P-IRIS CONTROLLER

C++ SDK

Manual

atesystem
FOCUSED ON DETAIL

ATEsystem s.r.o.
Studentská 6202/17
708 00 Ostrava-Poruba
Czech Republic

M +420 595 172 720
E produkty@atesystem.cz
W www.atesystem.cz

DOCUMENT INFORMATION

Revision No.	Author	Revision date	Description
0	Jandásek V.	25 April 2019	Document creation

Appendixes

Notes

Contact

ATEsystem s.r.o.
Studentská 6202/17
708 00 Ostrava 8 – Poruba
Czech Republic

T +420 595 172 720
F +420 595 170 100
E produkty@atesystem.cz
W www.atesystem.cz

TABLE OF CONTENTS

1 DESCRIPTION

ATEsystem.PIRIS-driver is part of software support for P-IRIS Controller product. It serves as a main interface between the hardware and the control application. The driver allows complete control of the controller unit in both **UART/Ethernet** and **RS232** version. It implements a proprietary ATEsystem communication protocol which is described in the datasheet of the controller. The driver is written in C++17 and is distributed in the form of source code under MIT licence. The software requires following dependencies: **Pylon 5.0.12** and **VC++ Redistributable 2013 and 2017** when using Windows OS. The driver uses **Serial** open-source software. The dependencies are distributed in two folders, *bin* and *install*, and are necessary for proper function. The driver (including its dependencies) is multi-platform. The distribution includes executable examples for both Windows OS (x86, amd64) and Linux OS (x86, amd64, armhf, arm64). The SDK has been developed in Microsoft Visual Studio 2017 IDE which allows multi-platform development of C++ applications. IDE settings and further details can be found below.

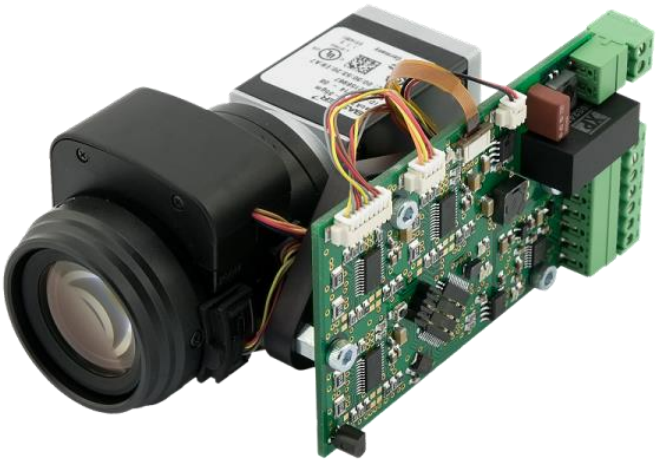
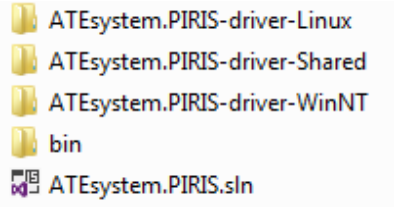


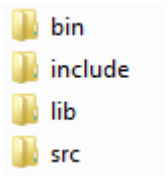
Fig. 1 – Control unit for the lens with stepper motors

2 INSTALLATION

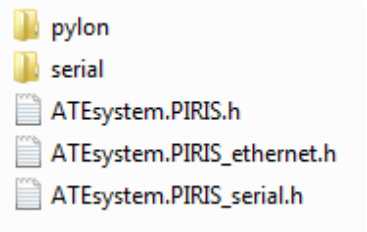
Upon extraction of the distributed archive open the *Solution* in Visual Studio 2017 using the included SLN file. **Linux** and **WinNT** folders include VS projects with corresponding settings of compiler and linker, separately for x86, amd64, armhf and arm64 architectures. **Shared** folder is the main container which includes the source code of the driver, header files, libraries and examples. **Bin** folder includes executable examples.



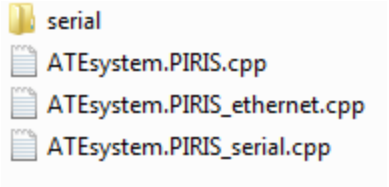
The source code of the driver is divided in four folders – *bin*, *include*, *lib* and *src*. **Bin** includes separate binary files for each of the supported architectures. The binaries are Pylon and GenICam dynamic libraries, which must be installed on the target machine. There is also a batch file prepared for Windows OS, however it is acceptable to install Pylon 5 runtime from the official distribution. **Lib** includes static libraries for Windows OS.



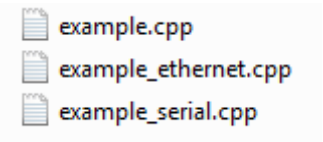
Include incorporates header files for the driver, ATEsystem.PIRIS is the main header file and the only one, which must be included in the application which uses the driver. **Pylon** and **serial** include corresponding header files for UART/Ethernet and RS232 unit.



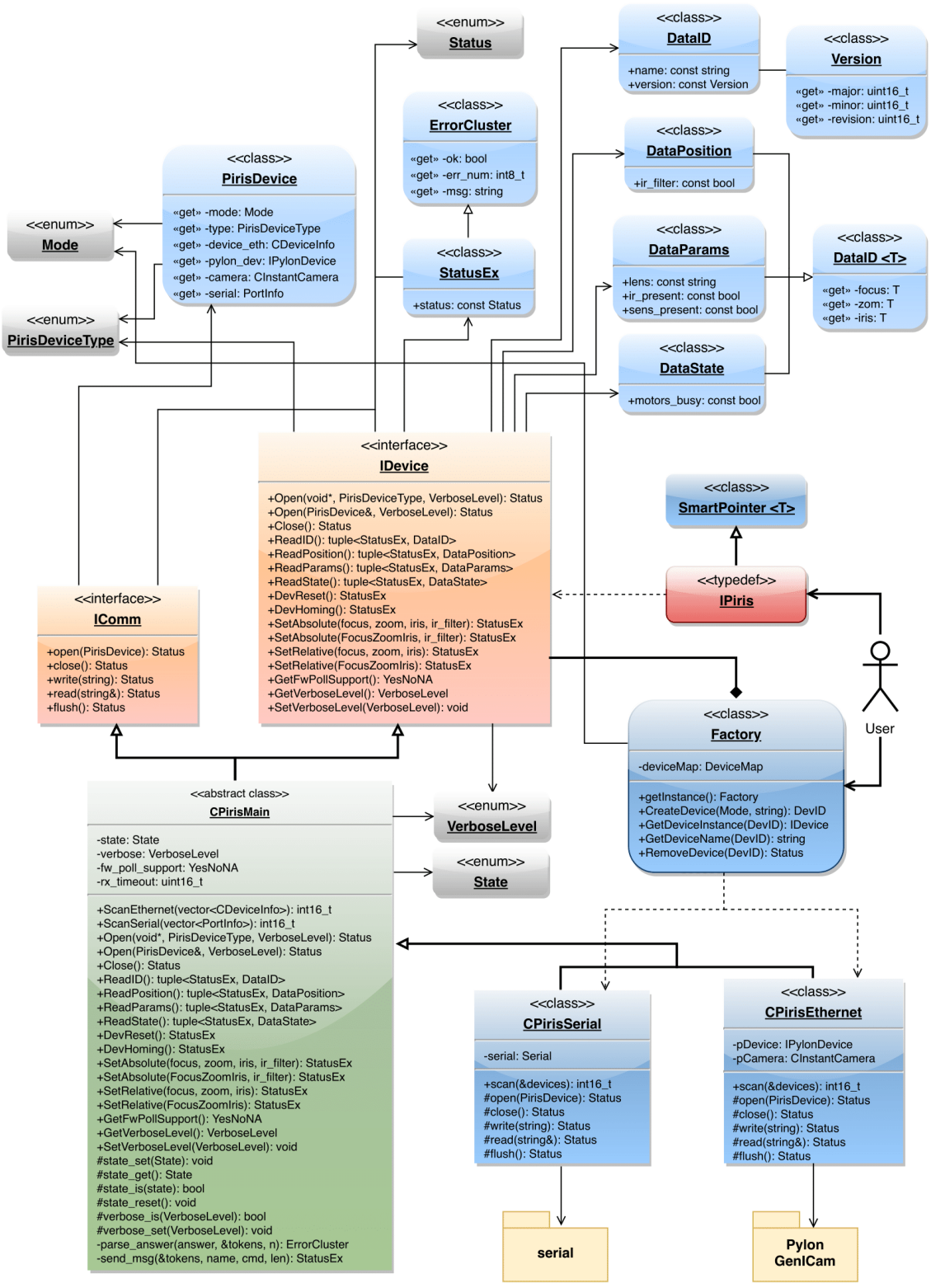
Src includes complete source code. The source code of the driver is located directly in the root of this folder, source files for multiplatform manipulation with serial ports (created by William Woodall and John Harrison) are in folder **serial**.



The root folder includes a few demonstrations of usage of the driver, named **example**. File *example.cpp* is the main demonstration file, files with suffixes *_ethernet* and *_serial* include examples based on the chosen mode. Corresponding executable binary files of this example can be found **/bin** folder in the root.



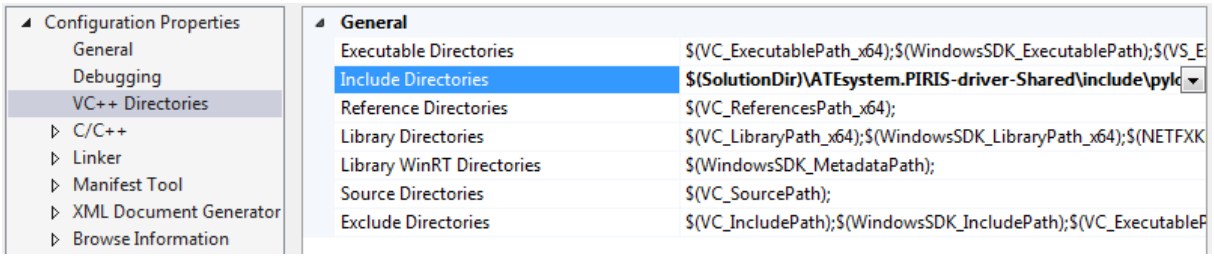
3 CLASS DIAGRAM



4 IDE SETTINGS

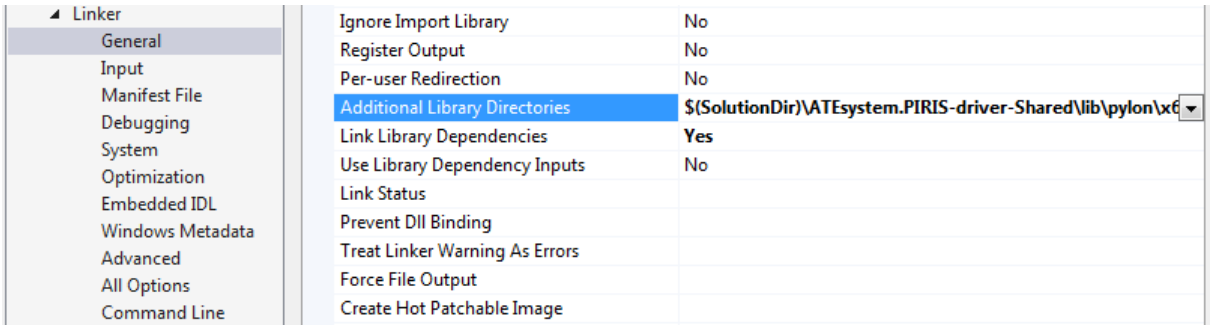
Build tools must be properly set up in order to create an executable file from the source code. Settings of the compiler, linker and other important parameters is described as in Microsoft Visual Studio 2017 Enterprise IDE.

Settings in *Configuration Properties* > *VC++ Directories* > **Include Directories** must be set as follows:



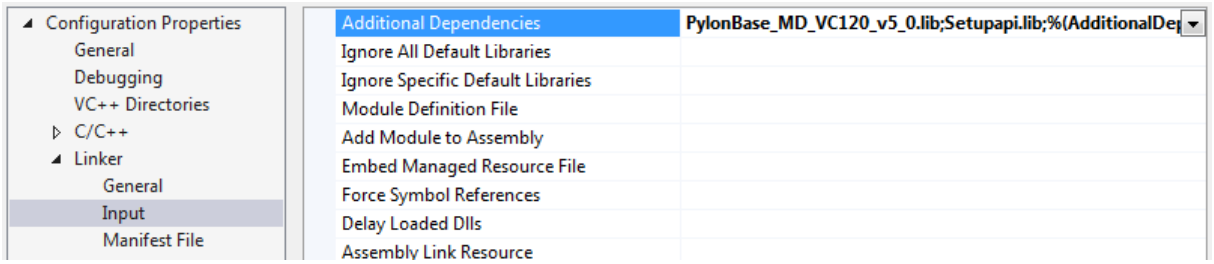
1) $(SolutionDir)\ATEsystem.PIRIS-driver-Shared\include\pylon\WinNT
2) $(SolutionDir)\ATEsystem.PIRIS-driver-Shared\include\serial
3) $(SolutionDir)\ATEsystem.PIRIS-driver-Shared\include

Value in *Configuration Properties* > *Linker* > *General* > **Additional Library Directories** must be changed to the path of the static Pylon library from lib/pylon (for corresponding architecture).



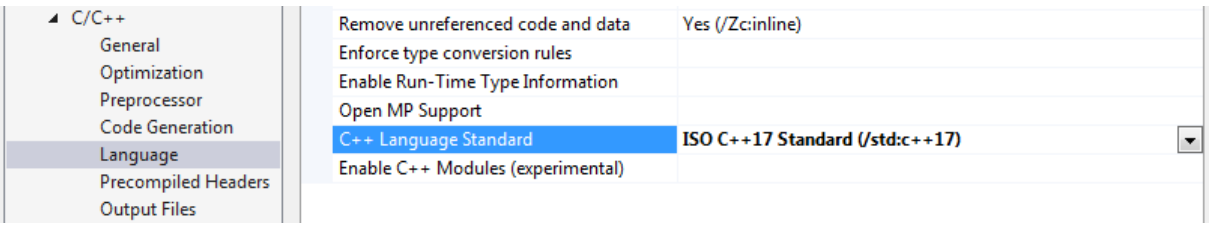
1) $(SolutionDir)\ATEsystem.PIRIS-driver-Shared\lib\pylon\Win32

Settings in *Configuration Properties* > *Linker* > *Input* > **Additional Dependencies** must include the name of static Pylon library and in case of Windows-based target also the name of *Setupapi* library.



1) PylonBase_MD_VC120_v5_0.lib
2) Setupapi.lib

Value in *Configuration Properties* > *C/C++* > **Language** must be set to *ISO C++17 Standard*.



- ISO C++17 Standard (/std:c++17)

5 USAGE

In order to successfully use the driver following objects are necessary: `Factory` (Singleton) class and `IPiris` interface, which is in fact implemented as a Smart Pointer; data containers (`DataID`, `DataPosition`, `DataParams` a `DataState`), which are returned using `std::tuple` collection and `StatusEx` object, which carries basic information on performed action, including error code, error message of the device – `ErrorCluster` and error message of the driver – `Status`.

`Factory` is a class based on the Factory design pattern. It serves to create instances of the driver for particular communication interface (UART/Ethernet – `CPirisEthernet`, RS232 – `CPirisSerial`). Multiple such instances can be created and used parallelly with the help of one instance of the factory, `DevID_t` serves as an identifier. The identifier can be used to create or delete the instances or to access the interfaces of the instance. As the interface is implemented as a Smart Pointer, there is no need to call `delete`, nor to use *`RemoveDevice()`* function, which is suitable for management of more than one instance of the driver. This system allows to use more P-IRIS Controller units at once, transparently and without any risk of memory leak.

```
DevID_t CreateDevice(Mode mode = Mode::ETHERNET);
DevID_t CreateDevice(const std::string& name, Mode mode);
IDevice* GetDeviceInstance(DevID_t id);
std::string GetDeviceName(DevID_t id);
Status RemoveDevice(DevID_t id);
```

`IPiris` is the main interface for manipulation with the driver instance. It is actually a Smart Pointer of type IDevice `typedef SmartPointer<IDevice> IPiris`. Functions of this interface are described in the table below. `SetAbsolute()` function is important to set values of the parameters, it can be called either directly with numeric values for focus, zoom, aperture and IR filter or with usage of class `FocusZoomIris<T>`, which wraps up these values into a generic container. This container serves also to return the values and state information from the functions `ReadPosition()`, `ReadParams()`, `ReadState()`, therefore it is sometimes more advantageous to use the first way, whereas in other cases the container might serve better. `SetRelative()` function can be used in some cases, which sets, unlike the previous function, relative values.

It is essential to call *ReadID()* directly after *Open()*, in order to correctly detect compatible firmware version.

```
Interface IDevice
{
public:

    virtual Status Open(void* dev, PirisDeviceType type, VerboseLevel verbose) = 0;
    virtual Status Open(const PirisDevice& dev, VerboseLevel verbose) = 0;
    virtual Status Close() = 0;

    virtual std::tuple<StatusEx, DataID> ReadID() = 0;
    virtual std::tuple<StatusEx, DataPosition> ReadPosition() = 0;
    virtual std::tuple<StatusEx, DataParams> ReadParams() = 0;
    virtual std::tuple<StatusEx, DataState> ReadState() = 0;

    virtual StatusEx DevReset() = 0;
    virtual StatusEx DevHoming() = 0;

    virtual StatusEx SetAbsolute(uint16_t focus = 0,
                                 uint16_t zoom = 0,
                                 uint16_t iris = 0,
                                 bool ir_filter = false) = 0;
    virtual StatusEx SetAbsolute(const FocusZoomIris<uint16_t>& values,
                                 bool ir_filter = false) = 0;
    virtual StatusEx SetRelative(int16_t focus = 0,
                                 int16_t zoom = 0,
                                 int16_t iris = 0) = 0;
    virtual StatusEx SetRelative(const FocusZoomIris<int16_t>& values) = 0;

    virtual YesNoNA GetFwPollSupport() = 0;
    virtual VerboseLevel GetVerboseLevel() = 0;
    virtual void SetVerboseLevel(VerboseLevel level) = 0;

    virtual ~IDevice() = 0;
};
```

Return value	Name	Parameters	Description
Status	Open	void* dev PirisDeviceType type VerboseLevel verbose	Opens the connection with the device using pointer with type conversion. Suitable for library implementation or call from another environment.
Status	Open	const PirisDevice& dev VerboseLevel verbose	Opens the connection with the device using class *PirisDevice*, which is created based on the data from search functions.
Status	Close		Closes the connection with the device.
tuple<StatusEx, DataID>	ReadID		Sends **IDN** command, returns object *StatusEx* and *DataID*, includes *name* of the device and *version* of the firmware

`tuple<StatusEx, DataPosition>`	`ReadPosition`		Sends **GP** command, returns object *StatusEx* and *DataPosition*, includes *position*: actual position of focus, zoom and aperture and *ir_filter*: IR filter is on.
`tuple<StatusEx, DataParams>`	`ReadParams`		Sends **GT** command, returns object *StatusEx* and *DataParams*, includes *max_value* of focus, zoom and aperture position, *lens*: type of lens, *ir_present*: IR filter is present and *sens_present*: end stop present.
`tuple<StatusEx, DataState>`	`ReadState`		Sends **GS** command, returns object *StatusEx* and *DataState*, includes *state* of motors (OK/ERR) and *motors_busy*: motors are moving.
`StatusEx`	`DevReset`		Sends **RST** command. Reboot of the device is initiated, which needs at least 10 seconds to be performed.
`StatusEx`	`DevHoming`		Sends **INI** command. Sets focus, zoom to initial position, aperture to fully open and switches IR filter on.

`StatusEx`	`SetAbsolute`	`uint16_t focus`
		`uint16_t zoom`
		`uint16_t iris`
		`bool ir_filter`
			Sends **SETA**:FX;ZX;PX;IX command, which sets focus, zoom and aperture to absolute position specified in the parameters, turns on/off the IR filter.

`StatusEx`	`SetAbsolute`	`const FocusZoomIris<uint16_t>& values` `bool ir_filter`	Sends **SETA**:FX;ZX;PX;IX command, which sets focus, zoom and aperture to absolute position specified in the parameters, turns on/off the IR filter.

`StatusEx`	`SetRelative`	`int16_t focus`
		`int16_t zoom`
		`int16_t iris`
			Sends **SETR**:FX;ZX;PX command, which moves focus, zoom and aperture by a relative value specified in the parameters.

`StatusEx`	`SetRelative`	`const FocusZoomIris<int16_t>& values`	Sends **SETR**:FX;ZX;PX command, which moves focus, zoom and aperture by a relative value specified in the parameters.
`YesNoNA`	`GetFwPollSupport`		Returns availability of cyclic status reading (FW version 1.7.2 and higher).
`VerboseLevel`	`GetVerboseLevel`		Returns current level of printing information to standard output.
`void`	`SetVerboseLevel`	`VerboseLevel level`	Sets level of printing information to standard output.

Tab. 1 – Description of functions of the main driver interface

Device search is done using two static functions, which return vector of corresponding objects – descriptors based on type of main instance. `Pylon::CDeviceInfo` is a descriptor of UART/Ethernet devices, `serial::PortInfo` is a descriptor of RS232 descriptors. Those objects serve to create instance

of class `PirisDevice`, which always describes only one device – product and which serves as a main input object of function *Open(const PirisDevice& dev)*.

```
static int16_t ScanEthernet(std::vector<Pylon::CDeviceInfo>& devices, bool verbose);
static int16_t ScanSerial(std::vector<serial::PortInfo>& devices, bool verbose);
```

Return value	Name	Parameters	Description
int16_t	ScanEthernet	vector<Pylon::CDeviceInfo>& devices bool verbose	Searches for all available UART/Ethernet devices, returns their list in vector *devices* and their count as return value. (-1: error)
int16_t	ScanSerial	std::vector<serial::PortInfo>& devices bool verbose	Searches for all available RS232 devices, returns their list in vector *devices* and their count as return value. (-1: error)

Tab. 2 – Description of static functions for searching

Note: With any firmware prior to **1.5.2** (included) it was impossible to continually read state of the unit during any action (a command was sent, action was performed and upon its completion a reply message was returned – „OK\r\n“). It was therefore not known how long it would take to complete the action. This behaviour is corrected since the release of the next firmware version **1.7.2**. A command is sent, a reply is sent immediately, and the required action starts. When the action being performed, it is advisable to cyclically query the status (GS) and check *motors_busy* state, which is *True* when an action is performed, *False* otherwise.

6 EXAMPLES

Shared folder includes source code of functional examples named *example*, which demonstrate all the features of the driver for both the UART/Ethernet and RS232 unit. **Bin** folder includes corresponding executable files separately for **Linux** OS (x86, x64, ARM, ARM64) and Windows OS (x86, x64). When using **Windows**, it is necessary to have the VC++ Redistributable 2013 and 2017 packages installed (from **install** folder). In case the target machine does not have Pylon 5 runtime installed, it is necessary to install it. Either official installers files from www.baslerweb.com or attached dynamic libraries (separate version depending on OS version and CPU architecture) can be used. When using Windows, dynamic libraries are automatically preferred (attached in the same directory).

OS Windows offers **Pylon GUI** for the UART/Ethernet version. When the demonstration application is started in Ethernet mode, a window with live preview from the camera is opened. If the GUI does not work or is not required, the example can be started with **--no-gui** parameter. Settings of the camera parameters must be done prior to starting the demo or a default configuration file can be used with Basler camera **acA2040-35gmATE.pfs**. If this file within the same directory as the demo, user is prompted to confirm the upload of the configuration to the camera.

When the program starts, the user chooses the type of the unit depending on the communication interface, **Serial** for RS232 and **Ethernet** for UART/Ethernet version.

```
This is ATEsystem P-IRIS Controller driver example.
www.atesystem.cz  produkty@atesystem.cz
================================================

Choose device connection mode:
  0 : Serial
  1 : Ethernet
> 1
```

Mode of demonstration follows. **User input** is a mode, when the user manually sets all the parameters, whereas in **Sequence** mode all the parameters are changed sequentially from zero to max and back – first of all focus, then zoom, aperture, IR filter and eventually all at once.

```
Choose demo mode:
  0 : User input
  1 : Sequence
> 1
```

The demo prompts user to choose one of the connected devices. When using UART/Ethernet version, it is necessary to configure the Basler camera properly, using e.g. Pylon IP Configurator. No other user can be connected to the camera and the subnet of the camera and the PC must be the same.

```
Choose device number:
  0. Basler acA2040-35gmATE#[redacted]
> 0
```

If RS232 version is chosen, any RS232/USB adapter can be used.

```
Choose device number:
  0. COM15   USB Serial Port (COM15)
  1. COM13   Standard Serial over Bluetooth link (COM13)
  2. COM6    Standard Serial over Bluetooth link (COM6)
  3. COM14   Standard Serial over Bluetooth link (COM14)
  4. COM1    Lantronix CPR Port (COM1)
  5. COM7    Standard Serial over Bluetooth link (COM7)
>
```

In case when the UART/Ethernet version is used, the user is prompted to choose the configuration of the camera, if a file with .PFS extension is present in the example folder. Option **Load default** loads the default configuration to the camera from the file, any previous configuration is therefore overwritten. **Use current** uses current configuration.

```
PIRIS open - success (Ethernet camera (camera overiden))
Camera Device Information
=========================
Vendor           : Basler
Model            : acA2040-35gmATE
Firmware version : 107603-01;U;acA2040_35gmATE;V1.0-2;0
=========================
PIRIS set up - success


Choose camera configuration:
  0 : Load default
  1 : Use current
>
```

After all the steps above are finished, communication with the unit is initiated. The following information is read: name of the unit, type of the lens, version of the firmware, current and maximal position of focus, zoom, aperture and IR filter, presence of end stops, indication of movement of the motors, error message of the device and driver. Reinitialization to default position starts after.

```
Camera configuration file acA2040-35gmATE.pfs successfully loaded.

CMD_READ_ID     success:  name=           P_IRIS
                          version=        1.7.2

CMD_READ_POS    success:  focus=          2025
                          zoom=           0
                          iris=           0
                          ir_filter=      1

CMD_READ_TYPE   success:  type=           TL1250P
                          focus=          2025
                          zoom=           800
                          iris=           19
                          ir_present=     1
                          sens_present=   1

CMD_READ_STATE success:   focus=          1
                          zoom=           1
                          iris=           1
                          busy=           0

CMD_HOMING      success

Wait for action finish....
```

In **User Input** mode the application prompts user to cyclically input four values from the ranges shown in pointy brackets – focus, zoom, aperture, IR filter.

```
(enter any string to quit)
The User Input Demo will start now.Press any key to continue . . .

------------------------------------------------------------
Enter Focus <0;2025> Zoom <0;800> Iris <0;19> and IR <0;1> in format: 0 0 0 0:
> 2000 250 3 1
```

In **Sequence** mode the application automatically sets values of focus, zoom, aperture and IR filter in following sequence: focus min. -> focus max. -> focus min., zoom min. -> zoom max. -> zoom min., aperture min. -> aperture max. -> aperture min., IR filter off -> IR filter on -> IR filter off, all min. -> all max.

```
The Demo Sequence will start now. Press any key to continue . . .

------------------------------------------------------------
```

The commands are sent to the unit in both modes in the same way. First, a command to set the lens to absolute values is sent. Then the application waits for the action to be completed (depending on the version of the firmware, as described in the note at the end of chapter 5). Reading of current position for confirmation follows.

```
CMD_SET_ABS     success:  focus=          2000
                          zoom=           250
                          iris=           3
                          ir_filter=      1

Wait for action finish....

CMD_READ_POS    success:  focus=          2000
                          zoom=           250
                          iris=           3
                          ir_filter=      1
```

The application exits automatically in Sequence mode after approximately 45 seconds. In User mode, any string literal which is written and then confirmed ends the application. The unit is restarted before the connection is closed.

```
Enter Focus <0;2025> Zoom <0;800> Iris <0;19> and IR <0;1> in format: 0 0 0 0:
> exit

------------------------ DEMO QUIT ------------------------
CMD_RESET       success


Wait for action finish and then Press any key to continue . . .

PIRIS close - success

Press any key to continue . . .
```