A family of universal modules of the ADC, DAC

E20-10

External high-speed module on the USB 2.0 bus

Programmer manual

Revision A3
March 2011

L-CARD

http://en.lcard.ru
en@lcard.ru

DAQ SYSTEMS DESIGN, MANUFACTURING & DISTRIBUTION

L-CARD LLC

117105, Moscow, Varshavskoye shosse, 5, block 4, bld. 2

tel.: (495) 785-95-19
fax: (495) 785-95-14

Internet contacts:

http://en.lcard.ru

E-Mail:

Sales department: en@lcard.ru
Technical support: en@lcard.ru

E20-10. External high-speed general-purpose module on the USB 2.0 bus.

Contents

1. Introduction

This description is designed for users who intend to develop their own applications in the operating environment *Windows'98 / 2000 / XP / Vista / 7* for working with high-speed modules *E20-10* from LLC "L-Card". It is strongly recommended to review "E20-10. User guide", where you can find detailed technical information about the module, including function circuit description, injection signal connexion, external connectors pinning, characteristic fails and many others.

"L-Card" LLC company supplies USB device drivers, ready dynamic link library **Lusbapi** with a whole range of completed samples for the high-speed module `E20-10`. As the base language, when writing the `Lusbapi` library, C ++ was selected, and more specifically, the old, reliable **Borland C ++ 5.02**. Moreover, the library itself and all examples are supplied along with the source code, provided with fairly detailed comments. The standard library `Lusbapi` includes a variety of functions that help the user to use all the features incorporated in the *E20-10* module.

The *E20-10* module was developed with the main goal to provide reliable high-speed collection of analog information to the computer. To this end, the standard library `Lusbapi` contains a range of functions that allows organizing multi-channel *continuous streaming* of analog data at ADC frequencies up to 10 MHz. When collecting analog information, the end user can use a wide range of types of data entry synchronization. The output of analogue (on DAC) and digital information input / output is realized only in a single, and therefore relatively slow, mode. We hope that the `Lusbapi` library described below will simplify and speed up the writing of your own *Windows* applications. The entire package of standard software for working with the *E20-10* module in the *Windows'98 / 2000 / XP / Vista / 7* is found on the supplied firmware CD-ROM in the `\ USB \ Lusbapi`. **!!!ATTENTION!!!** Further, in the text of this description, all the directories are indicated relative to it. Also all the regular software can be downloaded from our website en.lcard.ru from the section "File Library". There, from the "Software for Windows", you should select the self-extracting archive `lusbapi`***`XY`***`.exe`, where ***X.Y*** denotes the version number of the software. At the time of this writing, the latest *Lusbapi* library has version **3.3**, and its archive is called lusbapi33.exe.

2. General information

2.1. What's new?

As a rule, this paragraph will contain only main changes as a hardware and software nature. For more information, please refer to:

- "E20-10. User Manual";
- "E20-10. Library Lusbapi. Additions and changes log".

2.1.1. Library Lusbapi 3.3

In the `Lusbapi` library version **3.3**, only two minor changes were made, namely:

- *E20-10* module is available in two versions (designs):
  - ✓ with a bandwidth of the input signal equal to 1.25 MHz (basic version);
  - ✓ with a bandwidth of the input signal equal to 5.0 MHz;

  In order to inform the user about the current execution of the module, a new numeric Modification field was entered in the MODULE_INFO_LUSBAPI structure.
- For the module *E20-10 (Rev.'A ')* in the function ReadData (), the lower limit and the multiplicity of the NumberOfWordsToPass request value of the IO_REQUEST_LUSBAPI structure are corrected. Before these values were equal to 128counts, now it is 256.

2.1.2. Library Lusbapi 3.2

In early 2008, a new revision of the *E20-10 module (Rev.'B ')* was started. This modification is the product of a solid hardware upgrade *E20-10 (Rev.'A ')*.

The `Lusbapi` library version **3.2** is designed to provide full support for all new functions and properties that appeared on the *E20-10 module (Rev.'B ')*. Since there are a lot of changes we note only the following main differences from the previous revision of the module:

- The procedure for calibrating the data from the ADC can now be performed at the FPGA level of the module;
- Extended range of interframe delay;
- Introduced advanced synchronization capabilities when working with ADC;
- Improved checking the status of the data collection process.

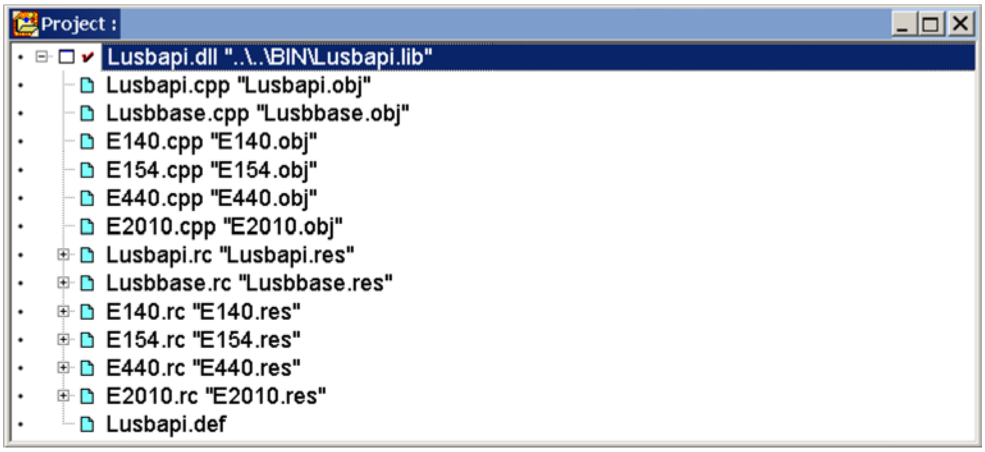
2.2. Connecting the E20-10 module to a computer

All details of the hardware connection procedure for the *E20-10* module to the end user's computer and the proper installation of the **USB** drivers can be found in "E20-10. User Manual, § 4 "Instalation and configuration".

It is worth emphasizing that, starting with version **3.2**, the main USB driver file has changed in the **Lusbapi** library. Now it is called **`Ldevusbu.sys`** instead of **`Ldevusb.sys`**. Thus, when upgrading from older versions of **Lusbapi** to a newer version **3.2** or higher, the end user should, through the *"Device Manager*", switch the *E20-10* module to work with the new **USB** driver.

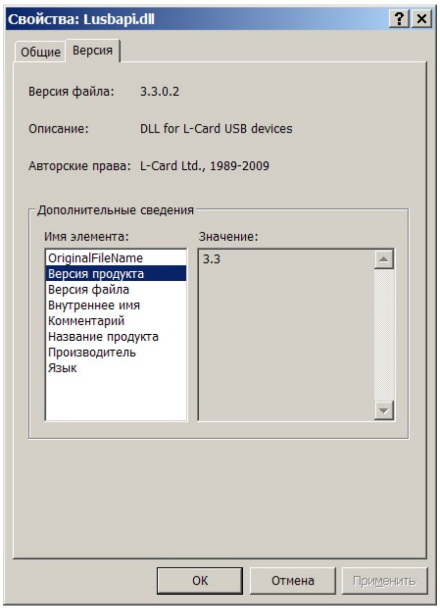
2.3. *Library Lusbapi*

The regular library `Lusbapi` is written using the very accessible programming language **Borland C ++ 5.02**. In addition to the *E20-10* module, the library also supports modules of the *E-154*, *E14-140* and *E14-440* types. The general view of the `Lusbapi` library project in the **Borland C ++ 5.02** integrated development environment is shown in the figure below:



The library itself contains only *two* exported functions, one of which CreateLInstance () returns a pointer to the interface of the *E20-10* module. Further, using this pointer, you can access all the interface functions of the standard DLL library (see the source code for the examples). **!!!Attention!!!** All interface functions, except for ReadData(), strictly speaking, do not provide *thread-safe*‖ operation of the library. Therefore, in order to avoid confusion, in multi-threaded applications, the user must organize himself, if necessary, correct synchronization of interface function calls in different threads (using, for example, critical sections, mutexes, etc.).

The library file `Lusbapi.dll` includes information about the current version of the DLL. To get information about this version in your application, you can use the second export function from the standard library: GetDllVersion(). In addition, to quickly identify the current version of the library can be using the regular features of *Windows*. For instance, right-click on the *Lusbapi.dll* library file in `_Windows Explorer'`. In the menu that pops up on the monitor screen, select the option _*Properties*', and then on the resulting panel select the _*Version* '*tab*. On this tab in the line _*File version* you can easily read the current version of the library. It looks something like this:



The file of the regular library Lusbapi.dll is located on the corporate CD-ROM in the \ DLL \ BIN directory. Its source texts can be found in the \ DLL \ Source \ Lusbapi directory. Header files are stored in the \DLL\ Include directory, and import libraries and declaration modules for various development environments can be found in the \DLL\Lib directory.

Texts of completed examples of application of interface functions from the standard DLL library for various application development environments can be found in the following directories:

- *E20-10*\Examples\Borland C++ 5.02;
- *E20-10*\Examples\Borland C++ Builder 5.0;
- *E20-10*\Examples\Borland Delphi 6.0;
- *E20-10*\Examples\MicroSoft Visual C++ 6.0.

For example, to get the ability to call interface functions in a custom project on Borland C ++, you need to do the following:

- create a project file (for example, for **Borland C ++ 5.02**, test.ide);
- add the import library file \DLL\Lib\Borland\LUSBAPI.LIB;
- create and add to your project your file with a future program (for example, test.cpp);
- include at the beginning of your file the header file #include "LUSBAPI.H", containing the interface description of the *E20-10* module;
- in principle, using the function GetDllVersion (), it is desirable to compare the version of the used DLL library with the version of the current software;
- call the CreateLInstance () function to get a pointer to the module interface; in general, **EVERYTHING!** Now you can write your program and at any place, using the received pointer, call the corresponding interface functions from the regular DLL of the library Lusbapi.dll.

To fans of the **Microsoft Visual C ++** dialect, you can recommend two ways to connect a standard DLL library to your application:

1. Dynamic load of the Lusbapi library at the application execution stage. For details, see the source code for the sample from the *E20-10*\Examples\MicroSoft Visual C ++ 6.0\DynLoad directory.
2. If you are statically building a standard DLL in your project, use the LUSBAPI.LIB import library file from the \DLL\Lib\MicroSoft directory.

While working with the *E20-10* module in the **Borland Delphi** environment, it is recommended to use the LUSBAPI.PAS declaration module located in the \ DLL\Lib\Delphi directory. Also, instead of the original ad unit, you can fully use the compiled version of LUSBAPI.DCU.

2.4. Microcontroller module

On the E20-10 module, as a _*workhorse*' a microcontroller (MCU) of AVR Atmega 162 type by Atmel Corporation is used. The MCU is responsible for the correct functioning of the **USB** interface of the module, as well as parses all user commands coming from the computer and specifying the various modes of operation of the module. A feature of software, which is the basis of the MCU's work, is its two-component. In other words, as it consists of two parts: the main program (Firmware) and the bootloader (BootLoader). The proprietary loader, as well as the main program, *is loaded* to the MCU during the setup phase of the E20-10 module in "L-Card" LLC and the end user does not have the option of updating it without a special firmware cable. But at the same time BootLoader provides the possibility of painless firmware reflashing of the module on the **USB** bus, which is extremely convenient when upgrading the main program. The latest version of Firmware MCU can always be downloaded from our website en.lcard.ru from the section "File Library". There, from the "Firmware and BIOS" subsection, select the archive `e2010fw_WXa_YZb.zip`, where **W.X** stands for the version number of the main MCU program for the *E20-10 module (Rev.'A ')*, and **Y.Z** for the E20-10 *module (Rev.'B')*. At the time of writing, this archive is named e2010fw_17a_21b.zip.zip.

2.5. Module loading

As one of the main functional units of the *E20-10* module, you can safely call the *programmable logic integrated circuit*(FPGA) of the ACEX family (for *Rev.'A' module)* or Cyclone (for *Rev.'B' module)* by Altera Corporation. The main functional purpose of the FPGA is to perform full hardware control of the streaming input of analog information. Applied FPGA has a so-called downloadable architecture. Thus, it must be loaded every time after power is applied to the module, and can also be reloaded already during the operation of the module.

In the operational software in the directory `\DLL\Source\` you can find the files of the firmware FPGA for various revisions of the module *E20-10*, namely:

- `E2010.pld` — for *E20-10 (Rev.'A')* module;
- `E2010m.pld` — for *E20-10 (Rev.'B')* module.

These files are also built in as resources in the library `Lusbapi.dll`, which has a special interface function LOAD_MODULE () to correctly load the firmware into the FPGA module. Only after loading the FPGA you can go directly to the very management of the module, i.e. shift it into various modes of operation with ADC, DAC, and so on.

2.6. Possible problems with the module

1. Before working with the regular *E20-10* module software, in order to avoid unpredictable behavior of the module, it is highly recommended to install the drivers for the chipset of the motherboard of the computer used. In particular, this applies to chipsets not from ***Intel*: *VIA***, ***SIS***, ***nVidia***, ***AMD+ATI*** and so on. Usually these drivers can be found on the company's CD-ROM, which comes with the motherboard. Also they can be downloaded from the Internet from the manufacturer's website.

2. Computers whose motherboard is based on the chipset from ***SIS*** (Silicon Integrated System Corporation), ***AMD*** + ***ATI*** (Advanced Micro Devices, Inc.) or ***nVidia*** (NVIDIA Corporation), do not work correctly on *Windows'98/2000/XP/Vista/7*. This is evident in queries with a large amount of data in the interface functions ReadData (). For example, if you call this function with the *NumberOfWordsToRead* = 1024 * 1024 parameter, the *Windows* operating system may well, it's called, hang '*tightly*' until the BSOD (Blue Screen Of Death) appears. The solution to this problem lies in the course of decreasing the value of *NumberOfWordsToRead*. And the value of *NumberOfWordsToRead*, in which everything starts working properly, depends on a specific instance of the computer. So you should try simply to modify the value of the *NumberOfWordsToRead* parameter.

3. Used terms and data formats

3.1. Terms

Name	Meaning
ADCRate	ADC frequency, *kHz*.
InterKadrDelay	Interframe delay, *mls*.
KardRate	Count frame frequency, *kHz*.
Buffer	Array of integers of data type *SHORT*.
ControlTable	A control table containing an integer array with *logical* channel numbers. Used by the equipment for organizing a cyclic interrogation of ADC channels during data collection.
ControlTableLength	The size of the control table.

3.2. Data formats

3.2.1. Word formate from the ADC data

The data coming from the 14 ^{bit} A/D converter of the *E20-10* module is represented in the format of the signed integer two byte number from -8192 to 8191. These *raw*‖ readings from the ADC are recommended to be adjusted, for example, using the *regular (factory)* correction coefficients stored in the module itself and accessible using the regular function GET_MODULE_DESCRIPTION (). The procedure for correcting the ADC data is possible both at the upper software level and at the FPGA level of the module, and is described in detail in § 4.5.1. "ADC data correction". The relationship between the corrected ADC code and the input voltage is given in the table below:

Table 1. Matching the corrected ADC code to the input voltage

Range, B	ADC code	Voltage, B
±3.0; ±1.0; ±0.3	+8000	+3.0; +1.0; +0.3
	0	0
	-8000	–3.0; –1.0; –0.3

3.2.2. Word format for DAC data

On the module, at the user's request, a 2$^{-channel}$ 12^{bit} DAC. To output any voltage at the output of the DAC to the *E20-10* module, a 16 ^{bit} data word must be transmitted. The format of this data word is given in the following table:

Table 2. Data word format for DAC

Bit number	Intended purpose
<11..0>	12-bit DAC code
12	DAC channel number: ‚0‘ – first channel; ‚1‘ – second channel.
<15..13>	Is not used

Actually, the code of the DAC itself is recommended to be corrected before sending it to the module. The correction coefficients are stored in the module and are accessible using the standard function GET_MODULE_DESCRIPTION(). The procedure for correcting the DAC data is described in detail in § 4.6.1. "ADC data correction". After this procedure, the corrected code sent by the module to the 12-bit DAC is connected to the voltage set on the external connector in accordance with the following table

Table 3. Correspondence of the corrected DAC code to the output voltage

DAC code	Voltage, B
+2047	+5.0
0	0
–2048	–5.0

3.2.3. Logical channel number

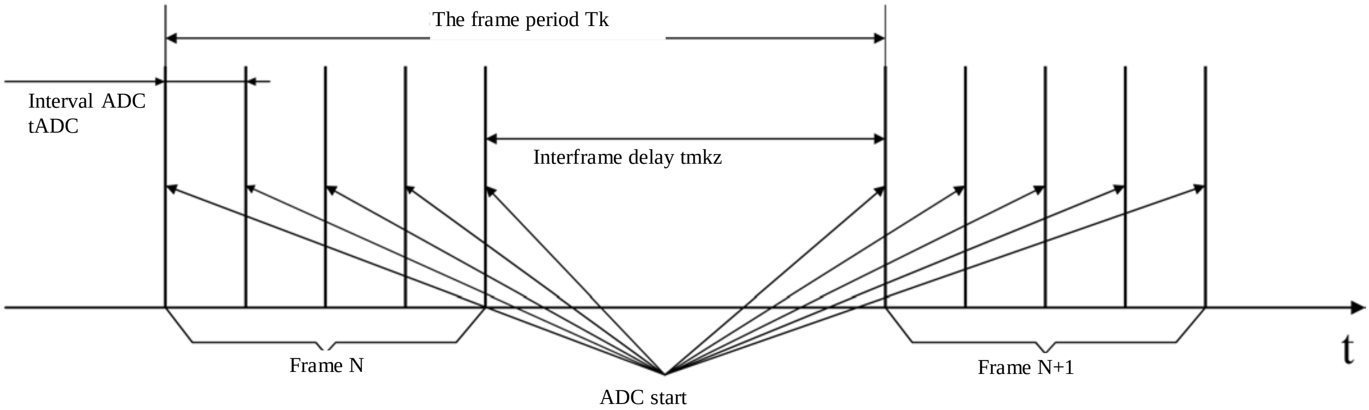
On the *E20-10* module, to control the operation of the analog input stage, a parameter is defined, such as a *16-bit logical channel number*. It is the array of logical channel numbers that forms the **ControlTable** control table that specifies the cyclic sequence of the ADC's operation when data is collected. For the *E20-10* module, the logical channel number contains only the actual physical number of the analog channel of the ADC. The bit-wise format of the logical channel number is shown in the table below:

Table 4. Logical channel number format

Bit fields	Designation	Application
<0..1>	**CH<0..1>**	Number of the ADC channel: ‚00‘ – first channel; ‚01‘ – second channel; ‚10‘ – third channel; ‚11‘ – forth channel.
<2..15>	———	Reserved

3.2.4. Frame format of counting

A frame is a sequence of counts from logical channels, from **ControlTable [0]** to **ControlTable [ControlTableLength-1]**, where **ControlTable** is the control table (array of logical channels), and **ControlTableLength** determines the size (length) of this table. The necessary control table can be loaded into the module using the interface function ***SET_ADC_PARS()*** (see § 4.5.4. "Module ADC operation parameter setting"). Time parameter of operating module frame **ControlTableLength = 5** are shown in the next figure:



where $\mathbf{T}_k$ is the time interval between adjacent frames (actually the frequency of the polling of the fixed logical number of the **KardRate** channel), $\mathbf{t}_{mkz}$ = **InterKadrDelay** is the time interval between the last count of the current frame and the first count of the next, $\mathbf{t}_{ADC}$ is the interval of the ADC start up or the interframe delay. Then $1/\mathbf{t}_{ADC}$ = **AdcRate** is the frequency of operation of the ADC or digitization of the data, and the value of $\mathbf{t}_{mkz}$ can not take values less than $\mathbf{t}_{ADC}$. If the *frame size*, i.e. the number of samples in the frame is equal to **ControlTableLength**, then all these time parameters can be related by the following formula:

$$\mathbf{T}_k = 1/\mathbf{KardRate} = (\mathbf{ControlTableLength} - 1) * \mathbf{t}_{ADC} + \mathbf{t}_{mkz},$$

or

$$\mathbf{T}_k = 1/\mathbf{KardRate} = (\mathbf{ControlTableLength} - 1)/\mathbf{AdcRate} + \mathbf{InterKadrDelay}.$$

The time parameters **AdcRate** and **InterKadrDelay** are used in the interface function SET_ADC_PARS () when specifying the required data collection mode.

4. Description of the Lusbapi library

This section provides a fairly detailed description of the constants, structures, and interface functions that make up the standard `Lusbapi` DLL library for the *E20-10 module*.

4.1. General principles of working with the module

The goal of the standard DLL library `Lusbapi`, supplied with the *E20-10* module, is to provide a fairly clear and user-friendly software interface when working with this device. The library contains a certain set of functions with which you can implement many standard algorithms of data I/O to/from the *E20-10* module. Before you start working with the `Lusbapi` library in the user program, you must make the following announcement (at least):

```
ILE2010 *pModule; // pointer to the interface of the E20-10  module
MODULE_DESCRIPTION_E2010 md; // structure of the service information about the module
```

Firstly, using the function GetDllVersion (), check the version of the `Lusbapi` library and the current software.

If the versions match, then in your application, you need to get a pointer to the module interface by calling the CreateLInstance () function. In the future, to access all interface functions of the library, it is necessary to apply this index (see the example below). After that, using the already received pointer to the module interface, you must initialize access to the corresponding virtual slot to which the *E20-10* module is connected. For this, the interface function OpenLDevice () is provided. If there is no error in running this function, you can be sure that a device of *E20-10* type is detected in the selected virtual slot.

Now, in principle, you can go to the stage of loading the detected module, but sometimes it is useful to determine the current speed of the used **USB** port. To this extend, the interface function GetUsbSpeed() is designed. To operate the module at effective data acquisition frequencies above 500 kHz, it is necessary that the module together with the **USB** port work in the so-called High-Speed Mode. This will correspond to the bandwidth of the **USB** bus to be ~ 60 MB/s. The maximum bandwidth of the module itself will be ~ 20 MB/s.

An important feature of the *E20-10* module is that it has a loadable FPGA on it. In order to *"heal"* the module and make it work according to the required algorithm, it is necessary to preload the firmware beforehand in the FPGA. You can use an interface function LOAD_MODULE(). In case of successful execution of this function, you need to check the operation of the loaded *LBIOS* using the interface function MODULE_TEST (). If this function is executed without error, it means that the *E20-10* module has been successfully loaded and is fully ready for further operation.

At the next stage, it is better to read the service information about the module. It is required while working with some interface functions of the regular DDL library `Lusbapi`. The interface function GET_MODULE_DESCRIPTION () is just intended for this purpose. If a function has not returned an error, it means that the information about the module is successfully received and you can continue the work.

In general, the preliminary stage of work with the *E20-10* module can be considered successfully completed. Now you can safely manage all available peripherals on the module with the appropriate interface functions of the `Lusbapi` library and organize various modes of the module. For example, such modes as:

- continuous *stream* collection with ADC with synchronization of data input;
- single, and therefore rather slow, data output to a dual-channel DAC;
- single, and therefore quite slow, work with input and output digital lines;
- work with user PROM module and many others.

As an example, we will give the source text, or rather say "*skeleton*," a small console program for working with the *E20-10* module, assuming the use of `Lusbapi` version no lower than 3.0:

```
#include <stdlib.h>
#include <stdio.h>
#include "Lusbapi.h"          // Lusbapi library header file
ILE2010 *pModule;            //a pointer to the module interface
MODULE_DESCRIPTION_E2010 md; // structure with the information about
the module
BYTE UsbSpeed;                // USB bus speed
char ModuleName[7];           // module name
int main(void)
{
 //verify the DLL library version
 if(GetDllVersion() != CURRENT_VERSION_LUSBAPI)
 {
     printf("Incorrect version Dll!");
     return 1;      //exit the program with an error
 }
 // we get a pointer to the module interface
 pModule = static_cast<ILE2010 *>(CreateLInstance("e2010"));
if(!pModule)
 {
     printf("It is impossible to get a pointer to the interface");
     return 1;      //exit the program with an error
 }
 // try to find some module
 // in a null virtual slot
if(!pModule->OpenLDevice(0))
 {
     printf("It is impossible to get the access to the module!");
     return 1;      //exit the program with an error
 }
 // try to get the speed of the USB bus
if (!pModule->GetUsbSpeed(&UsbSpeed))
 {
     printf("It is impossible to get the operational speed of USB!\n");
     return 1;      //exit the program with an error
 }
 // now display the received speed of the USB bus
printf ("USB is in% s \ n", UsbSpeed? "High-Speed Mode
                        (480 Mbit/s)" : "Full-Speed Mode (12 Mbit/s)");
 // read the module name in the null virtual slot
if (!pModule->GetModuleName (ModuleName))
 {
     printf("It is impossible to read the module name!\n");
```

```
    return 1;      //exit the program with an error
 }
 // just in case, check: this module is 'E20-10'?
if(strcmp(ModuleName, "E20-10"))
 {
    printf("In the null virtual slot of the module other than 'E20-
10'\n");
 return 1; //exit the program with an error }


 // Now you can try to download from the corresponding resource
 // libraries Lusbapi a firmware to the FPGA module
if (!pModule->LOAD_MODULE())
 {
    printf("Function LOAD_MODULE() is not executed!");
    return 1;      //exit the program with an error
 }

 // check the working capacity of the loaded module
if(!pModule->MODULE_TEST())
 {
    printf("Function MODULE_TEST() is not executed!");
    return 1;      //exit the program with an error
 }

 // try to read the information about the module
 if(!pModule->GET_MODULE_DESCRIPTION(&md))
 {
    printf("Function GET_MODULE_DESCRIPTION is not executed ()!");
    return 1;      //exit the program with an error
 }

 printf("Module E20-10 (serial number %s) is fully ready for\ work!",
                                 md.Module.SerialNumber);

 // further it is possible to have functions for direct //
management of the module, for example, on data collection
from ADC . . . . . .

 // complete the work with the module
 if(!pModule->ReleaseLInstance())
 {
    printf("Function ReleaseLInstance() failed!");
    return 1;      //exit the program with an error
 }
```

```
 // exit the program
 return 0;
}
```

4.2. Constants

The following basic constants are strongly recommended for use in the source code of the application when working with the *E20-10* module. This greatly improves the *readability* and *understandability* of source code, and also greatly facilitates the maintenance of programs. The constants in question are located in the file `\DLL\Include\Lusbapi.h`.

1. The module for starting the data acquisition process requires a hardware *start signal*. Constant data determines the source of this signal. The place to use these constants is usually the *StartSource* field of the ADC_PARS_E2010 structure.

Constant	Value	Intended purpose
INT_ADC_START_E2010	0	*Start signal* is internal and generated by the module itself. This impulse is not transmitted to the *DI16/START* line of the external controller ***DIGITAL I/O***.
INT_ADC_START_ WITH_TRANS_E2010	1	*Start signal* is internal and generated by the module itself. This signal is transmitted to the *DI16/START* line of the external ***DIGITAL I/O*** connector.
EXT_ADC_START_ON_ RISING_EDGE_E2010	2	It is expected to use an external *start signal*, which must be routed to the *DI16/START* line of the external ***DIGITAL I/O*** connector. In this case, data collection begins on the first incoming edge of this signal.
EXT_ADC_START_ON_ FALLING_EDGE_E2010	3	It is expected to use an external *start signal*, which must be routed to the *DI16/START* line of the external ***DIGITAL I/O*** connector. At the same time, data collection begins on the first descendant of this signal.

2. The module requires hardware clock pulses *for the operation of the ADC*. Constant data determines the source of this signal. The place to use these constants is usually the *StartSource* field of the ADC_PARS_E2010 structure.

Constant	Value	Intended purpose
INT_ADC_CLOCK_E2010	0	*The clock pulses* are internal and are generated by the module itself. The pulses are not transmitted to the *SYNC* line of the external ***DIGITAL I/O*** connector.

INT_ADC_CLOCK_ WITH_TRANS_E2010	1	*The clock pulses* are internal and are generated by the module itself. The pulses are transmitted to the *SYNC* line of the external ***DIGITAL I/O*** connector.
EXT_ADC_CLOCK_ON_ RISING_EDGE_E2010	2	It is expected to use external *clock pulses*, which must be connected to the *SYNC* line of the external ***DIGITAL I/O*** connector. In this case, the ADC operates along the edge of these pulses.
EXT_ADC_CLOCK_ON_ FALLING_EDGE_E2010	3	It is expected to use external *clock pulses*, which must be connected to the *SYNC* line of the external ***DIGITAL I/O*** connector. In this case, the ADC operates along the drop of these pulses.

3. Module *E20-10 (Rev.'B' and higher)* allows you to additionally use analog input data synchronization. The constants in the table below define the different modes of this synchronization. The place of use of these constants, as a rule, is the *SynchroAdMode* field of the SYNCHRO_PARS_E2010 structure, which is embedded with regard to the structure of ADC_PARS_E2010.

Constant	Value	Intended purpose
NO_ANALOG_SYNCHRO_E2010	0	Lack of analog synchronization.
ANALOG_SYNCHRO_ON_ RISING_CROSSING_E2010	1	Analog synchronization of the start of the data input upon the fact of the transition of the signal *' from below-upwards*' through the preset threshold on the selected channel.
ANALOG_SYNCHRO_ON_ FALLING_CROSSING_E2010	2	Analog synchronization of the start of the data input after the signal transition from '*top-down*' through the preset threshold on the selected channel.
ANALOG_SYNCHRO_ON_ HIGH_LEVEL_E2010	3	Analog data input synchronization only if the signal is located *above* the preset threshold on the selected channel.
ANALOG_SYNCHRO_ON_ LOW_LEVEL_E2010	4	Analog data input synchronization only if the signal is *below* the specified threshold on the selected channel.

4. The input channels of the *E20-10* module can be energized beyond the specified range. This leads to the congestion of the channels either to the *'plus'* or to the *'minus'*. The hardware of the *E20-10 module (Rev.'A ')* can differently fix the fact of the input channel congestion when data is collected from the ADC, which is determined by the following constants. Module *E20-10 (Rev.'B 'and above)* always works in overload limiting mode

(**CLIPPING_OVERLOAD_E2010**). The place to use these constants is usually the *OverloadMode* field of the ADC_PARS_E2010 structure.

Constant	Value	Intended purpose
CLIPPING_OVERLOAD_E2010	0	If there is an overload, the ADC code is limited to -8192 or 8191.
MARKER_OVERLOAD_E2010	1	If there is an overload, the ADC hardware generates ADC_MINUS_OVERLOAD_MARKER or ADC_PLUS_OVERLOAD_MARKER markers instead of the ADC code. Only for modules *Rev. A*.

5. The input channels of the *E20-10* module have three possible ranges of input voltages. Each of the ranges can be specified by the following constants. The place to use these constants is usually the *InputRange* field of the ADC_PARS_E2010 structure. The *InputRange* field is an array, each element of which specifies a specific input range for the corresponding physical channel of the ADC module.

Constant	Value	Intended purpose
ADC_INPUT_RANGE_3000mV_E2010	0	When used in the *InputRange* field, the input range is set to ± 3000 mV. You can also use it as an index to access the first element of the constant array ADC_INPUT_RANGES_E2010.
ADC_INPUT_RANGE_1000mV_E2010	1	When used in the *InputRange* field, the input range is set to ±1000 mV. You can also use it as an index to access the second element of the constant array ADC_INPUT_RANGES_E2010.
ADC_INPUT_RANGE_300mV_E2010	2	When used in the *InputRange* field, the input range is set to ±300 mV. You can also use it as an index to access the third element of the constant array ADC_INPUT_RANGES_E2010.

6. The module *E20-10* has two possible types of connection of the input channels. The required connection type is set by the following constants. The place to use these constants is usually the *InputSwitch* field of the ADC_PARS_E2010 structure. The *InputSwitch* field is an array, each element of which specifies a specific type of connection for the corresponding physical channel of the ADC module.

Constant	Value	Intended purpose
ADC_INPUT_ZERO_E2010	0	This constant corresponds to the grounded channel of the ADC module.
ADC_INPUT_SIGNAL_E2010	1	This constant sets the input signal to the input of the ADC module.

7. On the *E20-10* module, a dual-channel 12-bit DAC chip can be installed at the user's request. The status of the *Dac.Active* field of the service information structure MODULE_DESCRIPTION_E2010 reflects the presence of the DAC on board the module.

Constant	Value	Intended purpose
DAC_INACCESSIBLED_E2010	0	The module completely lacks the DAC chip.
DAC_ACCESSIBLED_E2010	1	The module contains a DAC chip.

8. The revision of the *E20-10* module reflects certain design features of the module. It is specified by one uppercase letter and placed in the Revision field of the embedded structure MODULE_INFO_LUSBAPI of the service structure MODULE_DESCRIPTION_E2010. For example, the *first* revision of the module is designated as the letter *'**A**'*.

Constant	Value	Intended purpose
REVISION_A_E2010	0	This constant can be used as an index to access the first element of the constant array REVISIONS_E2010.
REVISION_B_E2010	1	This constant can be used as an index to access the second element of the constant array REVISIONS_E2010.

9. Module *E20-10* allows you to monitor the internal buffer overflow of the module, which leads to a violation of the integrity of the data collected from the ADC. This information is reflected in the bit with the number 0 or **BUFFER_OVERRUN_E2010** in the `BufferOverrun` field of the structure DATA_STATE_E2010. The appearance of the logical state '1' in this bit indicates that during the data acquisition time the internal buffer of the module has overflowed.

10. Module *E20-10 (Rev. 'B' and above)* allows you to monitor the global (for all time collection) and local (during the time of one query) bit attributes of the overflow of the bitmap. The global bit flag is activated (goes into the "1" state) when the bit grid overflows at any of the 4 physical ADC channels for the entire time interval from START_ADC () and up to STOP_ADC (). Each of their local bit attributes is activated (goes into the state of the log "1") when the bitmap overflow occurs at the corresponding physical ADC channel during the time of one ReadData() request. Each of these features occupies the corresponding bit in the field `ChannelsOverFlow` of the DATA_STATE_E2010 structure. All numbers of available bits are listed in the table below:

Bit number	Constant name	Intended purpose
0	**OVERFLOW_OF_CHANNEL_1_E2010**	Local sign of the word size overflow of the 1st physical ADC channel.
1	**OVERFLOW_OF_CHANNEL_2_E2010**	Local sign of the word size overflow of the 2nd physical ADC channel.
2	**OVERFLOW_OF_CHANNEL_3_E2010**	Local sign of the word size overflow of the 3rd physical ADC channel.
3	**OVERFLOW_OF_CHANNEL_4_E2010**	Local sign of the word size overflow of the 4th physical ADC channel.
<4..6>	———————	Reserved
7	**OVERFLOW_E2010**	Global flag for word size overflow.

11. Various constants for working with the *E20-10* module.

Constant	Value	Intended purpose
CURRENT_VERSION_LUSBAPI	——	The version of the `Lusbapi` library used. Typically, it is used together with the GetDllVersion() function.
MAX_CONTROL_TABLE_LENGTH_E2010	256	The maximum possible number of logical channels in the **ControlTable** control table.
ADC_CHANNELS_QUANTITY_E2010	4	Number of physical channels of the ADC on the module.
ADC_CALIBR_COEFS_QUANTITY_E2010	12	The number of correction factors for these ADC data. One for each channel and for each input range. The reset and the scale of the ADC data are subject to adjustment.
DAC_CHANNELS_QUANTITY_E2010	2	Number of physical channels DAC on the module (subject to the presence of a DAC chip on the module).

DAC_CALIBR_COEFS_ QUANTITY_E2010	2	The number of correction coefficients for the ADC data. One for each channel. The reset and the scale of the ADC data are subject to adjustment.
ADC_INPUT_RANGES_ QUANTITY_E2010	3	The number of input range.
ADC_INPUT_TYPES_ QUANTITY_E2010	2	The number of types of input channel connections.
TTL_LINES_QUANTITY_ E2010	16	The number of input and output digital lines.
USER_FLASH_SIZE_E2010	512	The size of the user PROM area in bytes
REVISIONS_QUANTITY_ E2010	2	The number of revisions (modifications) of the module.
ADC_PLUS_OVERLOAD_ MARKER	0x5FFF	Marker '*plus*' for the ADC channel overload. The marker mode of channel overloading fix is implied. Only for the module *Rev.'A'*.
ADC_MINUS_OVERLOAD_ MARKER	0xA000	Marker '*minus*' for the ADC channel overload. The marker mode of channel overloading fix is implied. Only for the module *Rev.'A'*.

12. Different *constant arrays* for working with the *E20-10* module.
    12.1. The array of available ADC input voltage ranges in Volts:

```
const double
        ADC_INPUT_RANGES_E2010[ADC_INPUT_RANGES_QUANTITY_E2010] =
{
 3.0, 1.0, 0.3
};
```

    12.2. The output voltage range of the DAC in Volts:

```
const double DAC_OUTPUT_RANGE_E2010 = 5.0;
```

    12.3. The module audit reflects certain design features of the module. It is given by one uppercase letter. For example, the *first* revision of the module is denoted by the letter ***'A'***. The current revision of the module is contained in the *Module.Revision* field of the service information structure MODULE_DESCRIPTION_E2010. An array of available module revisions is specified as follows:

```
const BYTE REVISIONS_E2010[REVISIONS_QUANTITY_E2010] =
{
'A', 'B'
};
```

4.3. Structures

This section shows the main types of structures that are used in the `Lusbapi` library when working with the *E20-10* module.

4.3.1. Structure of MODULE_DESCRIPTION_E2010

Structure of *MODULE_DESCRIPTION_E2010* is described in the file `\DLL\Include\Lusbapi.h` and presented as follows:

```
struct MODULE_DESCRIPTION_E2010
{
 MODULE_INFO_LUSBAPI Module;   // general information about the module
 INTERFACE_INFO_LUSBAPI Interface;  // interface information
 MCU_INFO_LUSBAPI<MCU_VERSION_INFO_LUSBAPI> Mcu;  // information about MCU
 PLD_INFO_LUSBAPI Pld;     // information about FPGA
 ADC_INFO_LUSBAPI Adc;     // information about ADC
 DAC_INFO_LUSBAPI Dac;     // information about DAC
 DIGITAL_IO_INFO_LUSBAPI DigitalIo;  // information about digital I/O  };
```

This structure provides the most common service information about the instance of the *E20-10* module used. This structure is used when working with the interface functions SAVE_MODULE_DESCRIPTION () and GET_MODULE_DESCRIPTION (). The definition of this structure uses the auxiliary constants and data types described in Appendix A.

4.3.2. Structure of the ADC_PARS_E2010

Structure of the *ADC_PARS_E2010* is a group of parameters that specify the parameters for data collection from the ADC. This structure is described in the file `\DLL\Include\Lusbapi.h` and is presented below:

```
struct ADC_PARS_E2010
{
 BOOL IsAdcCorrectionEnabled;  // automatic adjustment control
                               // at the FPGA level of the module received from the ADC
                               // data (for the Rev. 'B' module and above)
 WORD OverloadMode;  //fixing the overload of the input channels (for the module Rev.'A')
 WORD InputCurrentControl;        // input offset current control
                                  // (for module Rev.'B' and above)
 SYNCHRO_PARS_E2010 SynchroPars;  // input synchronization options
                                  // data from the ADC
 WORD ChannelsQuantity;           // number of active channels (frame size)
 WORD ControlTable[256];          // control table with logical channels
 WORD InputRange[ADC_CHANNELS_QUANTITY_E2010]; //input voltage range
 WORD InputSwitch[ADC_CHANNELS_QUANTITY_E2010]; // channel connection type
 double AdcRate;                  // operation frequency of the ADC, kHz
 double InterKadrDelay;           // interframe delay, ms
```

```
 double KadrRate;                        // frame frequency, kHz
 double AdcOffsetCoefs[ADC_INPUT_RANGES_QUANTITY_E2010]
                      [ADC_CHANNELS_QUANTITY_E2010];
                       // array of coefficients to correct the ADC offset:
                       // (3 ranges)* (4 channels) (for module Rev.'B' and above)
 double AdcScaleCoefs[ADC_INPUT_RANGES_QUANTITY_E2010]
                     [ADC_CHANNELS_QUANTITY_E2010];
                      // an array of coefficients for adjusting the ADC scaling:
                      // (3 ranges)* (4 channels) (for module Rev.'B' and above)
};
```

Before working with the ADC, you must fill in the fields of this structure and transfer it to the module using the interface function SET_ADC_PARS (). In the description of this function, the meaning and purpose of all fields of the given structure are explained in detail. Also, if necessary, you can read the current parameters of the ADC from the module using the interface function GET_ADC_PARS ().

4.3.3. Structure of the SYNCHRO_PARS_E2010

Structure of the `SYNCHRO_PARS_E2010` is a set of parameters used to specify a variety of synchronization modes for data input from the ADC. This structure is described in the file `\DLL\Include\Lusbapi.h` and is presented below:

```
struct SYNCHRO_PARS_E2010
{
 WORD StartSource;         // source of the impulse to start data collection from the ADC
DWORD StartDelay;          // delay start of data collection in frame count c
                           // the ADC (for module Rev.'B' and above)
 WORD SynhroSource;        // source of ADC startup clock
 DWORD StopAfterNKadrs;    // stop collecting data after the one specified here
                           // count of the collected frames of ADC samples (for
                           // module Rev.'B' and above)
 WORD SynchroAdMode;       // analogue synchronization mode: by a transition
                           // or by a level (for module Rev.'B' and above)
 WORD SynchroAdChannel;    // physical channel ADC for an analogue
                           // synchronization (for module Rev.'B' and above)
 SHORT SynchroAdPorog;     // threshold for analog
                           // synchronization (for module Rev.'B' and above)
 BYTE IsBlockDataMarkerEnabled; // marking the beginning of a data block, that
                                // is quite convenient, for example, for an analogue
                                // synchronization of data input by level
                                // (for module Rev.'B' or above)
};
```

Structure of the `SYNCHRO_PARS_E2010` is a part of the ADC_PARS_E2010 structure.

4.3.4. Structure of the IO_REQUEST_LUSBAPI

Structure of the *IO_REQUEST_LUSBAPI* is described in the file \DLL\Include\LusbapiTypes.h and presented as follows:

```
struct IO_REQUEST_LUSBAPI
{
 SHORT * Buffer;                // buffer for transmitted data
 DWORD NumberOfWordsToPass;     // number of counts to be transferred
 DWORD NumberOfWordsPassed;     // number of really transmitted counts
     OVERLAPPED * Overlapped;
                                // for a synchronous request – NULL, and for an asynchronous
                                // request – a pointer to the structure OVERLAPPED
     DWORD TimeOut;             // for synchronous request, timeout in ms, and for
                                // asynchronous request it is not used
};
```

This structure is used by the function ReadData() while transmitting the data received from the ADC from the module to a computer. In the description of this function, the meaning and purpose of the fields of this structure are explained in detail.

4.3.5. Structure of the USER_FLASH_E2010

Structure of the *USER_FLASH_E2010* is described in the file \DLL\Include\Lusbapi.h and presented as follows:

```
struct USER_FLASH_E2010
{
 BYTE Buffer[USER_FLASH_SIZE_E2010]; // size of the function in bytes };
```

This structure is designed to store or read the user information. A region of USER_FLASH_SIZE_E2010 bytes in the PROM of the microcontroller is allocated to operate with it. It is used in functions READ_FLASH_ARRAY() and WRITE_FLASH_ARRAY().

4.3.6. Structure of the LAST_ERROR_INFO_LUSBAPI

Structure of the *LAST_ERROR_INFO_LUSBAPI*is described in the file \DLL\Include\LusbapiTypes.h and presented below:

```
struct LAST_ERROR_INFO_LUSBAPI
{
 BYTE ErrorString[256];   // a line with a brief description of the last error
 DWORD ErrorNumber;       // a number of the last error of the library Lusbapi
};
```

This structure is used by the GetLastErrorInfox() function to detect errors in the interface functions of the Lusbapi library.

4.3.7. Structure of the DATA_STATE_E2010

Structure of the *DATA_STATE_E2010* is described in the file \DLL\Include\Lusbapi.h and presented below:

```
struct DATA_STATE_E2010
```

```
{
 BYTE ChannelsOverFlow;  // bit signs of input channel overload
                         // for the module Rev.'B' and above
 BYTE BufferOverrun;     // bit signs of internal overflow
                         // module's hardware buffer
 DWORD CurBufferFilling; // current internal buffer occupacy
                         // for the module Rev.'B' and above, in counts
 DWORD MaxOfBufferFilling; // for the duration of the collection, the maximum occupancy
                         // of the module's internal buffer Rev.'B' and above, in counts
 DWORD BufferSize;       // the buffer capacity of the module Rev.'B' and above, in counts
 double CurBufferFillingPercent;   // the current level of occupancy of the internal
                         // buffer of the module Rev.'B' and above, %
 double MaxOfBufferFillingPercent; // and during the collection the maximum degree
                         // of occupancy of the internal buffer of the module Rev.'B' and
above, % };
```

This structure is used by the GET_DATA_STATE() function while polling the current state of the data collection process. In the description of this function, the meaning and purpose of the fields of this structure are explained in detail.

4.4. General functions

4.4.1. Getting the library version

Format:	**DWORD**	***GetDllVersion***(*void*)

Assignment:

This function is one of *two* exported functions from the regular library `Lusbapi` by the function. It returns the current version of the used library. The format of the version number is:

Bit field	Intended purpose
<31..16>	The high word of the library version
<15..0>	The low word of the library version

For the recommended sequence of calls for interface functions, see § 4.1. "General principles of working with module".

Transmitted parameters: no.

Returned value: version number of the `Lusbapi` library.

4.4.2. Getting the pointer to the module's interface

Format:	**LPVOID**	***CreateLInstance***(*PCHAR const DeviceName*)

Assignment:

This function should always be called at the beginning of each user program that works with the *E20-10* modules. It is one of *two* functions exported from the regular `Lusbapi` library and returns a pointer to the interface for a device called *DeviceName*. All subsequent interface functions of the standard library are called precisely through this returned pointer.

For the recommended sequence of calls for interface functions, see § 4.1. "General principles of working with the module"

Transmitted parameters:

DeviceName is a string with the device name (for this module it is `"E2010"`).

Returned value: If successful, the pointer to the interface, otherwise — **NULL**.

4.4.3. Shutting down with the module interface

Format:	**BOOL**	***ReleaseLInstance***(*void*)

Assignment:

This interface function implements the correct release of the interface of the *E20-10* module, initialized with the CreateLInstance() function. It is used to close the session with the module neatly (if the CreateLInstance() function was successfully executed beforehand). **!!!Attention!!!** This function must be called in the application before it is terminated in order to avoid leakage of *Windows* resources.

For the recommended sequence of calls for interface functions, see § 4.1. "General principles of working with module".

Transmitted parameters: no.

Returned value: *TRUE* – function was successfully executed;
FALSE – function was executed with an error.

4.4.4. Initialization of access to the module.

Format: BOOL	***OpenLDevice***(*WORD VirtualSlot*)

Assignment:
From a programmatic point of view, without going into too much detail, the *E20-10* module connected to the computer can be considered as a device connected to a virtual slot with a strictly individual number. The main purpose of this interface function is just to determine that it is the *E20-10* module that is in the specified virtual slot. If ***OpenLDevice()*** function was successfully executed for a given virtual slot, you can go directly to the module load and its subsequent management with the appropriate interface functions of the `Lusbapi` library.

For the recommended sequence of calls for interface functions, see § 4.1. "General principles of working with module".

Transmitted parameters:
VirtualSlot is the virtual slot number to which the *E20-10* module is supposed to be connected.

Returned value: *TRUE* – the module *E20-10* is in the selected virtual slot and you can start module loading;
FALSE – there is no device of the *E20-10* module type in the selected virtual slot. You should try another virtual slot number.

4.4.5. Shutting down the access to the module

Format: BOOL	***CloseLDevice***(*void*)

Assignment:
This interface function interrupts any interaction with the *current* virtual slot to which the module is connected. This virtual slot is neatly closed and the *Windows* resources associated with it are released. After successful execution of this function, any access to the *E20-10* module becomes impossible. To resume normal access to the device, use the OpenLDevice() interface function again. Thus, this function is opposite to the OpenLDevice() function. In fact, this function is used in such interface functions as OpenLDevice() and ReleaseLInstance().

Transmitted parameters: no.

Returned value: *TRUE* – function was successfully executed;
FALSE – function was executed with an error.

4.4.6. Module loading

Format: BOOL	***LOAD_MODULE***(*PCHAR const FileName = NULL*)

Assignment:
This interface function performs the operation of downloading the firmware (standard or user) in the FPGA module. The binary file *FileName* with the firmware code must be in the current application directory. It is possible to load FPGA firmware, stored in the body of the library in the form of a corresponding resource, in the standard library. To do this, simply set the *FileName* parameter as **NULL**. **NULL** is also the default value for the *FileName* parameter.

For the recommended sequence of calls for interface functions, see § 4.1. "General principles of working with module".

Transmitted parameters:
FileName – a string with the name of the binary firmware FPGA module. For example, for firmware files, this is the string `"E2010.pld"` (for the *Rev.'A'* module) or `"E2010m.pld"` (for the *Rev.'B'* module or above). If this parameter is set to **NULL** or absent at all, then the module will be

loaded by the *firmware* that is in the form of a resource in the body of the standard library.	
Returned value:	*TRUE* – function was successfully executed; *FALSE* – function was executed with an error.

4.4.7. Module loading check

Format: BOOL *TEST_MODULE*(*void*) (version 3.1 and below) BOOL *TEST_MODULE*(*WORD TestModeMask = 0x0*) (version 3.2 and higher)	
Assignment: For the *E20-10 module (Rev.'A')*, this interface function is only a stub and does not carry any functional load. For the *E20-10 module (Rev.'B' and above)*, this interface function checks the functional state of the FPGA module. In addition, this function allows you to transfer the module to the test mode of operation, which is used exclusively for commissioning purposes.	
Transmitted parameters: *TestModeMask* – the required test mode of the module. If the *TestModeMask* parameter is zero, the module goes into normal operation mode.	
Returned value: *TRUE* – FPGA is loaded and functions properly; *FALSE* – there was an error loading or functioning of the FPGA module.	

4.4.8. Getting the module name

Format: BOOL ***GetModuleName***(*PCHAR const ModuleName*)	
Assignment: This auxiliary interface function allows you to get the name of the module connected to the slot. An array named module *ModuleName* (at least 6 characters plus the end-of-line character of _\0', that is, zero byte) must be predefined. For the recommended sequence of calls for interface functions, see § 4.1. "General principles of working with module".	
Transmitted parameters: *ModuleName* – returns a string, at least 6 characters, with the name of the module (in our case this should be the string `"E20-10"`).	
Returned value:	*TRUE* – function was successfully executed; *FALSE* – function was executed with an error.

4.4.9. Getting the speed of the module

Format: BOOL ***GetUsbSpeed***(*BYTE * const UsbSpeed*)	
Assignment: This function allows you to determine at what speed the **USB** bus works with the module.	
Transmitted parameters: *UsbSpeed* – the return value of this variable can take the following: ✓ 0 – the module functions with the **USB** bus in the *Full-Speed Mode* mode (12 Mb/s) ✓ 1 – the module functions with the **USB** bus in the *High-Speed Mode* mode (480 Mb/s).	
Returned value:	*TRUE* – function was successfully executed; *FALSE* – function was executed with an error.

4.4.10. Getting the module descriptor

Format: **HANDLE** ***GetModuleHandle***(*void*)	
Assignment: This function allows you to get the descriptor (handle) of the used module *E20-10*.	
Transmitted parameters: no	
Returned value:	In case of success, the descriptor of the *E20-10* module; Otherwise, **INVALID_HANDLE_VALUE**.

4.4.11. Obtaining a description of the functions execution errors

Format: **BOOL** ***GetLastErrorInfo***(*LAST_ERROR_INFO_LUSBAPI * const LastErrorInfo*)	
Assignment: If during the work with the Lusbapi library some interface function of the standard library returned an error, then only after this, by calling this interface function, you can get a brief interpretation of the failure. For some functions, such as ReadData(), you may need to call the standard *Windows API* ***GetLastError()*** function to identify the errors of the *Windows* itself, if the cause of the error is identified.	
Transmitted parameters: *LastErrorInfo* – a pointer to the type structure LAST_ERROR_INFO_LUSBAPI, in which a short description and the number of the last error is returned.	
Returned value:	*TRUE* – function was successfully executed; *FALSE* – function was executed with an error.

4.5. *Functions for working with ADC*

The hardware of the *E20-10* module and, accordingly, the Lusbapi library are designed to organize continuous *streaming* of data collection from the ADC at frequencies up to 10 MHz. At the same time, even a multi-module (by the principle *"master-slave"*) mode of data acquisition is quite feasible under different conditions of input synchronization.

Before starting the data collection from the ADC, you must send the required parameters to the module: type of synchronization, frequency of operation of ADC, control table, etc. This operation can be performed using the interface function SET_ADC_PARS(). After that, in principle, you can run the module to collect data by executing the function START_ADC(). The data received from the ADC module will be transmitted via the **USB** bus to the computer as necessary. To implement the ADC data transfer from the module to the PC, you should use the regular ReadData() function. The ReadData() function can be executed in both *synchronous* and *asynchronous* modes. After completing the last portion of the collected data, but no later than 400 ms, it is strongly recommended to perform the function of completing the collection function STOP_ADC(). After that, using the function GET_DATA_STATE(), you can check the status of the completed data collection process for failures or collection errors.

Before operating the module at an effective collection frequency of more than 500 kHz, you need to make sure that the *E20-10* module communicates with the **USB** bus in the so-called High-Speed Mode. To this end it is recommended to use the GetUsbSpeed() function.

A whole set of examples on how to organize continuous data collection with the *E20-10* module for various development environments can be found on our CD-ROM in the *E20-10*\Examples\ directory.

4.5.1. ADC data correction

Circuitry and used electronic components provide the linearity of the transmission characteristic of the ADC path of the *E20-10* module. However, the module does not have any trimmer resistors. And although this allows improving the noise characteristics of the module and increasing the reliability of the module, it inevitably leads to the fact that the ADC input may have some mixing of zero and inaccuracy in the transmission scale. Therefore, either at the FPGA level of the module or at the application level, it is necessary to organize the correction of the data received from the ADC.

For the module *E20-10 (Rev.'A')*, the correction of data collection at the application level was provided. But in the module *E20-10 (Rev.'B' and above)* there is an additional possibility of automatic correction at the FPGA level. That is, the module itself corrects the received ADC data. At the same time, it is perfectly possible to use both *standard (factory)* and own ones, *user*, adjustment factors. *User* adjustment factors can be used, for example, to compensate for errors in the whole measuring path of a stand, of which the *E14-140* module can be a component. At the same time, the entire responsibility for the formation and correct application of the *user* adjustment factors lies entirely on the shoulders of the end user.

Standard (factory) adjustment factors are located in the fields *Adc.OffsetCalibration[]* and *Adc.ScaleCalibration[]* of the service information structure MODULE_DESCRIPTION_E2010. All service information together with adjustment factors is recorded in the module at the stage when it is set up at LLC "L-Card". The coefficient fields are arrays of type *double*. For the *E20-10* module, only the first ADC_CALIBR_COEFS_QUANTITY_E2010 elements are used in each of these arrays. An array *Adc.OffsetCalibration* contains coefficients to adjust the zero offset, and an array *Adc.ScaleCalibration* to adjust the scale. If we denote by *i* the physical channel number of the ADC module, and via *j* – the index of the input range of this channel, then the channel correction factors can be obtained as follows:

- offset: *Adc.OffsetCalibration[i + j**ADC_CHANNELS_QUANTITY_E2010*]*;
- scale: *Adc.ScaleCalibration[i + j**ADC_CHANNELS_QUANTITY_E2010*]*.

In general, the procedure for correcting the ADC samples is performed using the following formula:

Y = (X+A)*B,

where: X – uncorrected ADC data [in ADC counting],

Y – corrected ADC data [in ADC counting],

A – a zero offset coefficient [in ADC counting],

B – a scale coefficient [unsized].

For example, from the second ADC channel tuned to the input range of ±1.0 V (index range is 1 or ADC_INPUT_RANGE_1000mV_E2010), the following data is received: X1 = 1000, X2 = –1000 and X3 = 0. Then the coefficients and the corrected data can be represented as follows:

A = Adc.OffsetCalibration[1 + 1*ADC_CHANNELS_QUANTITY_E2010];
B = Adc.ScaleCalibration[1 + 1*ADC_CHANNELS_QUANTITY_E2010];
Y1 = (A+1000)*B, Y2 = (A-1000)*B, Y3 = A*B.

4.5.2. Running ADC data collection

Format:	**BOOL**	***START_ADC***(*void*)
Assignment: This function starts the *E20-10* module for continuous *streaming* of data from the ADC. Before any start of data collection, it is highly recommended to perform STOP_ADC() function. Before the start of the collection, you can set the required parameters of the ADC's operation, which are transferred to the module using the interface function SET_ADC_PARS(). Also, the ***START_ADC()*** function resets the integrity of the module data to its original zero state. The extraction of data coming from the module can be performed using the interface function ReadData().		
Transmitted parameters: no		
Returned value:	*TRUE* – function was successfully executed; *FALSE* – function was executed with an error.	

4.5.3. Stopping ADC data collection

Format:	**BOOL**	***STOP_ADC***(*void*)
Assignment: This function stops the data acquisition mechanism from the ADC in the *E20-10* module. Along the way, this function *"brings to mind"* the main program (*Firmware*) of the microcontroller module, and also resets the data channel used via the **USB** bus. Therefore, it is strongly recommended to use this function before every data collection run by the START_ADC() function. It is also highly recommended that ***STOP_ADC()*** be used after the last portion of the collected data is entered, but no later than a certain period of time. For example, for a collection frequency of 10 MHz, this period should not be more than 400 ms. This makes it possible to use GET_DATA_STATE() function to get the integrity indicator of *all* the latest data collected from ADC.		
Transmitted parameters: no		
Returned value:	*TRUE* – function was successfully executed; *FALSE* – function was executed with an error.	

4.5.4. Module ADC operation parameter setting

Format:	BOOL	***SET_ADC_PARS***(*ADC_PARS_E2010 * const AdcPars*)

Assignment:

This function sends to the *E20-10* all the necessary information that is used by the module to organize the specified data collection mode from the ADC. The described interface function retrieves all the necessary information from the fields of the transmitted structure of type ADC_PARS_E2010. Actually, the use of this particular transmitted information by the module starts only after the interface function START_ADC() is executed. It is also highly discouraged to call this function in the actual data collection process. It should be used only after performance of the STOP_ADC() function.

The format of the structure of ADC_PARS_E2010 is given earlier in § 4.3.2. "Structure ADC_PARS_E2010", and the meaning and purpose of its individual fields is described in sufficient detail below.

- Field AdcPars->IsAdcCorrectionEnabled. Entry. For the *E20-10 (Rev.'A')* module, this field does not carry any functional load. For the *E20-10 module (Rev.'B' and above)* with this field, you can set the automatic correction (at the FPGA module level) of the data received from the ADC. When using automatic correction, it is necessary to correctly fill `AdcOffsetCoefs[]` and `AdcOffsetCoefs[]` arrays with the corresponding coefficients. For most cases, you can use the *factory* adjustment coefficients that are located in the *Adc.OffsetCalibration[]* and *Adc.ScaleCalibration[]* fields of the service information structure MODULE_DESCRIPTION_E2010.
- Field AdcPars->OverloadMode. Entry. The input channels of the *E20-10* module can be energized beyond the specified range. This leads to the congestion of the channels either to the *'plus'* or to the *'minus'*. The hardware of the *E20-10 module (Rev. 'A')* can differently record the fact of the input channel overload when data is collected from the ADC, which is set by the following overload constants. And the module *E20-10 (Rev.'B' and above)* always functions in the mode of overflow limiting (**CLIPPING_OVERLOAD_E2010)**. The meaningful loading of the values of this field is presented in the table below:

Value	Constant	Description
0	**CLIPPING_ OVERLOAD_E2010**	***Limiting.*** If there is an overload, the reference code from the ADC is limited to -8192 or 8191.
1	**MARKER_ OVERLOAD_E2010**	***Markers.*** If there is an overload, the hardware of the module mixes the overload *indicator* in the ADC reference code. In this case, markers ADC_MINUS_OVER-LOAD_MARKER (with *_minus'* overload) or ADC_PLUS_OVERLOAD_MARKER (with *_plus'* overload) are formed. Only for the module *Rev. A*.

- Field AdcPars->InputCurrentControl. Entry. For the *E20-10 (Rev.'A')* module, this field does not carry any functional load. For the module *E20-10 (Rev.'B' and above)* with this field it is possible to control the input current of the analog module's offset. This field can be either 0 or 1. For more information about the input offset current, see "E20-10. User Manual. § 6.5.4. ADC entry point connection.".

- Field AdcPars->SynchroPars. Write-Read. This field is a nested structure type SYNCHRO_PARS_E2010 in which all the parameters are grouped related to synchronization of data input:
  - Field AdcPars.StartSource Entry. To start the process of collecting the *start signal*. This field defines the source of the signal. This field can take one of four, and you can also use the start signal constants. The meaningful loading of the values of this field is presented in the table below:

Value	Description
0	***Internal source of the start signal without its translation to the output*** In this mode, when the method START_ADC() is called, the data collection from the ADC is automatically started. In this case, the digital *DI16/START* of the external ***DIGITAL I/O*** connector is configurable *with* respect to the module and there is no *start signal* on it.
1	***Internal source of the start signal with its translation to the output*** In this mode, when the method START_ADC() is called, the data collection from the ADC is automatically started. In this case, the digital *START* of the external ***DIGITAL I/O*** connector is configured both in relation to the module and when the ADC starts, an event will occur on it in the form of a *DI16/START* transition from the state 'to _1' (*rising* edge). The *DI16/START* line will be logged as _1' as long as the data is being collected from the ADC and the initial state is log. _0', when data collection is programmed to stop using the STOP_ADC() function.
2	***External source of the start signal with activity on the leading edge.*** In this mode, when the function START_ADC() is called, the module goes into the standby mode of the event on the digital line *DI16/START* of the ***DIGITAL I/O*** connector. In this case, the digital line is configured as an *input* in relation to the module and events on it in the form of a transition of the line *DI16/START* 0' to _1' (*rising* edge) the ADC collection starts.
3	***External source of the start signal with the activity on the back edge.*** In this mode, when the function START_ADC() ***is called*** the module enters the event waiting mode on the *DI16/START* digital line of the ***DIGITAL I/O*** terminal. In this case, the digital line is configured as an *input* with respect to the module and when an event is detected in the form of a transition of the *DI16/START* line from 1' to _0' (*trailing* edge), the ADC collection starts.

  - Field AdcPars.StartDelay Write-Read. For the module *E20-10* this field does not have any functional load. For the module *E20-10* with this field, it is possible to set the delay of the start moment in *ADC count frame*. Range of admissible values. Thus, after the arrival of the hardware *start signal* itself only after the specified number of count frames has been skipped.
  - Field AdcPars->SynchroPars.SynhroSource. Entry. The operation of the ADC requires the presence of hardware *clock pulses*. This field determines the source of the formation of these pulses. This field can take one of the four values from 0 to 3, and you can also use the clock impulse constants. It should be remembered that when selecting an external *clock impulse* source, their frequency should be strictly in the

range from 1 MHz to 10 MHz. The meaningful loading of the values of this field is presented in the table below:

Value	Description
0	***Internal clock source without their translation to the module output.*** In this mode, when the method *START_ADC()* is called, internal *clock impulses* generation is performed for the ADC module. In this case, the *SYNC* digital line of the external ***DIGITAL I/O*** connector is configured as an input *to the* module and *the clock pulses are* not transmitted to it.
1	***Internal source of clock pulses with their translation to the module output.*** In this mode, when the method *START_ADC()* is called, internal *clock impulses* generation is performed for the ADC module. In this case, the *SYNC* digital line of the external ***DIGITAL I/O*** connector is configured as *output* with regard to the module and the generated *clock pulses* are transmitted to it.
2	***External clock source with activity on the leading edge.*** In this mode, when the *START_ADC()* function is called, an *external clock* source is used, that is connected to the digital line *SYNC* of the external ***DIGITAL I/O connector***. In this case, the *SYNC* line is configured as an *input* with regard to the module and an event in the form of a signal transition on this line from the log state _0' to _1' (*rising* edge) is interpreted as an external sync signal for ADC operation.
3	***External clock source with activity on the falling edge.*** In this mode, when the *START_ADC()* function is called, an *external clock* source is used, that is connected to the digital line *SYNC* of the external ***DIGITAL I/O connector***. In this case, the *SYNC* line is configured as an *input* with regard to the module and an event in the form of a signal transition on this line from the log state _1' to _0' (*trailing* edge) is interpreted as an external sync signal for ADC operation.

- Field AdcPars->SynchroPars.StopAfterNKadrs. Write-Read. For the *E20-10 (Rev.'A')* module, this field does not carry any functional load. For the module *E20-10 (Rev.'B' and higher)* with this field, you can organize stopping the data collection after the number of *collected frame counts* specified here. Range of values from 0 to 16 777 215. If the StopAfterNKadrs field is set to 0, this parameter will be completely ignored by the module when data is collected. Particular mention should be made of the advanced features of the module with StopAfterNKadrs 0 and certain synchronization conditions: by an external *start* signal (the StartSource field is 2 or 3) and/ or with analog synchronization on the *transition* (the SynchroAdMode field is 1 or 2). In this case, the module will collect data by the block after StopAfterNKadrs for each execution of synchronization conditions.

  For example, if StopAfterNKadrs = 1024 and StartSource = 0x2 are specified, then after executing START_ADC(), the module will go into the standby mode of the active drop to the external clock line. If this is detected, the module will collect a 1024-frame data block, and then *automatically* return to waiting for the next active sync. impulse. This process will continue cyclically until the STOP_ADC() function is executed. In principle, the module can label each received block of data in a special way, if the IsBlockDataMarkerEnabled field is set to 1 in addition.

- Field AdcPars->SynchroPars.SynchroAdMode. Entry. For the *E20-10 (Rev.'A')* module, this field does not carry any functional load. For the *E20-10 module (Rev.'B' and higher)*, this field allows you to specify different modes of analog input data synchronization. This field can take one of five values from 0 to 4, and you can also use the constants of analog synchronization modes. The meaningful loading of the values of this field is presented in the table below:

Constant	Value	Intended purpose
NO_ANALOG_SYNCHRO_E2010	0	Lack of analog synchronization.
ANALOG_SYNCHRO_ON_ RISING_CROSSING_E2010	1	Analog synchronization of the start of the data input upon the fact of the transition of the signal *' from below-upwards*' through the preset threshold on the selected channel.
ANALOG_SYNCHRO_ON_ FALLING_CROSSING_E2010	2	Analog synchronization of the start of the data input after the signal transition from '*top-down*' through the preset threshold on the selected channel.
ANALOG_SYNCHRO_ON_ HIGH_LEVEL_E2010	3	Analog data input synchronization only if the signal is located *above* the preset threshold on the selected channel.
ANALOG_SYNCHRO_ON_ LOW_LEVEL_E2010	4	Analog data input synchronization only if the signal is *below* the specified threshold on the selected channel.

- Field AdcPars->SynchroPars.SynchroAdChannel. Entry. For the *E20-10 (Rev.'A')* module, this field does not carry any functional load. For the *E20-10 module (Rev.'B' and above)*, using this field, you can set the physical ADC channel for the selected *analogous synchronization*. This field can take one of four values from 0 to 3.
- Field AdcPars->SynchroPars.SynchroAdPorog. Entry. For the *E20-10 (Rev.'A')* module, this field does not carry any functional load. For the *E20-10 module (Rev.'B' and above)*, the *analogous synchronization* threshold can be set using this field. The threshold is set in the ADC codes in the range from -8192 to 8191.
- Field AdcPars->SynchroPars.IsBlockDataMarkerEnabled. Entry. For the *E20-10 (Rev.'A')* module, this field does not carry any functional load. When the field IsBlockDataMarkerEnabled is set to 1, the hardware module *E20-10 (Rev.'B' and above)* inserts an artificial logical marker in the first sample of each *continuous* (by

time) data block, which is encoded by setting the value "01" in the fields <15..14> countdown of the data. Such a marker allows you to distinguish the beginning of one continuous piece of data from the other at the upper software level. The presence of such a marker can be particularly useful for, for example, data entry using analog level synchronization.

- Field AdcPars->ChannelsQuantity. Write-Read. This field specifies the number of active *logical channels* in the **ControlTable** control table. That is, when data is collected from the ADC, the first AdcPars->ChannelsQuantity of the elements of the AdcPars->ControlTable array will be used. The limit value for this parameter is 256 or MAX_CONTROL_TABLE_LENGTH_E2010. If the AdcPars->ChannelsQuantity value passed to the function exceeds the specified limit value, the function automatically performs the necessary adjustment. And when the function is completed, the number of active logical channels in the AdcPars-> ChannelsQuantity field will be *actually* set.
- Field AdcPars->ControlTable[]. Entry. This field specifies the **ControlTable** control table. That is, the same array of *logical channels* that the module will use when working with the ADC to specify a cyclic sequence of counts from the input channels.
- Field AdcPars->InputRange[]. Entry. This field sets the input ranges for all ADC physical channels of the *E20-10* module. The field is an array of type *WORD*, consisting of 4 (ADC_CHANNELS_QUANTITY_E2010) elements. An array element with index 0 corresponds to the input range of the first ADC channel, etc. Each element of the array can be equal from 0 to 2. You can also use input range constants. The meaningful loading of the values of this field is presented in the table below:

The value of an array element	Constant	Description
0	**ADC_INPUT_RANGE_ 3000mV_E2010**	Input range is ±3000
1	**ADC_INPUT_RANGE_ 1000mV_E2010**	Input range is ±1000
2	**ADC_INPUT_RANGE_ 300mV_E2010**	Input range is ±300

- Field AdcPars->InputSwitch[]. Entry. This field specifies the type of connections or the input switching mode for all physical ADC channels of the *E20-10* module. The field is an array of type *WORD*, consisting of 4 (ADC_CHANNELS_QUANTITY_E2010) elements. An array element with index 0 corresponds to the input range of the first ADC channel, etc. Each element of the array can be equal from 0 to 1. You can also use connection-type constants. The meaningful loading of the values of this field is presented in the table below:

Array element value	Constant	Description
0	**ADC_INPUT_ZERO_E2010**	The input of the ADC channel is switched to the analog ground of the module.
1	**ADC_INPUT_SIGNAL_E2010**	The input signal is transmitted to the ADC channel entrance.

- Fields AdcPars->AdcRate and AdcPars->InterKadrDelay. Write-Read. These fields are valid only when the module uses internal *clock pulses*, which is determined by the *AdcPars->SynchroPars.SynhroSource* field. When entering the function, these fields must contain the required ***time*** parameters for data collection: ADC operational frequency **AdcRate** (inverse value of the *interchannel* delay) and interframe delay **InterKadrDelay**. In this case **AdcRate** is set in *kHz*, and **InterKadrDelay** is set in *ms*. After executing the function ***SET_ADC_PARS()***, these fields return the real values of the inter-channel and inter-frame delay values, which are as close as possible to the initial ones. This is due to the fact that the actual values of **AdcRate** and **InterKadrDelay** are not continuous values, but form a certain frequency spectrum. So, the frequency of the ADC is determined by the following formula: **AdcRate** = 30000/N, where **N** – integer value from 3 to 30. Therefore, this function simply calculates the discrete value closest to the given value **AdcRate** passes it to the module as an integer **N**, and returns its value in the AdcPars->AdcRate field. All the same is true for interframe delay, with the only difference being that it is set in units of 1/**AdcRate**, with the previously corrected **AdcRate**. **InterKadrDelay** can be in the range from 1/**AdcRate** to 255/**AdcRate** (for the *E20-10 module (Rev.'A')*) or 65535/**AdcRate** (for the *E20-10 module (Rev.'B' and high)*). For example, if you set AdcPars->AdcRate=0.0, then ***SET_ADC_PARS()*** sets and returns the lowest possible value for this variable, i.e. 1000.0 *kHz*. Similarly: if set AdcPars->InterKadrDelay=0.0 then this function will set and return the minimum possible inter-frame delay, i.e. 1/AdcPars->AdcRate.
- Field AdcPars->KadrRate. Read. This field returns the frequency of the **KardRate** frame in *kHz*. This field is effective only when the module uses internal *clock pulses*, which is determined by the *AdcPars->SynchroPars.SynhroSource* field. This frequency is calculated based on the AdcPars->ChannelsQuantity, as well as the already adjusted AdcPars->AdcRate and AdcPars->InterKadrDelay. In addition, for the relations between the above values AdcPars->ChannelsQuantity, AdcPars->AdcRate, AdcPars->InterKadrDelay and AdcPars->KadrRate, see *§ 3.2.4. "Format of frame count"*.
- Field AdcPars->AdcOffsetCoefs[][]. Entry. The field is a two-dimensional array of type double, consisting of *ADC_INPUT_RANGES_QUANTITY_E2010* **x** *ADC_CHANNELS_QUANTITY_E2010* elements. For the E20-10 (Rev.'A') module, this field does not carry any functional load. For the module E20-10 (Rev.'B' and high) in this array, the coefficients used by the FPGA module should be located to perform automatic correction of the offset obtained from the ADC data. The resolution to use automatic correction of data set by the field AdcPars->CorrectionEnabled. For details on adjusting the data, see *§ 4.5.1. "ADC data correction"*.
- Field AdcPars->AdcScaleCoefs[][]. Entry. The field is a two-dimensional array of type *double*, consisting of *ADC_INPUT_RANGES_QUANTITY_E2010* **x** *ADC_CHANNELS_QUANTITY_E2010* elements. For the *E20-10 (Rev.'A') module*, this field does not carry any functional load. For the module *E20-10 (Rev.'B' and high)* in *this* array, the coefficients used by the FPGA module should be located to perform automatic correction of the offset obtained from the ADC data. The resolution to use automatic correction of data set by the field AdcPars->CorrectionEnabled. For details on adjusting the data, see *§ 4.5.1. "ADC data correction"*.

Transmitted parameters:

AdcPars is the address of a structure of type *ADC_PARS_E2010* with the required parameters of the data acquisition function from the ADC.

Returned value:	*TRUE* – function was successfully executed; *FALSE* – function was executed with an error.

4.5.5. Getting the current ADC work parameters

Format:	**BOOL**	***GET_ADC_PARS***(*ADC_PARS_E2010 * const AdcPars*)
Assignment: This function reads all the current information from the *E20-10* module, which is used to collect data from the ADC.		
Transmitted parameters: *AdcPars* is the address of a structure of type ADC_PARS_E2010 with the required parameters of the data acquisition function from the ADC.		
Returned value:	*TRUE* – function was successfully executed; *FALSE* – function was executed with an error.	

4.5.6. ADC data acquisition

Format: **BOOL** ***ReadData***(*IO_REQUEST_LUSBAPI * const ReadRequest*)

Assignment:

This function is designed to obtain the next portion of data from the ADC module. This function must be used in conjunction with the START_ADC() and STOP_ADC() functions.

The fields of the transmitted structure type IO_REQUEST_LUSBAPI define the parameters and the required mode for obtaining data from the *E20-10* module. The assignments for the fields of this structure are given in the table below:

Field name	Description
Buffer	Data buffer. Read. **Buffer** is intended for storage of data received from the module of ADC. Before using it in a function, the application itself must take care of allocating a sufficient amount of memory for this buffer. The received data in the buffer will be located in a frame-by-frame manner: 1st frame, 2nd frame and so on. And the position of the counts in the frames will be the same as the ordering of the corresponding *logical channels* in the **ControlTable** control table.
NumberOf-WordsToPass	Number of transmitted data. Write-Read. This parameter specifies the number of samples of the ADC, which this function simply must try with the module. Depending on the revision of the module, this parameter has the following limitations: • for module *E20-10 (Rev.'A')*, the NumberOfWords-ToPassNumberOf- value must be in the range from 256 to (1024 * 1024), and also WordsToPass must be a multiple of 256; • for the module *E20-10 (Rev. 'B' and above)* value NumberOfWordsToPass should be in the range from 1 to (1024 * 1024).

	Otherwise, this function corrects the value of this field itself, and upon returning from the function it will *actually* be the used value of the number of requested data.
NumberOf-WordsPassed	Number of transmitted data. Read. In this parameter, the number of ADC counts that this function actually received from the module is returned. For the *asynchronous* mode of this function (see the field Overlapped below WordsPassed), the number 0 may well return in this parameter, which *is not* an error, given the specifics of this mode.
Overlapped	Structure Overlapped. This field determines in which mode this function will be executed: *synchronous* or *asynchronous*: Overlapped = NULL. In this case, the function requires a *synchronous* execution mode. In this case, the function honestly tries to get all the requested data from the module, and during this time the function does not return control to the application that caused it. If during the TimeOut time *ms* (see below) all the required data from the module is not received, the function terminates and returns an error. Overlapped ≠ NULL. In this case, the function requires an *asynchronous* execution mode. It is assumed that the application has already assigned a pointer to a pre-prepared structure of type *OVERLAPPED*. In this mode, this function exposes the system, i.e. *Windows*, *asynchronous* request to get the required number of data from the module and immediately returns the control to the application. That is, there is as it were a complete shift of the task of collecting data on the *core* of the system. Since an *asynchronous* request is already executed at the *core* level, while it is processing it, the application can fully handle its own tasks. The end of the current *asynchronous* query application can be monitored using standard *Windows API* functions such as: *WaitForSingleObject()*, *GetOverlappedResult()* or *HasOverlappedIoCompleted()*. These functions use the *Event* event, which must previously have been defined by the application in the corresponding field of the Overlapped structure. Event *Event* is activated by the system at the end of the collection of *all* requested data, thus completing the current *asynchronous* request. In some cases, it may simply be necessary to interrupt the running *asynchronous* request. For this purpose, and there is a regular *Windows API* function *CancelIo()*. Unfortunately, this function exists only on *Windows NT* systems.
TimeOut	Waiting time for data collection. This field is intended for use only in *synchronous* mode. It specifies the maximum time in *ms* for waiting for the completion of a *synchronous* request to collect the required quantity of the data. If after this time *all* data requested by the request is not received, the function completes and returns an error.

On the *E20-10* module, an internal hardware *FIFO* data buffer of 8 MB is installed. Such a large buffer requires reliable data collection at large input frequencies. So, at collection frequencies of the order of 10 MHz, the buffer overflow will occur only after 400 *ms*, which is a sufficiently long period of time even for such a "*wistful*" system like *Windows*. Now we should mention some specific features of the modes of this function:

1. *Synchronious* mode. This mode is recommended to be used when organizing a single data collection, in which the number of counts does not exceed 1024 * 1024 = 1 M words. In this mode, the ***ReadData()*** function should only be called after the successful execution of the START_ADC() function, which, in principle, must be preceded by the call to the function STOP_ADC(). It is necessary to use this mode with great care at sufficiently slow collection frequencies and a large amount of requested data. Otherwise, this function can '*go*' for a long time waiting for the data collection to complete and, therefore, for a very long time, do not return control to the application. An example of the correct use of the regular functions of the `Lusbapi` library in *synchronous* mode as a normal console application can be found on our proprietary CD-ROM in the directory `\E20-10\Examples\Borland C++ 5.02\ReadDataSynchro`.
2. *Asynchronous* mode. This mode is functionally much more flexible than *synchronous* mode and it is recommended to use it when organizing various algorithms for continuous *streaming* data collection, when the number of entered counts exceeds 1 M per word. This mode, for example, allows you to organize a queue of *asynchronous* requests on the *Windows* system. So you can generate a queue of preliminary queries even immediately before starting the data collection, but after the function STOP_ADC(). Using the query queue can dramatically improve the reliability of data collection. The *Windows* operating system is not, as they say, a real-time environment. Therefore, working in it, as it usually happens, only at the *user* level, and not at the *core* level, you can never be completely sure that the system at the right time will not be distracted by its own needs for a more or less long period of time. For example, if for a collection frequency of 10 MHz after START_ADC(), but before starting the ***ReadData()*** function, the system _*thought* for more than 400 ms (which is very rare, but it is possible), then the failure in the data received is almost obvious. This data failure is manifested as a data integrity violation and can be tracked using the GET_DATA_STATE() function. However, if several (or one) preliminary requests can be set up with the help of ReadData() just before ***START_ADC()***, which will be processed at the *core* level of the system, there will be no failures. This is because the response time for working out some event (in our case, a request) at the *core* level is much less than at the *user* level. So, it turns out that after performing the function START_ADC() we already have ready-to-service requests at the *core* level of the system. There are almost no delays. And now, as long as the system fulfills our preliminary requests, you can take the time to submit one or more of the following requests as required. It is important to understand that for each queued or already running request, the application must have its own instance of a structure of type IO_REQUEST_LUSBAPI with its individual event *Event*.

Transmitted parameters:
ReadRequest – structure of type IO_REQUEST_LUSBAPI with parameters of extraction of ADC finished data from the module *E20-10*.

Returned value: *TRUE* – function was successfully executed;
FALSE – failure in the function execution.

4.5.7. Check the status of the data collection process

Format:	**BOOL**	***CHECK_DATA_INTERGRITY***(*BYTE * const DataIntegrity*) (version 3.1 and below)
	BOOL	***GET_DATA_STATE***(*DATA_STATE_E2010 * const DataState*) (version 3.2 and higher)

Assignment:
This function allows you to get the current state of the data collection process in the structure of type DATA_STATE_E2010. The format of the structure of DATA_STATE_E2010 is given earlier in § 4.3.7. "Structure of the DATA_STATE_E2010", and the purpose of its individual fields is described in more detail below.

- Field DataState->ChannelsOverFlow. Read. For the *E20-10 (Rev.'A')* module, this field does not carry any functional load. For *E20-10 module (Rev.'B' and above)* with this field, you can get global (for all time of collection) and local (during one request) bit attributes of the bit grid overflow. The global bit flag is activated (goes into the "1" state) when the bit grid overflows at any of the 4 physical ADC channels for the entire time interval from START_ADC () and up to STOP_ADC (). Each of their local bit attributes is activated (goes into the state of the log "1") when the bitmap overflow occurs at the corresponding physical ADC channel during the time of one ReadData() request. As the numbers of the bits used, you can use the constants of the bit numbers of the channel overload. The meaning load of the bits of this field is shown in the table below:

Bit number	Constant name	Intended purpose
0	**OVERFLOW_OF_CHANNEL_1_E2010**	Local sign of the word size overflow of the 1st physical ADC channel.
1	**OVERFLOW_OF_CHANNEL_2_E2010**	Local sign of the word size overflow of the 2nd physical ADC channel.
2	**OVERFLOW_OF_CHANNEL_3_E2010**	Local sign of the word size overflow of the 3rd physical ADC channel.
3	**OVERFLOW_OF_CHANNEL_4_E2010**	Local sign of the word size overflow of the 4th physical ADC channel.
<4..6>	———————	Reserved
7	**OVERFLOW_E2010**	Global flag for word size overflow.

- Field DataState->BufferOverrun. Read. *E20-10* allows you to monitor the global sign of an overflow of the internal hardware buffer of the module. And the module keeps track of this feature for the entire time interval from the moment START_ADC() and up to STOP_ADC(). This information is reflected in the bit with the number 0 or **BUFFER_OVERRUN_E2010** of this structure field. The appearance of the logical state '1' in this bit indicates that during the data acquisition time the internal buffer of the module has overflowed. In this case, the version of the main MCU program of the module should be 1.7 or higher. In addition, if the module detects buffer overflow during collection with the ADC, it visually informs about this situation by flashing its LED indicator in red (for *E20-10 module (Rev.'A')*) or red-green (for *E20-10 module (Rev.'B' and above)*).
- Field DataState->CurBufferFilling. Read. For the *E20-10 (Rev.'A')* module, this field does not carry any functional load. For the module *E20-10 (Rev.'B' and above)*, this field contains the current occupancy of the internal hardware buffer of the module. It is expressed in counts.
- Field DataState->MaxOfBufferFilling. Read. For the *E20-10 (Rev.'A')* module, this field does not carry any functional load. For the *E20-10 module (Rev.'B' and above)*, this field shows which maximum occupancy of the internal hardware buffer of the module was achieved during the entire data acquisition interval from the START_ADC() and up to STOP_ADC(). It is expressed in counts.
- Field DataState->BufferSize. Read. For the *E20-10 (Rev.'A')* module, this field does not carry any functional load. For *E20-10 module (Rev.'B' and above)* this field contains the full size of the internal hardware buffer of the module. It is expressed in counts.
- Field DataState->CurBufferFillingPercent. Read. For the *E20-10 (Rev.'A')* module, this field does not carry any functional load. For module *E20-10 (Rev.'B' and above)*, this field shows the percentage level of the current occupancy of the internal hardware buffer. Expressed in %.
- Field DataState->MaxOfBufferFillingPercent. Read. For the *E20-10 (Rev.'A')* module, this field does not carry any functional load. For the *E20-10 module (Rev.'B' and above)*, this field shows what the maximum percentage level of internal hardware buffer occupancy has been reached during the entire data collection interval from START_ADC() and up to STOP_ADC(). Expressed in %.

Transmitted parameters:
DataState – the returned structure, with the current state of the data collection process.

Returned value: *TRUE* – function was successfully executed;
FALSE – function was executed with an error.

4.6. *Functions for working with the DAC*

The hardware of the *E20-10* module and, accordingly, the Lusbapi library allows you to control the output of the data to the DAC only in asynchronous (one-time) mode. So, the output to the DAC is obtained by a relatively slow operation, as the module does not implement hardware support for *streaming* work with the DAC.

Examples of the correct application of the interface function for working with the DAC can be found in the directory *E20-10*\Examples\Borland C++ 5.02\DacSample.

4.6.1. ADC data correction

The circuitry and the components used ensure the linearity of the transmission characteristic of the DAC of the *E20-10* module. However, *for now*, the module does not know how to automatically adjust the output to the DAC. This leads to the fact that the DAC output reading can have some mixing of zero and inaccuracy in the transmission scale. Therefore, at the application level, it is necessary to implement the whole tedious task of updating the DAC data. To this end, the appropriate calibration factors stored in the service information of the module are intended. Service information together with the required coefficients is recorded in the module at the stage when it is set up at LLC "L-Card". Due to this, there are no trimming resistors on the module, which improves the noise characteristics of the module and increases their reliability.

The coefficients themselves are located in the fields *Dac.OffsetCalibration[]* and *Dac.ScaleCalibration[]* of the structure of the service information MODULE_DESCRIPTION_E2010. These fields are arrays of type *double*. For the *E20-10* module, only the first DAC_CALIBR_COEFS_QUANTITY_E2010 elements are used in each of these arrays. The *Dac.OffsetCalibration* array contains the coefficients for correcting the zero offset of the first and second DAC channels, and the *Dac.ScaleCalibration* array for scale correction.

ADC data correction is performed as follows: Y = (X+A)*B, where: X – uncorrected DAC data [in DAC codes], Y – corrected DAC data [in DAC codes], A – zero offset coefficient [in DAC codes], B – scale factor [unsized]. For example, on the second DAC channel, it is necessary to set the voltage corresponding to the following DAC codes: X1 = 1000, X2 = -1000, X3 = 0. Then, the adjustment coefficients and the data for the second DAC channel can be represented as follows: A = Dac.OffsetCalibration[1], B = Dac.ScaleCalibration[1], Y1=(A+1000)*B, Y2=(A-1000)*B, Y3=A*B.

4.6.2. Single output to the DAC

Format:	**BOOL**	***DAC_SAMPLE***(*SHORT * const DacData, WORD DacChannel*)
Assignment: This function allows you to set the voltage on the specified channel *DacChannel* according to the *DacData* value (in the DAC's codes). The *DAC_SAMPLE()* function is executed quite *slowly* and, using it, you can achieve the frequency of data output to the DAC of the order of several hundred *Hz*. For compliance of the DAC code with the value of the analog voltage module installed at the output, see § 3.2.2. "Word format for DAC data"		
Transmitted parameters: • *DacData* – the set voltage value in the DAC codes (from -2048 to 2047). • *DacChannel* – requested channel number for the DAC (0 or 1).		
Returned value:	*TRUE* – function was successfully executed; *FALSE* – function was executed with an error.	

4.7. Functions for working with digital lines

All input and output digital lines of the *E20-10* module are located on the external connector ***DIGITAL I/O***. If no special *EN_OE* line is used at this connector, (see User Manual), by default, immediately after the external power supply is applied to the module, the digital output lines are in *high-impedance* state. If this line *EN_OE* is properly used, then after the power is applied, *all* output lines become active.

The hardware of the *E20-10* module and, accordingly, the `Lusbapi` library allows you to work with digital lines only asynchronously (one-time). So, work with digital lines is obtained by a relatively slow operation, as the module does not provide hardware support for *streaming* work with them.

4.7.1. Resolution of output digital lines

Format: BOOL ***ENABLE_TTL_OUT***(*BOOL EnableTtlOut*)
Assignment: This interface function allows you to control the resolution of *all* output lines of the external digital ***DIGITAL I/O*** connector. So, there is a possibility of transferring them to the third (*high-impedance*) state and back. If the special *EN_OE* line of the ***DIGITAL I/O*** connector is properly used (see User Manual), then this function does not have any influence on the module operation.
Transmitted parameters: *EnableTtlOut* – flag that controls the status of the resolution of all digital output lines.
Returned value: *TRUE* – function was successfully executed; *FALSE* – function was executed with an error.

4.7.2. Reading of the external digital lines

Format: BOOL ***TTL_IN***(*WORD * const TtlIn*)
Assignment: This interface function performs a single asynchronous reading of the states of *all* 16 input digital lines on the external ***DIGITAL I/O*** connector. The ***TTL_IN()*** function is *slow* enough and, using it, you can achieve the frequency of data entry from digital lines on the order of a few hundred *Hz*.
Transmitted parameters: *TtlIn* – a variable containing the bit-state of the input digital lines.
Returned value: *TRUE* – function was successfully executed; *FALSE* – function was executed with an error.

4.7.3. Output to external digital lines

Format: BOOL ***TTL_OUT***(*WORD TtlOut*)
Assignment: This interface function establishes the installation of *all* 16 digital output lines on the external ***DIGITAL I/O*** connector of the *E20-10* module in accordance with the bits of the transmitted parameter *TtlOut*. If necessary, the work with the digital outputs must first be enabled using the interface function ENABLE_TTL_OUT(). The *TTL_OUT()* function is executed quite *slowly* and, using it, it is possible to achieve a frequency of data output to digital lines of the order of several hundred *Hz*.
Transmitted parameters: *TtlOut* – a variable containing the bit-state of the input digital lines.
Returned value: *TRUE* – function was successfully executed; *FALSE* – function was executed with an error.

4.8. Functions for working with the user PROM

On the *E20-10* module, part of the microcontroller's memory is allocated to the user's PROM. The size of this area is USER_FLASH_SIZE_E2010 bytes. The user can safely use all this area in their purely private property interests.

4.8.1. Permission to write to the PROM

Format:	**BOOL**	***ENABLE_FLASH_WRITE***(*BOOL IsUserFlashWriteEnabled*)
Assignment: This interface function allows (*TRUE*) or disables (*FALSE*) the write mode in the user PROM using the standard interface function WRITE_FLASH_ARRAY(). It should be remembered that after completing all the required operations for writing information to the user's PROM, it is necessary to disable the recording mode with this interface function.		
Transmitted parameters: *EnableFlashWrite* – a variable can take the following values: ✓ if *TRUE*, then the writing mode in the user PROM is allowed, ✓ if *FALSE*, then the writing mode in the user PROM is forbidden.		
Returned value:	*TRUE* – function was successfully executed; *FALSE* – function was executed with an error.	

4.8.2. Data writing to the PROM

Format:	**BOOL**	***WRITE_FLASH_ARRAY***(*USER_FLASH_E2010 * const UserFlash*)
Assignment: This interface function writes a byte array size USER_FLASH_SIZE_E2010 to the PROM. So, the *entire* accessible area of the user PROM is overwritten immediately. Before starting the write procedure in the user's PROM, you must enable this operation using the interface function ENABLE_FLASH_WRITE(). After finishing the procedure of writing all the required information, you must disable the recording mode with the same function ENABLE_FLASH_WRITE().		
Transmitted parameters: *UserFlash* – in fact, this is a byte array that must be written to the user's PROM.		
Returned value:	*TRUE* – function was successfully executed; *FALSE* – function was executed with an error.	

4.8.3. Data reading from the PROM

Format:	**BOOL**	***READ_FLASH_ARRAY***(*USER_FLASH_E2010 * const UserFlash*)
Assignment: This interface function reads the contents of the entire area of the user PROM.		
Transmitted parameters: *UserFlash* – in this byte array, the image of the entire user PROM is returned.		
Returned value:	*TRUE* – function was successfully executed; *FALSE* – function was executed with an error.	

4.9. Functions for working with service information

The service information contains the most general data about the *E20-10* module used: the name of the module, its serial number and revision, the adjustment coefficients for the ADC and DAC, the versions of the FPGA and MCU firmware used, the clock frequencies of the actuators (FPGA, MCU), and so on. Some data from this service information is needed by the functions of the regular `Lusbapi` library for its correct operation.

4.9.1. Reading service information

Format: **BOOL** ***GET_MODULE_DESCRIPTION***(*MODULE_DESCRIPTION_E2010 * const ModuleDescription*)	
Assignment: This interface function reads all the service information into a structure of type MODULE_DESCRIPTION_E2010. This information is required when working with some of the interface functions of the regular `Lusbapi` library. Therefore, this function, in order to avoid unpredictable behavior of applications, should be called immediately after loading the FPGA module and verifying its operability (see § 4.1. "General principles of working with the module")	
Transmitted parameters: *ModuleDescription* is a pointer to a structure of type MODULE_DESCRIPTION_E2010, into which all the service information of the module is read.	
Returned value:	*TRUE* – function was successfully executed; *FALSE* – function was executed with an error.

4.9.2. Service information writing

Format: **BOOL** ***SAVE_MODULE_DESCRIPTION***(*MODULE_DESCRIPTION_E2010 * const ModuleDescription*)	
Assignment: This interface function allows storing in the module all the service information from a structure of the type MODULE_DESCRIPTION_E2010. **!!!Attention!!!** Use this function only in case of emergency. For example, when, for one reason or another, the contents of the official information deteriorated.	
Transmitted parameters: *ModuleDescription* – a pointer to a structure of type MODULE_DESCRIPTION_E2010, from which the service information is transferred to the module.	
Returned value:	*TRUE* – function was successfully executed; *FALSE* – function was executed with an error.

Annex A. AUXILIARY CONSTANTS AND TYPES

Auxiliary constants and types data are described in a header file `\DLL\Include\LusbapiTypes.h` and considered in the sections below.

A.1. Constants

The auxiliary constants defined in the `Lusbapi` library are listed in the following table:

Name	Value	Meaning
NAME_LINE_LENGTH_LUSBAPI	25	The length of the line with the name of something. For example, the name of the manufacturer or product, the name of the author, etc.
COMMENT_LINE_LENGTH_ LUSBAPI	256	The length of the line with the comment in some auxiliary structure.
ADC_CALIBR_COEFS_QUANTITY_ LUSBAPI	128	The maximum possible number of ADC adjustment coefficients.
DAC_CALIBR_COEFS_QUANTITY_ LUSBAPI	128	The maximum possible number of DAC adjustment coefficients.

A.2. Structure of the VERSION_INFO_LUSBAPI

The auxiliary structure of the *VERSION_INFO_LUSBAPI* contains more or less detailed information about the software running in any executive device: MCU, DSP, PLD, etc. This structure is described as follows:

```
struct VERSION_INFO_LUSBAPI
{
 BYTE Version[10];    // version of the software for the executive device
 BYTE Date[14];       // software assembly date
 BYTE Manufacturer[NAME_LINE_LENGTH_LUSBAPI]; // software manufacturer
 BYTE Author[NAME_LINE_LENGTH_LUSBAPI];       // software author
 BYTE Comment[COMMENT_LINE_LENGTH_LUSBAPI];   // comment string };
```

A.3. Structure of the MCU_VERSION_INFO_LUSBAPI

The auxiliary structure of the *MCU_VERSION_INFO_LUSBAPI* consists of two structures VERSION_INFO_LUSBAPI and contains information about the software of the executive device, which includes information about the firmware of both the main program (*Firmware*) and the bootloader (*Bootloader*). This structure is described as follows:

```
struct MCU_VERSION_INFO_LUSBAPI
{
    VERSION_INFO_LUSBAPI FwVersion;    // main program version (Firmware)
    VERSION_INFO_LUSBAPI BlVersion;    // bootstrap version (BootLoader)
};
```

A.4. Structure of the MODULE_INFO_LUSBAPI

This auxiliary structure of the *MODULE_INFO_LUSBAPI* contains the most general information about the module: the name of the manufacturer of the product, the name of the product, the serial number of the product, the product revision and the comment line. This structure is described as follows:

```
struct MODULE_INFO_LUSBAPI
{
    BYTE CompanyName[NAME_LINE_LENGTH_LUSBAPI];   // name of the manufacturer
                                                  // of the product
    BYTE DeviceName[NAME_LINE_LENGTH_LUSBAPI];    // product name
    BYTE SerialNumber[16];                        // product serial number
    BYTE Revision;                                // product revision
 BYTE Modification;  // execution (option) of the module;
 BYTE Comment[COMMENT_LINE_LENGTH_LUSBAPI];       // comment line
};
```

A.5. Structure of the INTERFACE_INFO_LUSBAPI

The auxiliary structure of the *INTERFACE_INFO_LUSBAPI* contains the most general information about the interface used to access the module. This structure is described as follows:

```
struct INTERFACE_INFO_LUSBAPI
{
 BOOL Active;   // the validity flag of the rest of the structure fields
 BYTE Name[NAME_LINE_LENGTH_LUSBAPI];             // interface name
 BYTE Comment[COMMENT_LINE_LENGTH_LUSBAPI];       // comment line
};
```

A.6. Structure of the MCU_INFO_LUSBAPI

The auxiliary structure of the *MCU_INFO_LUSBAPI* contains the most general information about the operating device used, such as a microcontroller (MCU). This structure is described as follows:

```
template <class VersionType> struct MCU_INFO_LUSBAPI
{
 BOOL Active;   // confidence flag of the rest of the structure fields
 BYTE Name[NAME_LINE_LENGTH_LUSBAPI]; // MCU name
 double ClockRate;                    // MCU clock frequency in kHz
 VersionType Version;                 // information about Firmware and BootLoader
 BYTE Comment[COMMENT_LINE_LENGTH_LUSBAPI]; // comment line };
```

A.7. Structure of the PLD_INFO_LUSBAPI

The auxiliary structure of the *PLD_INFO_LUSBAPI* contains the most general information about the operating device used such as a programmable logic integrated circuit (FPGA). This structure is described as follows:

```
struct PLD_INFO_LUSBAPI
{
 BOOL Active;  // the validity flag of the remaining fields of the structure
 BYTE Name[NAME_LINE_LENGTH_LUSBAPI];  // FPGA name
```

```
 double ClockRate;                               // clock frequency kHz
 VERSION_INFO_LUSBAPI Version;                   // information about FPGA firmware
 BYTE Comment[COMMENT_LINE_LENGTH_LUSBAPI]; // comment line
};
```

A.8. Structure of the ADC_INFO_LUSBAPI

The auxiliary structure of the *ADC_INFO_LUSBAPI* contains the most general information about the device used in the ADC type. This structure is described as follows: **struct** *ADC_INFO_LUSBAPI*

```
{
 BOOL Active;   // the validity flag of the remaining fields of the structure
 BYTE Name[NAME_LINE_LENGTH_LUSBAPI];    // ADC name
 double OffsetCalibration[ADC_CALIBR_COEFS_QUANTITY_LUSBAPI];
                                  // zero offset adjustment coefficients for ADC
 double ScaleCalibration[ADC_CALIBR_COEFS_QUANTITY_LUSBAPI];
                                 // adjustment coefficients of the ADC scale
 BYTE Comment[COMMENT_LINE_LENGTH_LUSBAPI]; // comment line
};
```

A.9. Structure of the DAC_INFO_LUSBAPI

The auxiliary structure of the *DAC_INFO_LUSBAPI* contains the most general information about the device used for the DAC type. This structure is described as follows: **struct** *DAC_INFO_LUSBAPI*

```
{
 BOOL Active;                            // validity flag of the rest of the structure fields
 BYTE Name[NAME_LINE_LENGTH_LUSBAPI]; // DAC name
 double OffsetCalibration[DAC_CALIBR_COEFS_QUANTITY_LUSBAPI];
                                 // zero offset adjustment coefficients for ADC
 double ScaleCalibration[DAC_CALIBR_COEFS_QUANTITY_LUSBAPI];
                                 // adjustment coefficient of the DAC scale
 BYTE Comment[COMMENT_LINE_LENGTH_LUSBAPI]; // comment line
};
```

A.10. Structure of the DIGITAL_IO_INFO_LUSBAPI

The auxiliary structure of the *DIGITAL_IO_INFO_LUSBAPI* contains the most general information about the digital I/O devices used. This structure is described as follows:

struct *DIGITAL_IO_INFO_LUSBAPI*

```
{
 BOOL Active;                    // the validity flag of the rest of the structure fields
 BYTE Name[NAME_LINE_LENGTH_LUSBAPI];        // digital microcircuit name
 WORD InLinesQuantity;                       // number of the input lines
 WORD OutLinesQuantity;                      // number of the outline lines
 BYTE Comment[COMMENT_LINE_LENGTH_LUSBAPI]; // comment string
};
```