap. 1.6: "Power supply of the device" on p. 8).
- `HARDWARE_CRITICAL_ERROR`: the UART chip of the computer makes the

communication with the device impossible (e.g. makes transmitting 8 bits data impossible). This error seldom occurs.

If no error occurs, the variable has value `HARDWARE_NO_PROBLEM = 0`. If an error occurs, you can identify it making an operation of binary conjunction (see example 8). In the variable with address `max_baud_rate_ptr`, is stored the value of the maximal baud rate, which the UART chip provides. In variable, which has `number_of_available_baud_rate_ptr` address, is stored the number of available baud rates, which the UART chip provides. The maximal value of this parameter is 11. In table with address `table_of_available_baudrate`, are stored the baud rates, which are provided by the UART chip. In order to check for example if 19200 b/s baud rate is available, you can do:

```
if ( table_of_available_baudrate[BR_19200] != FALSE )
{
  // the 19200 b/s baud rate is available
}
```

The functions `check_accessibility_com_len` and `len::check_accessibility_com` return the result of the success of operation. If these functions return `FALSE` or `false`, the system doesn't give the information (because e.g. the handle to the COM port is busy by another program) and returned parameters are incorrect.

Example 8.

```
#include "lendev.hpp"
#include <iostream>

using namespace std;

int main()
{
  int port = 1;
  cout << "Give the number of port, which you want to check: ";
  cin >> port;
  int problems;
  BaudRate maximal_available_baud_rate;
  int number_of_available_baud_rates;
  bool table_of_available_baud_rates[BR_MAX_AVAILABLE];
  if ( len::check_accessibility_com(port, &problems,
    &maximal_available_baud_rate, &number_of_available_baud_rates,
    &table_of_available_baud_rates[0]) )
  {
    if ( (problems & HARDWARE_CRITICAL_ERROR) != 0 )
      cout << "The UART chip doesn't provide communication with "
        << "the device\n";
    else
    {
      if ( problems != HARDWARE_NO_PROBLEM )
      {
        cout << "Problems:\n";
        if ( (problems & HARDWARE_NO_OPERATED_9600_BAUDRATE) != 0 )
          cout << "The UART chip doesn't provide 9600 b/s "
            << "baud rate \n";
        if ( (problems & HARDWARE_ONLY_WITH_EXTERNAL_SOURCE) != 0 )
```

Example 8.

```
            cout << "The UART chip of the computer doesn't enable "
              << "supply "
              << "the device\n";
        }
        const long baud_rate[11] = { 300, 600, 1200, 2400, 4800, 9600,
          14400, 19200, 38400, 57600, 115200 };
        cout << "Number of available baud rates = "
          << number_of_available_baud_rates << " from 11.\n";
        cout << "Maximal available baud rate = "
          << baud_rate[maximal_available_baud_rate]
          << " b/s.\n";
        cout << "Moreover the UART chip enables communication with the "
          << "baud rates:\n";
        for (int i = 0; i < maximal_available_baud_rate; ++i)
        {
          if ( table_of_available_baud_rates[i] )
            cout << "\t - " << baud_rate[i] << " b/s\n";
        }
      }
    }
    else
      cout << "Unable to get the information about COM" << port
        << " port of the computer.\n";
    system("PAUSE");
    return 0;
}
```

2.21. OPENLENDEVICE PROGRAM

The OpenLenDevice program, provided with the software on CD, serves to turn on the LENDEVICE80RS232 device and it is shown on Fig. 6.

The procedure of connecting the device to the computer:

1. Run OpenLenDevice program.
2. Set the number of COM port and the startup baud rate (the default setting is 9600 b/s baud rate).
3. Connect the device to the COM port of the computer.
4. Click button to connect the device with the computer.
5. If the points 1 – 4 were done correct, the button should change to green colour (like on Fig. 6) and on the device should light a LED diode.
6. Close the OpenLenDevice program. The device will be supplied for the current session of the Windows system.

A lighted LED diode on the device signals, that the device is supplied. You can now connect to the LENDEVICE80RS232 device the other devices and chips. If the LENDEVICE80RS232 device uses an external power source, it is unnecessary to use this program.

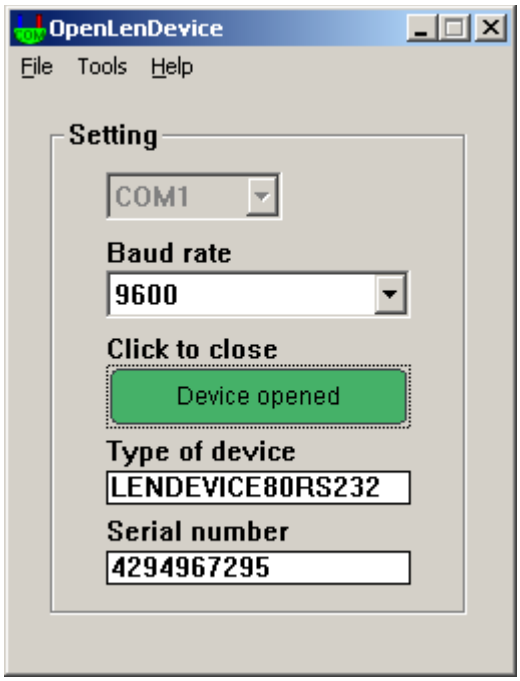


Fig. 6. *OpenLenDevice program.*

The program is bilingual (change the language: Tools → Options). The program enables set the new startup baud rate of the LENDEVICE80RS232 device (Tools → Set startup baud rate) and load a new main program to the microcontroller of the device (Tools → Upgrade device, see Chap. 2.13: "Programming the device" on p. 20).

2.22. LENOSCILLOSCOPE PROGRAM

The LenOscilloscope program given with the software on CD serves to control the LENDEVICE80RS232 device. It can control analog and digital outputs, the counter, measure the analog voltage on inputs and read the states on digital inputs. The measured analog voltage(s) are displayed on the graph and their values (maximal 10000) can be stored in a text file.

Using the program you can change: number of displayed points on the graph, the method of controlling the digital outputs, the format of displayed value in edit controls (hexadecimal and decimal) and change the language (Tools → Options). The program enables setting of a new startup baud rate of the LENDEVICE80RS232 device (Tools → Set startup baud rate) and loading a new main program (firmware) to the microcontroller of the device (Tools → Upgrade device, see Chap. 2.13: "Programming the device" on p. 20).

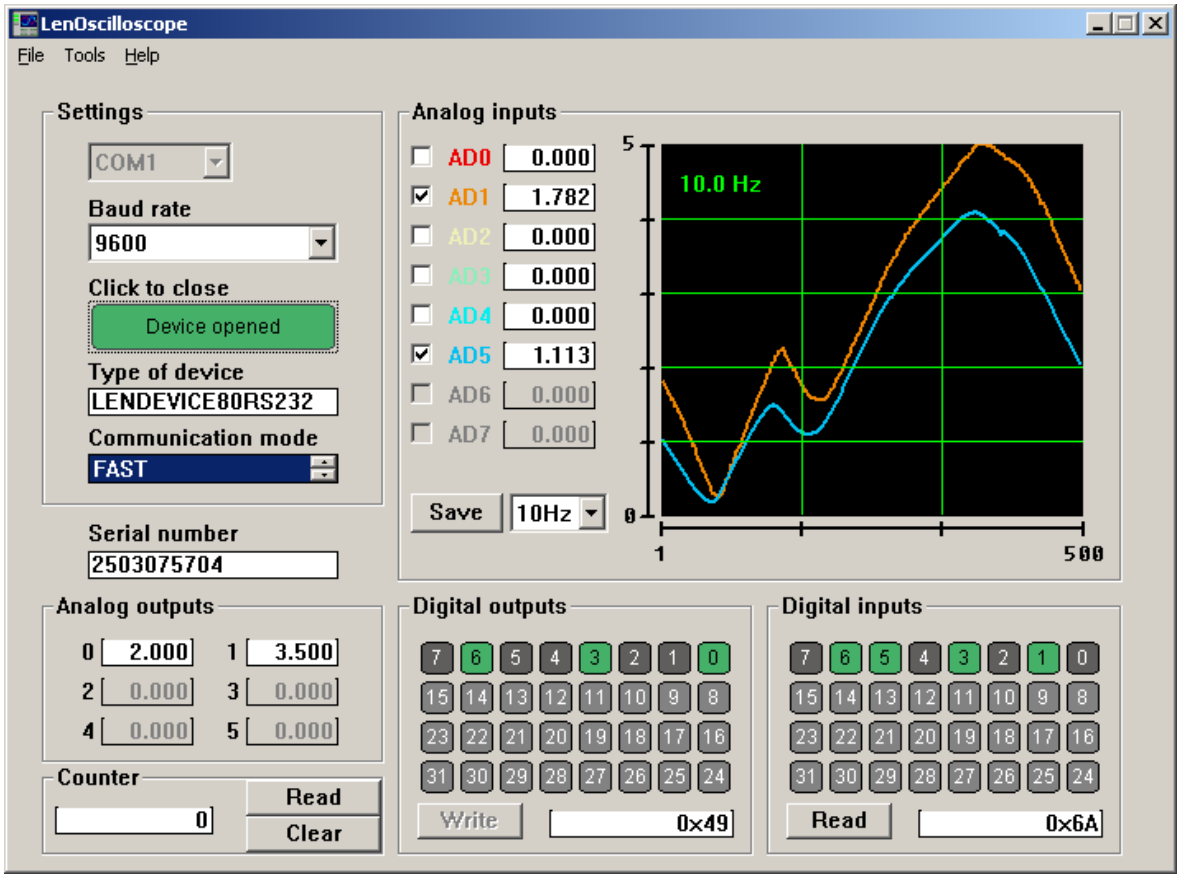


Fig. 7. *LenOscilloscope program.*

APPENDICES

Notes: Exceeding the 'Max' values stated or shorting to ground the device outputs, may cause permanent damage to the device.

APPENDIX A: DESIGNATIONS

GND – Ground signal of the device

VCC – Voltage of power supply

VDD – typically 4,997 V ≈ 5 V

APPENDIX B: ELECTRICAL CHARACTERISTICS ON THE RS PLUG-IN SOCKET

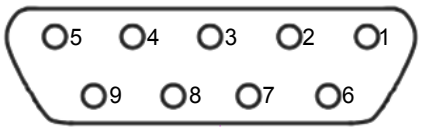


Fig. B1. *The layout of the plug-in socket, RS.*

Tab. B1. *Electrical characteristic of lines.*

No of line	Symbol of line	Function of line	Electrical characteristics: power supply from RS plug-in socket	Electrical characteristics: power supply from external source
1	DCD	output	not used	The same DTR
2	RD	output	Max ±14 V (1)	Max ±14 V (3)
3	TD	input	Max ±14 V	Max ±14 V
4	DTR	input	Max ±14 V (2)	Max ±14 V
5	SG	signal ground	0 V	0 V
6	DSR	output	not used	The same DTR
7	RTS	input	Max ±14 V (2)	not used
8	CTS	output	not used	The same DTR
9	RI	not used	–	–

(1) Output voltage is not higher than DTR and RTS inputs
(2) The difference of voltages between DTR and RTS inputs must be less than 0.1 V, and the current should not exceed 50 mA per input.
(3) The voltage is less than the voltage of the power supply.

Alternative spacing of the transmission lines, which can be in COM port are presented in Table B2. If your computer has different typical spacing, you will have to use a converter to change the spacing.

Tab. B2. *Untypical spacing of lines in DB9 and DB25 plug-in socket of COM port of the computer.*

Symbol of line	DB9	DB9	DB25	DB25
DCD	1	5	8	15
RD	3	4	3	5
TD	5	3	2	3
DTR	7	2	20	14
SG	9	1	7	13
DSR	2	9	6	11
RTS	4	8	4	7
CTS	6	7	5	9
RI	8	6	22	18

APPENDIX C: CHARACTERISTICS OF ANALOG INPUTS

Parameter	Value / Range
Maximum input voltage	GND – 0.5 V to VDD + 0.5 V
Resolution	10 bits
Absolute accuracy	typical 1 LSB
Range of measured voltage	GND to (1023/1024) VDD
Maximum output resistance of measured source	10 kΩ

APPENDIX D: CHARACTERISTICS OF ANALOG OUTPUTS

Parameter	Value / Range
Output voltage	typical: GND + 5 mV to VDD – 5 mV
Resolution	10 bits
Absolute accuracy	typical 1 LSB
Typical output resistance	50 Ω
Maximum current, which can flow through the output	±10 mA
Typical time of establishing output voltage after the change	50 ms

APPENDIX E: CHARACTERISTICS OF DIGITAL AND COUNTER INPUTS

Parameter	Value / Range
Maximum input voltage range	GND – 0.5 V to VDD + 0.5 V
Maximum input voltage, which is read as LOW state $V_{IN\ LOW}$	1.3 V
Minimal input voltage, which is read as HIGH state $V_{IN\ HIGH}$	3.7 V
Input resistance	typical 1 MΩ

APPENDIX F: CHARACTERISTICS OF DIGITAL OUTPUTS

Parameter	Value / Range
Maximum source / sink of the current	±5 mA
Minimum output voltage for HIGH state $V_{OUT\ HIGH}$	3.8 V
Maximum output voltage for LOW state $V_{OUT\ LOW}$	0.5 V

APPENDIX G: ADDING A NEW PATH TO THE DIRECTORY, WHICH ARE SCANNED IN INTEGRATED DEVELOPMENT ENVIRONMENTS (IDE)

Visual C++ 2005	Options → Projects and solutions → VC++ directories: ● files of type *.lib → Library files ● file of type *.h, *.hpp → Include files
Visual C++ 6.0	Options → Directories: ● file of type *.lib → Library files ● file of type *.h, *.hpp → Include files
Dev – C++ 4.9.2.2	Options of compiler → Directories: ● file of type *.lib → Libraries ● file of type *.h, *.hpp → Header Files C++
Code::Blocks v1.0	Settings → Compiler Settings → Directories: ● file of type *.lib → Linker ● file of type *.h, *.hpp → Compiler

After adding the path to the directories in IDE, you can include all files like standard files and without giving full path to the file. For example instead of writing:

```
#include ”C:\Program Files\Lentec Design\Interface\lendev.hpp”
```

you can write:

```
#include ”lendev.hpp”
```

APPENDIX H: ADDING THE STATIC LIBRARY (*.LIB) TO THE PROJECT

Visual C++ 2005	● Project → Properties → Linker → Input → Additional Dependencies → write the name of file (best with full path in “”) ● In tabbed window „Solution Explorer” → Add → Existing Item → choose a file *.lib
Visual C++ 6.0	● Project → Settings → Link → Overlap General → Object/Library modules → write the name of file (best with full path in “”) ● In tabbed window „File View” → Add File to Project → Library Files → choose a file *.lib
Dev – C++ 4.9.2.2	● Project → Options of project → Parameters → Consolidation → push the button „Add file” and you should choose a file
Code::Blocks v1.0	● Project → „Project's Build Options → Overlap „Linker” → Push the button „Add” and choose a library

APPENDIX I: INFORMATION FOR DEVELOPERS

In order to connect with the device, you have to use “lendev.dll” library and lenR232.exe driver. Both files have to be in the same folder. Typically these files are installed in system directory (the API function GetSystemDirectory() returns the path to the system directory). The lendev.dll library exports the function with __stdcall convention:

- open_len
- open_with_baudrate_len
- close_len
- is_open_len
- get_last_error_len
- change_baudrate_len
- set_startup_baudrate_len
- change_communication_mode_len
- reset_device_len
- info_len
- check_device_len
- program_device_len
- digital_output_len
- hi_digital_output_len
- lo_digital_output_len
- digital_input_len
- is_hi_digital_input_len
- is_lo_digital_input_len
- analog_input_len
- analog_output_len
- get_counter_len
- clear_counter_len
- get_serial_number_len
- check_accessibility_com_len
- DigitalOutputLen
- HiDigitalOutputLen
- LoDigitalOutputLen
- DigitalInputLen
- IsHiDigitalInputLen
- IsLoDigitalInputLen

- AnalogInputLen
- AnalogOutputLen
- GetCounterLen
- ClearCounterLen
- GetSerialNumberLen
- type_of_device_len

The lendev.dll library uses kernel32.dll library, and lenR232.exe driver uses kernel32.dll and lendev.dll libraries.

APPENDIX J: DIMENSIONS OF THE DEVICE

The device has the dimensions like on Fig. J1 and Tab. J1.

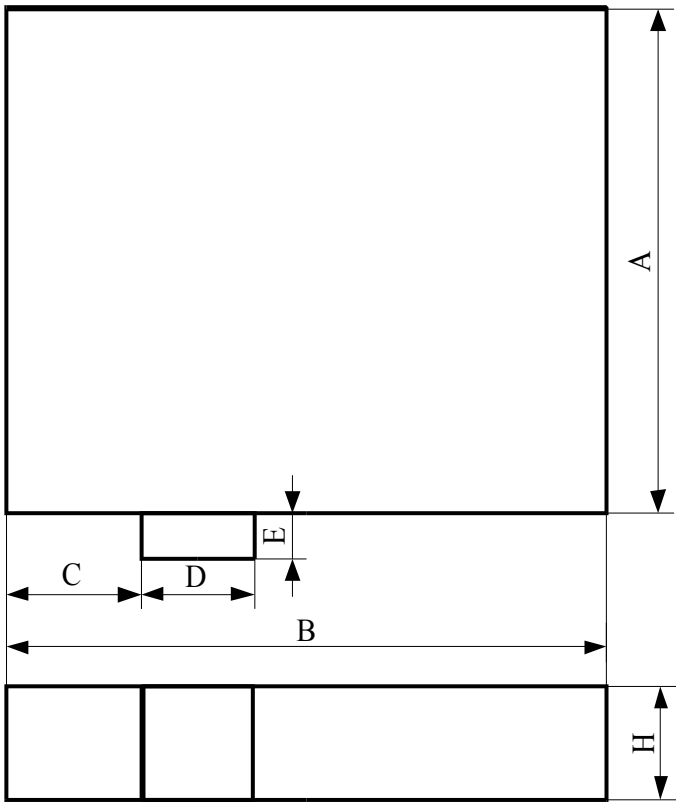


Fig. J1. *The view of outer dimensions of the LENDEVICE80RS232 device.*

Tab. J1. *The outer dimensions of the LENDEVICE80RS232 device [mm].*

A	B	C	D	E	H
80	100	18	30	6	24

APPENDIX K: THE LENDEV.H AND LENDEV.HPP FILES

```
// lendev.h:
// (most important fragments)

#include <windows.h>

#ifndef DLL_IMPORT_DEVICERS232MOD1_LEN_H
#define DLL_IMPORT_DEVICERS232MOD1_LEN_H extern "C" __declspec(dllimport)
#endif

#define I_L(TYPE) DLL_IMPORT_DEVICERS232MOD1_LEN_H TYPE __stdcall

typedef LPVOID LENDEVICE;
#define NO_LENDEVICE ((LENDEVICE) 0)

enum DeviceMode { DEVICE_OFF = 0, DEVICE_ON = 1, DEVICE_BAD_TRANSMISSION = 2,
  DEVICE_WITHOUT_PROGRAM = 3 };

enum CommunicationMode { COMM_FAST_MODE = 0, COMM_CONTROLLED_MODE = 1 };

enum BaudRate { BR_300 = 0, BR_600 = 1, BR_1200 = 2, BR_2400 = 3, BR_4800 = 4,
  BR_9600 = 5, BR_14400 = 6, BR_19200 = 7, BR_38400 = 8, BR_57600 = 9, BR_115200 = 10,
  BR_MAX_AVAILABLE = 11 };

I_L( LENDEVICE )  open_len(const INT port);

I_L( LENDEVICE )  open_with_baudrate_len(const INT port, const BaudRate baudrate);

I_L( BOOL )       close_len(const LENDEVICE hdevice);

I_L( BOOL )       is_open_len(const LENDEVICE hdevice, const BOOL check_obligatorily);

I_L( INT )        get_last_error_len(const LENDEVICE hdevice);

I_L( BOOL )       change_baudrate_len(const LENDEVICE hdevice,
                    const BaudRate new_baudrate);

I_L( BOOL )       set_startup_baudrate_len(const LENDEVICE hdevice,
                    const BaudRate new_baudrate);

I_L( BOOL )       change_communication_mode_len(const LENDEVICE hdevice,
                    const CommunicationMode new_mode);

I_L( BOOL )       reset_device_len(const LENDEVICE hdevice);

I_L( INT )        type_of_device_len(const LENDEVICE hdevice);

I_L( VOID )       info_len(const LENDEVICE hdevice, BYTE information[40]);

I_L( DeviceMode ) check_device_len(const LENDEVICE hdevice, const INT how_many);

I_L( BOOL )       program_device_len(const LENDEVICE hdevice, const BYTE* file_ansi);

I_L( VOID )       digital_output_len(const LENDEVICE hdevice, const DWORD outputs);

I_L( VOID )       hi_digital_output_len(const LENDEVICE hdevice, const DWORD outputs);

I_L( VOID )       lo_digital_output_len(const LENDEVICE hdevice, const DWORD outputs);
```

```
I_L( DWORD )      digital_input_len(const LENDEVICE hdevice);

I_L( BOOL )       is_hi_digital_input_len(const LENDEVICE hdevice, const DWORD inputs);

I_L( BOOL )       is_lo_digital_input_len(const LENDEVICE hdevice, const DWORD inputs);

I_L( VOID )       analog_output_len(const LENDEVICE hdevice, const INT channel,
                    const FLOAT value);

I_L( FLOAT )      analog_input_len(const LENDEVICE hdevice, const INT channel);

I_L( DWORD )      get_counter_len(const LENDEVICE hdevice, const INT number);

I_L( VOID )       clear_counter_len(const LENDEVICE hdevice, const INT number);

I_L( DWORD )      get_serial_number_len(const LENDEVICE hdevice);

I_L( BOOL )       DigitalOutputLen(const LENDEVICE hdevice, const DWORD outputs);

I_L( BOOL )       HiDigitalOutputLen(const LENDEVICE hdevice, const DWORD outputs);

I_L( BOOL )       LoDigitalOutputLen(const LENDEVICE hdevice, const DWORD outputs);

I_L( BOOL )       DigitalInputLen(const LENDEVICE hdevice, DWORD* inputs_ptr);

I_L( BOOL )       IsHiDigitalInputLen(const LENDEVICE hdevice, const DWORD inputs,
                    BOOL* result_ptr);

I_L( BOOL )       IsLoDigitalInputLen(const LENDEVICE hdevice, const DWORD inputs,
                    BOOL* result_ptr);

I_L( BOOL )       AnalogOutputLen(const LENDEVICE hdevice, const INT channel,
                    const FLOAT value);

I_L( BOOL )       AnalogInputLen(const LENDEVICE hdevice, const INT channel,
                    FLOAT* inputs_ptr);

I_L( BOOL )       GetCounterLen(const LENDEVICE hdevice, const INT number,
                    DWORD* result_ptr);

I_L( BOOL )       ClearCounterLen(const LENDEVICE hdevice, const INT number);

I_L( BOOL )       GetSerialNumberLen(const LENDEVICE hdevice, DWORD* result_ptr);

I_L( BOOL )       check_accessibility_com_len(const INT port, INT* problems_ptr,
                    BaudRate* max_baudrate_ptr, INT* number_of_available_baudrate_ptr,
                    BOOL table_of_available_baudrate[11]);

/* for function: check_accessibility_com_len(...), see documentation */
#define HARDWARE_NO_PROBLEM                          0x000
#define HARDWARE_NO_OPERATED_9600_BAUDRATE           0x001
#define HARDWARE_ONLY_WITH_EXTERNAL_SOURCE           0x002
#define HARDWARE_CRITICAL_ERROR                      0x004

/* returns by get_last_error_len */
#define ERROR_SUCCESS_LEN             0
#define ERROR_SYSTEM_PROBLEM_LEN      1
#define ERROR_INVALID_PARAMETERS_LEN  2
#define ERROR_COMMUNICATION_LEN       3
```

```
#define ERROR_INVALID_INSTRUCTION_LEN 4
#define ERROR_BAD_HANDLE_LEN          -1

/* for function: type_of_device_len(...) */
#define LENTECDEVICE_NOOPENED    0
#define LENTECDEVICE80RS232     10
#define LENTECDEVICE320RS232    20
#define LENTECDEVICE1280RS232   30


/* names of digital outputs, for functions:
  digital_output_len(...), hi_digital_output_len(...), lo_digital_output_len(...)
  DigitalOutputLen(...), HiDigitalOutputLen(...), LoDigitalOutputLen(...) */
#define DOUT0   (1ul << 0)
#define DOUT1   (1ul << 1)
#define DOUT2   (1ul << 2)
#define DOUT3   (1ul << 3)
#define DOUT4   (1ul << 4)
#define DOUT5   (1ul << 5)
#define DOUT6   (1ul << 6)
#define DOUT7   (1ul << 7)
#define DOUT8   (1ul << 8)
#define DOUT9   (1ul << 9)
#define DOUT10  (1ul << 10)
#define DOUT11  (1ul << 11)
#define DOUT12  (1ul << 12)
#define DOUT13  (1ul << 13)
#define DOUT14  (1ul << 14)
#define DOUT15  (1ul << 15)
#define DOUT16  (1ul << 16)
#define DOUT17  (1ul << 17)
#define DOUT18  (1ul << 18)
#define DOUT19  (1ul << 19)
#define DOUT20  (1ul << 20)
#define DOUT21  (1ul << 21)
#define DOUT22  (1ul << 22)
#define DOUT23  (1ul << 23)
#define DOUT24  (1ul << 24)
#define DOUT25  (1ul << 25)
#define DOUT26  (1ul << 26)
#define DOUT27  (1ul << 27)
#define DOUT28  (1ul << 28)
#define DOUT29  (1ul << 29)
#define DOUT30  (1ul << 30)
#define DOUT31  (1ul << 31)

/* names of digital inputs, for functions:
  digital_input_len(...), is_hi_digital_input_len(...), is_lo_digital_input_len(...)
  DigitalInputLen(...), IsHiDigitalInputLen(...), IsLoDigitalInputLen(...) */
#define DIN0  (1ul << 0)
#define DIN1  (1ul << 1)
#define DIN2  (1ul << 2)
#define DIN3  (1ul << 3)
#define DIN4  (1ul << 4)
#define DIN5  (1ul << 5)
#define DIN6  (1ul << 6)
#define DIN7  (1ul << 7)
#define DIN8  (1ul << 8)
#define DIN9  (1ul << 9)
#define DIN10 (1ul << 10)
```

```
#define DIN11 (1ul << 11)
#define DIN12 (1ul << 12)
#define DIN13 (1ul << 13)
#define DIN14 (1ul << 14)
#define DIN15 (1ul << 15)
#define DIN16 (1ul << 16)
#define DIN17 (1ul << 17)
#define DIN18 (1ul << 18)
#define DIN19 (1ul << 19)
#define DIN20 (1ul << 20)
#define DIN21 (1ul << 21)
#define DIN22 (1ul << 22)
#define DIN23 (1ul << 23)
#define DIN24 (1ul << 24)
#define DIN25 (1ul << 25)
#define DIN26 (1ul << 26)
#define DIN27 (1ul << 27)
#define DIN28 (1ul << 28)
#define DIN29 (1ul << 29)
#define DIN30 (1ul << 30)
#define DIN31 (1ul << 31)

/* names of analog outputs, for functions:
   analog_output_len(...), AnalogOutputLen(...) */
#define AOUT0  0
#define AOUT1  1
#define AOUT2  2
#define AOUT3  3
#define AOUT4  4
#define AOUT5  5

/* names of analog inputs, for functions:
   analog_input_len(...), AnalogInputLen(...) */
#define AIN0  0
#define AIN1  1
#define AIN2  2
#define AIN3  3
#define AIN4  4
#define AIN5  5
#define AIN6  6
#define AIN7  7

/* name of counter, for functions:
  get_counter_len(...), clear_counter_len(...),
  GetCounterLen(...), ClearCounterLen(...) */
#define L0    0

#undef I_L
```

```
// lendev.hpp
// (most important fragments)

#include <string>

namespace len
{

class DeviceRS232
{
public:
DeviceRS232();

DeviceRS232(const int port);

DeviceRS232(const int port, const BaudRate baudrate);

~DeviceRS232();

bool          open(const int port);

bool          open_with_baudrate(const int port, const BaudRate baudrate);

bool          close();

bool          is_open(const bool check_obligatorily = false);

bool          change_baudrate(const BaudRate new_baudrate);

bool          set_startup_baudrate(const BaudRate new_baudrate);

bool          change_communication_mode(const CommunicationMode new_mode);

std::string   info();

int           type_of_device();

DeviceMode    check_device(const int how_many = 10);

bool          reset_device();

bool          program_device(const std::string& file_ansi);

unsigned long get_serial_number();

int           get_last_error();

void          digital_output(const unsigned long outputs);

template<int N> void digital_output(const std::bitset<N>& outputs);

void          hi_digital_output(const unsigned long outputs);

template<int N> void hi_digital_output(const std::bitset<N>& outputs);

void          lo_digital_output(const unsigned long outputs);

template<int N> void lo_digital_output(const std::bitset<N>& outputs);

unsigned long digital_input();
```

```
bool          is_hi_digital_input(const unsigned long  inputs);

template<int N> bool is_hi_digital_input(const std::bitset<N>& inputs);

bool          is_lo_digital_input(const unsigned long inputs);

template<int N> bool is_lo_digital_input(const std::bitset<N>& inputs);

float         analog_input(const int channel);

void          analog_output(const int channel, const float value);

unsigned long get_counter(const int number);

void          clear_counter(const int number);

bool          DigitalOutput(const unsigned long outputs);

template<int N> bool DigitalOutput(const std::bitset<N>& outputs);

bool          HiDigitalOutput(const unsigned long outputs);

template<int N> bool HiDigitalOutput(const std::bitset<N>& outputs);

bool          LoDigitalOutput(const unsigned long outputs);

template<int N> bool LoDigitalOutput(const std::bitset<N>& outputs);

bool          DigitalInput(unsigned long* inputs_ptr);

bool          IsHiDigitalInput(const unsigned long  inputs, bool* result_ptr);

template<int N> bool IsHiDigitalInput(const std::bitset<N>& inputs, bool* result_ptr);

bool          IsLoDigitalInput(const unsigned long  inputs, bool* result_ptr);

template<int N> bool IsLoDigitalInput(const std::bitset<N>& inputs, bool* result_ptr);

bool          AnalogInput(const int channel, float* result_ptr);

bool          AnalogOutput(const int channel, const float value);

bool          GetCounter(const int number, unsigned long* result_ptr);

bool          ClearCounter(const int number);

bool          GetSerialNumber(unsigned long* result_ptr);

private:
  LENDEVICE lendevice;
};  // end of class "DeviceRS232"

bool        check_accessibility_com(const int port, int *const problems_ptr,
              BaudRate *const max_baud_rate_ptr,
              int *const number_of_available_baud_rate_ptr,
              bool table_of_available_baud_rate[11]);

} // end of namespace "len"
```

Lentec Design
Poland
www.lentecdesign.com
Office: office@lentecdesign.com
Technical Support: support@lentecdesign.com