PhotoSound SDK 1.1.2 User Manual

Contents

Introduction 5

Working with the PhotoSoundClassesClassLibrary.dll 7

Getting started in MATLAB 7

Getting Started in Visual Studio C# 13

Controlling Data Acquisition in MATLAB 16

Controlling Data Acquisition in LabVIEW 19

Controlling Data Acquisition in Visual Studio C# 22

Recording data to a File in MATLAB 26

Recording Data to a File in Visual Studio C# 30

ADC AFE5818 setup in MATLAB 34

ADC AFE5818 setup in LabVIEW 37

ADC AFE5818 setup in Visual Studio C# 38

ADC AFE5832 setup in Matlab 41

ADC AFE5832 setup in LabVIEW 45

ADC AFE5832 setup in Visual Studio C# 46

Real-time data processing in MATLAB 47

Real-time data processing in LabVIEW 50

Real-time data processing in Visual C# 52

PhotoSoundClasses.dll Class Library Reference 54

Capture class 54

Configure 54

SamplesToCapture 54

FramesPerPacket 54

DecimationFactor 55

WaitTrigger 55

EnabledAdcMask 55

AutoUpdate 55

Trigger class 56

Configure 56

GetInputFrequencies 57
TriggerOutputs 57
InputNames 57
SlaveDelays 57
GeneratorFrequency 57
ConnectToGenerator 57
InputsDelay 57
InputsGuard 57
EnabledInputsMask 58
InvertedInputsMask 58
AutoUpdate 58
TriggerOutput class 58
Configure 59
PulseWidth 59
Delay 59
SourcesMask 59
ConnectToGenerator 59
Enable 59
Invert 60
AutoUpdate 60
DataLogger class 61
Configure 61
StartLoggingToFile 62
StartLoggingToMemory 62
StopLogging 62
GetFrame 62
OnStartLogging 62
OnStopLogging 63
LimitNumFrames 63
LimitLoggingTime 63
LimitFileSize 63

DataFolder 63
DevicesMask 63
MaxLoggedFrames 63
MaxFileSize 63
LoggingTimeout 64
Logging 64
Progress 64
NumLoggedFrames 64
LoggingTime 64
FileSize 64
AutoUpdate 64
Class AFE5818 65
Configure 65
ConfiguredDevicesMask 65
ConfiguredAdcMask 65
Vca1EqualsVca2 66
AutoUpdate 66
Vca1, Vca2 66
HpfCutoffDivided 67
LowNoiseMode 67
PgaHpfDisabled 67
LnaHpfDisabled 67
PgaClampEnabled 67
F5MHzLpfEnabled 67
TgcAttEnabled 67
PowerMode 67
HpfCutoffFreq 67
LpfCutoffFreq 68
TgcAttenuation 68
LnaGlobalGain 68
PgaGain 68

Class AFE5832 68
Configure 69
ConfiguredDevicesMask 69
ConfiguredAdcMask 69
EnableAttenuatorHpf 69
AttenuatorHpfCorner 69
OddEqualEven 69
AutoUpdate 69
Odd, Even 70
LpfCutoffFreq 70
HpfCutoffFreq 70
DtgcGain 70
EnableLnaHpf 70
LowPowerMode 71
EnableDtgcAttenuator 71
Class AFE5832LP 71
Configure 71
ConfiguredDevicesMask 71
ConfiguredAdcMask 72
HpfCornerFreq 72
LpfCutoffFreq 72
PgaGain 72
LnaGain 72
LowPowerMode 72
LowLatencyEnable 72
Attenuator 73
AutoUpdate 73
Data file format 73

Introduction

The Software Development Kit (SDK) is intended for developing applications for PhotoSound devices in MATLAB, LabVIEW, and Visual Studio C# environments. The package consists of several software layers, an example of which for Legion ADC256 is shown in Figure 1. The first level is the system level and it includes drivers that provide data exchange with devices via the system bus. The second level is intermediate and it consists of 32 or 64-bit libraries (LIB or DLL) that implement the interface for interaction between drivers and top-level software. Components of the first level and, in some cases, the second level are copied to the user's PC during the installation of drivers and are stored separately from other SDK components in the Windows folder. The third level is functional, it is 32 or 64-bit DLLs that ensure the operation of devices – loading firmware, initializing the hardware, implementing the protocol for transferring control commands and data. The fourth level is a library one, this is a .NET assembly, which also has a DLL extension, which contains classes for collecting and saving data, configuring devices, storing configuration settings, saving settings in INI files and reading settings from INI files. .NET assembly of this level can already be used in MATLAB, LabVIEW and Visual Studio C# software environments. The fifth level is applied, it is also represented by a .NET assembly with a DLL extension, which contains graphical controls through which the end user of the application works with devices. These graphics can be placed on a Windows Forms C# application or LabVIEW front panel through a .NET container. The sixth level is custom, it is a standalone application with an EXE extension, which is also a .NET assembly. With this application, the user can not only perform basic operations with the device, but also use it as a dialog box in a more complex application written in any of the above software environments.

Figure 1 SDK Software Levels

The SDK has two sets of files in the sdk\x86 and sdk\x64 folders. The sets differ in the number of machine code of dynamic libraries of the 3rd level. To build applications, only one of the sets is used: x86 - for 32-bit applications and x64 - for 64-bit applications. The components for creating custom applications are located in the PhotoSoundSDK sub-folder. The list and description of files in this subfolder is presented in the table below.

File name/folder	Description
PhotoSoundClasses.dll	Main .NET assembly with a class library for working with devices
PhotoSoundControls.dll	.NET assembly with graphical controls
PhotoSoundDAQ.exe	. NET build with app to perform basic operations with devices
PhotoSoundDAQ.exe.config	Configuration file for PhotoSoundDAQ.exe
PhotoSoundLibs	Folder with 3rd level software components, firmware and INI configuration files with settings
PhotoSoundLibs\Device\PhotoSoundDevice.dll	Functional dynamic library for working with ADC devices
PhotoSoundLibs\Device\PhotoSoundDevice.img	Cypress FX3 USB 3.0 Controller Firmware File
PhotoSoundLibs\Device\AFE5832.ini	Configuration file with the values of the Texas Instruments AFE5832 ADC registers loaded when the software is started for the first time
PhotoSoundLibs\Device\AFE5818.ini	Configuration file with the values of the Texas Instruments AFE5818 ADC registers loaded when the software is started for the first time
PhotoSoundLibs\Device\adc*.bin	FPGA firmware file. The full file name is determined by the device type and revision of its printed circuit board. To read firmware file name for specific ADC board run SetRevision2.1\GetFirmwareName.exe
Config	Config Folder with configuration INI files
Config\Default.ini	Configuration file with default device settings. If the file does not exist, then it is created automatically when the software is started for the first time.
Config*.ini	Alternative configuration files with device settings
Data	Default folder to save data captured from ADC
Maps	Folder for storing files with sensor maps. The map is a column with sensor numbers in order from the first channel of the first ADC to the last channel of the last ADC on the board.
Maps*.map	Sensor map files. For each type of device there is a file with a sensor map.

In addition to the PhotoSoundSDK software components, the SDK includes:

- doc folder - contains this user manual and Excel files - calculators of configuration INI files for ADC: AFE5832.xlsm and AFE5818.xlsm;

- examples folder - contains code examples for MATLAB, LabVIEW and Visual Studio C# software environments.

Working with the PhotoSoundClassesClassLibrary.dll

Getting started in MATLAB

To get started with the class library, you need to load the build using the command **NET.addAssembly(asm_path)**, where *asm_path* is the full path to *PhotoSoundClasses.dll*. Next, you can get a list of classes in the library by calling **disp(asm.Classes)**. The instances of these classes can be used to control PhotoSound devices and collect data, but creation of instances of **DeviceManager** and **Settings** classes is possible. Instances of other classes are created automatically when connected to a device and accessible as properties of an instance of the **DeviceManager** class, then just a device manager. So we create a device manager by command **dev = PhotoSoundClasses.DeviceManager** and start connecting to the device by command **dev.Connect**. Since it takes some time to connect to the device, especially when connecting for the first time after turning on the power of the device, then you can do other tasks, and then go to the cycle of waiting for the connection to complete. You should wait for a connection until one of the **Connected** or **ConnectFailure** properties in the device manager equals 1.

In the process of writing program code, you often need to know the list of methods and properties of a particular class, as well as the events that it can generate. For this, MATLAB has **methods**, **events** и **properties** commands. So, if we execute **methods(dev)**, we get

```
>> methods(dev)

Methods for class PhotoSoundClasses.DeviceManager:

Connect          Disconnect       GetPlotData
CreateLogger     Equals           GetType
DeviceManager    GetHashCode      ToString
```

Equals, GetType, GetHashCode, and **ToString** methods are standard for all NET classes. A description of the rest of the methods, properties and events can be found in the tables at the end of the section. In addition, if you type . and press the Tab key, a list of methods and properties will appear from which you can select the desired one.

During the process of connecting to the device and when working with it, various errors may occur, for example, if the device's power is not turned on or the cable is not connected. The **OnError** event is provided to notify the user of errors in Device Manager. By subscribing to this event using the **addlistener(dev,'OnError',@onerror)** command, you can display an error message if it occurs. The **onerror** handler function here takes two arguments: the first is the source of the event (always the device manager), and the second is a reference to an instance of the **MessageEventArgs** class. The **MessageEventArgs** class has a **Message** property that contains

a description of the error, and a **Source** property — a reference to an instance of the class that is the source of the error. An example of such a function code is presented below:

```
function onerror(~,event)
    disp([char(event.Source.ToString) ' error: ' char(event.Message)]);
end
```

The next step after successfully connecting the ADC to the device, as a rule, is to display the ADC data on a graph. To do this, the Device Manager has a **GetPlotData** method. The command **num_samples = dev.GetPlotData(buffer, buffer_length, device_id, adc_num, chan_num)** will copy the **chan_num** channel data samples for the **adc_num** ADC and **device_id** device to the data **buffer** in the buffer memory. Arguments to this method are numbered from zero. The method returns the number of samples **num_samples** copied to the buffer. It can be less than the number of samples requested or the length of **buffer_length** if data collection is performed for fewer samples. To allocate memory for the data buffer in the Matlab environment there is the **NET.createArray** command. The size of the buffer can be selected based on the maximum number of data samples that can be obtained from one ADC channel. To find out this number, just read the value of the **MaxSamplesToCapture** property of the Device Manager. If **num_samples** is 0, then there is no data yet. The **GetPlotData** method is intended only for data visualization in order to control data collection. To process ADC data in real time or to write this data to a file, the **DataLogger** class is intended.

When you finish working with the device, you should disconnect from the device. To do this, run the **dev.Disconnect** command. When connected to a device, the device settings are automatically loaded from the configuration INI file and configured. And when disabled, the settings are automatically saved in the configuration file.

The table below shows the *simple.m* script code from *examples\matlab* folder, which consists of the above commands.

Table 1: Sample MATLAB script for connecting to the device, collecting, and visualizing data on the graph

```
Filename = mfilename('fullpath');
app_path = fileparts(filename);
asm_path = fullfile(app_path,'..\..\PhotoSoundSDK\PhotoSoundClasses.dll');
asm = NET.addAssembly(asm_path);
disp(asm.Classes);
dev = PhotoSoundClasses.DeviceManager;
methods(dev);
properties(dev);
events(dev);

disp('Connecting...');
addlistener(dev,'OnError',@onerror);
dev.Connect;
while ~dev.Connected && ~dev.ConnectFailure
    pause(0.1);
end

if dev.Connected
    disp('Successfully connected to device');
    data = NET.createArray('System.Int16',dev.MaxSamplesToCapture);

    adc = 0;
    chan = 0;

    fig = figure('Name','Plot data example');
    while isvalid(fig)
        samples = dev.GetPlotData(data,data.Length,0,adc,chan);
        if samples > 0
            tmp = int16(data);
            plot(tmp(1:samples));
        end
        pause(0.1);
    end
else
    disp('Failed to connect to device');
end

dev.Disconnect;
disp('Disconnected');
```

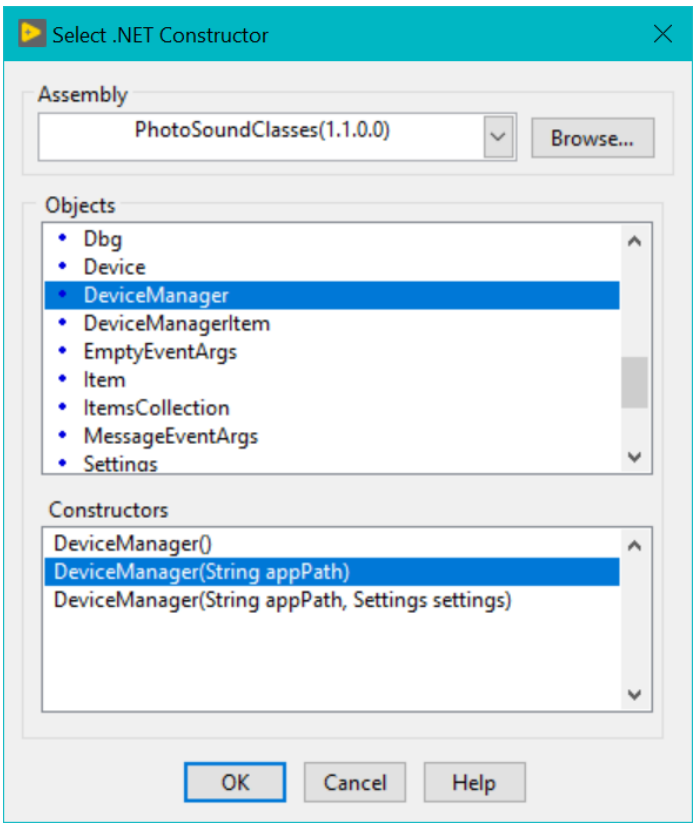

Figure 1 Creating a Device Manager in LabVIEW

You can start working with a class library in LabVIEW right away by creating an instance of the **DeviceManager** class, then just a device manager. To do this, right-click (RMB) on the diagram, select **Connectivity\.NET\Constructor Node** in the Functions window. Next, left-click on the diagram and in the dialog box that appears, make an overview using the **Browse...** button and find PhotoSoundClasses.dll. After that, a list of **Objects** classes and **Constructors** appears in the window from which we select **DeviceManager** and **DeviceManager(String appPath)** (Figure 1). Many other classes can be seen in the list of **Objects** classes. The user can use instances of these classes to control PhotoSound devices and collect data, but he can only create instances of the **DeviceManager** and **Settings** classes. Instances of the rest of the classes are created automatically when connected to a device and are available as properties of the device manager. After placing the device manager constructor on the diagram, we launch the connection to the device. To do this, add **Connectivity\.NET\Invoke Node (.NET)** to the diagram using the RMB, connect the link input to the constructor and select the **Connect(Boolean autoApdate)** method through the **Method** menu (Figure 2).

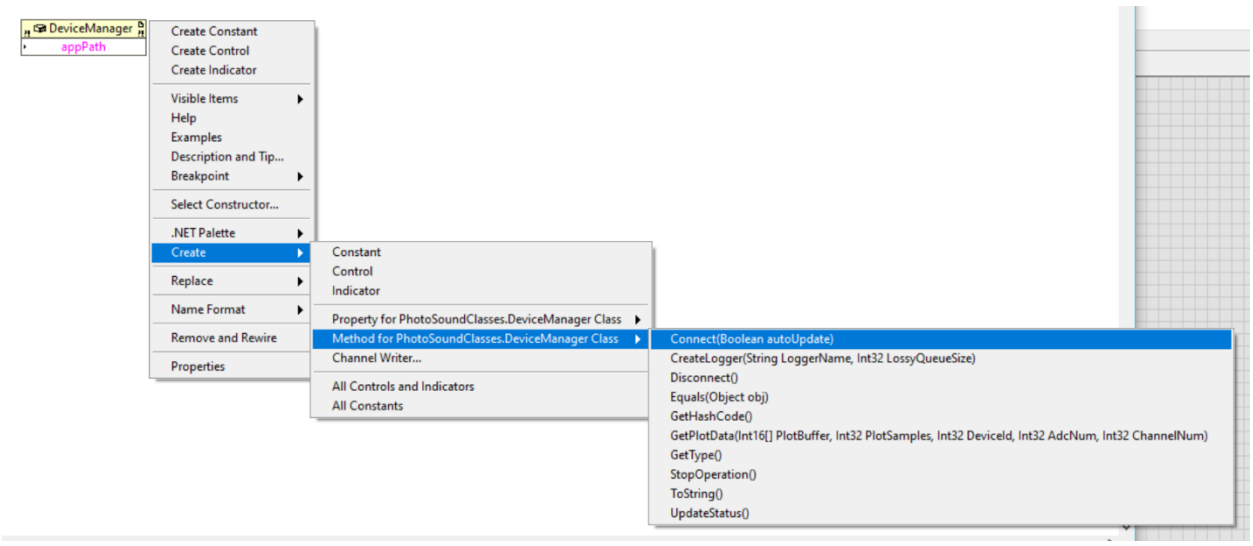

Figure 2 Calling Connect Method in LabVIEW

Since it takes some time to connect to the device, especially when connecting for the first time after turning on the power of the device, then you can do other tasks, and then go to the cycle of waiting for the connection to complete. You should wait for a connection until the value of one of the **Connected** or **ConnectFailure** properties of the device manager becomes True. To read the properties of the device manager, add **Connectivity\.NET\Property Node (.NET)** to the diagram using the right mouse button, connect the link input to the constructor and through the **Property** menu select a specific property, for example **Connected** (Figure 3):

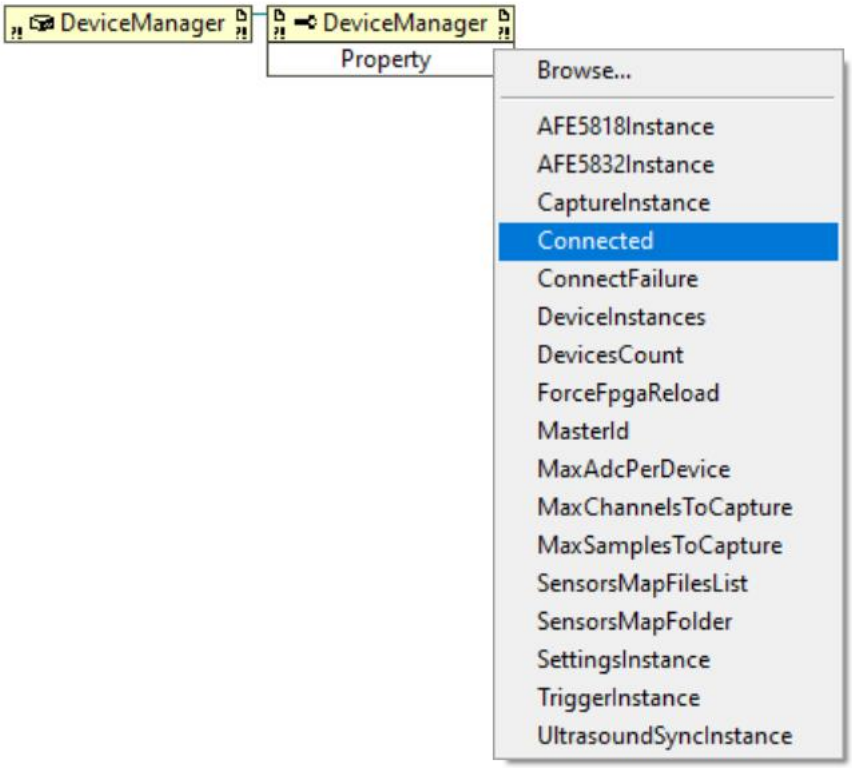

Figure 3: Reading Device Manager Property in LabVIEW

During the process of connecting to the device and when working with it, various errors may occur, for example, if the device's power is not turned on or the cable is not connected. The **OnError** event is provided to notify the user of errors in Device Manager. If you register a handler for this event, you can display an error message if it occurs. To do this, add the **Connectivity\.NET\Register Event Callback** to the diagram using the RMB, connect the **Event** input to the device manager constructor and select **OnError** from the **Event** menu (Figure 4). Now, through RMB at the **VI Ref** input, select **Create Callback VI**. LabVIEW will create a new Vi with the desired interface, on the diagram of which you can add the output of a dialog box with an error message (Figure 4). The handler diagram has an **Event Data** input - a cluster with two fields: **sender** – an event source (always a device manager), **e** – a reference to an instance of the **MessageEventArgs** class. The **MessageEventArgs** class has a **Message** property that contains a description of the error, and a **Source** property is a reference to the instance of the class that is the source of the error.

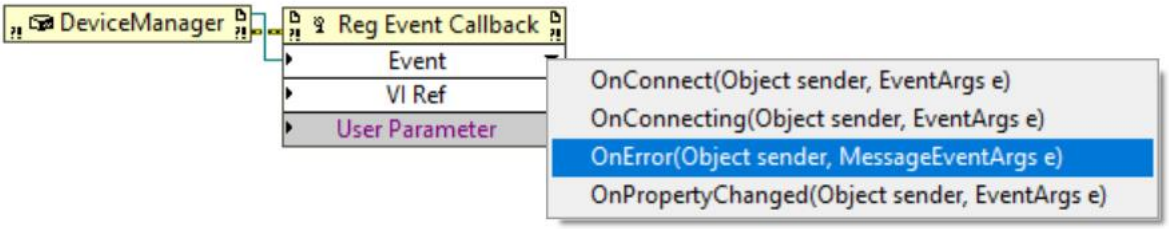

Figure 4: Create an OnError event handler in LabVIEW

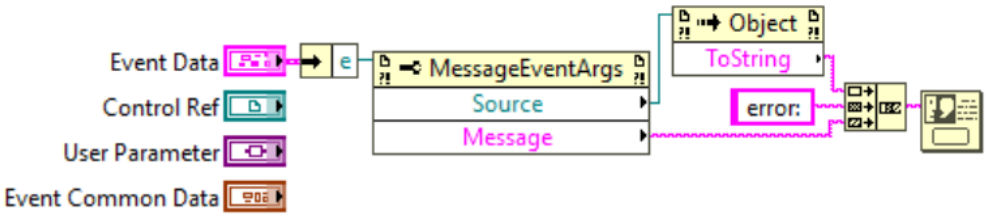

Figure 5: OnError Event Handler chart

After successfully connecting the ADC to the device, the next step is usually to display the ADC data on a graph. To do this, the Device Manager has a **GetPlotData** method that copies the data samples of the **ChannelNum** channel for the ADC **AdcNum** and the **DeviceId()** device to the data buffer in the **PlotBuffer** memory. Arguments to this method are numbered from zero. The buffer size can be selected based on the maximum number of data samples that can be obtained from one ADC channel. To find out this number, just read the value of the **MaxSamplesToCapture** property of the Device Manager. At the output of **GetPlotData**, the method returns the number of samples copied to the buffer. It can be less than the number of samples requested or the length of the **PlotSamples** buffer if data collection is done for fewer samples. If the output of **GetPlotData** is 0, then there is no data yet. The **GetPlotData** method is intended only for data visualization in order to control data collection. The **DataLogger** class is intended to process ADC data in real time or to write this data to a file in a user-formatted format.

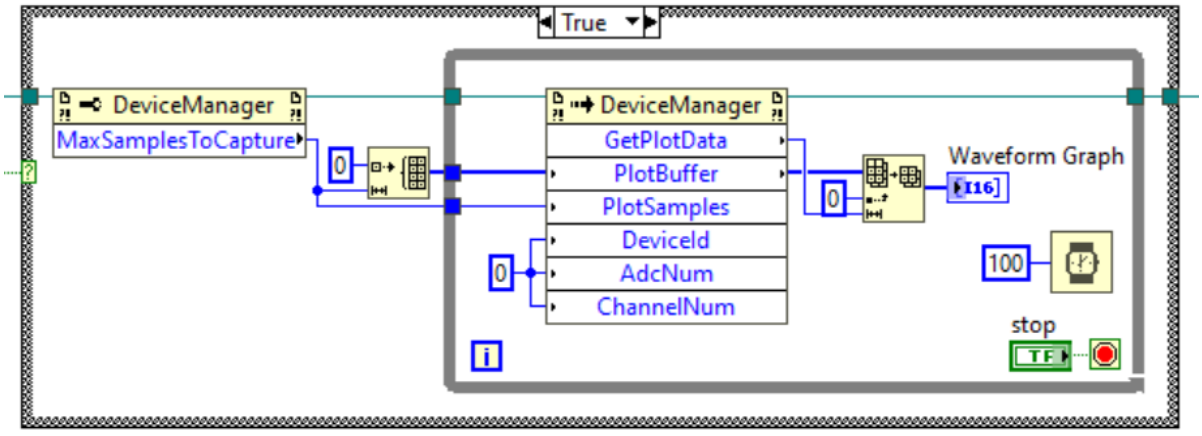

Figure 6: Displaying ADC data on a graph

When you finish working with the device, you should disconnect from the device. The **Disconnect** method is intended for this. When connected to a device, the device settings are

automatically loaded from the configuration INI file and configured. And when disabled, the settings are automatically saved in the configuration file.

The examples\labview\ folder contains a simple.vi example that implements the above steps.

Getting Started in Visual Studio C#

To get started with the class library, you need to add a reference to the PhotoSoundClasses.dll library in your Visual Studio project. Next, select the target build platform for the project – x64 or x86 and copy the files from the x64\PhotoSoundSDK\ or x86\ PhotoSoundSDK\ folder to the project's output folder. The class library contains many classes, instances of which the user can use to control PhotoSound devices and collect data, but he himself can only create instances of the **DeviceManager** and **Settings** classes. Instances of other classes are created automatically when connected to a device and are available as properties of an instance of the **DeviceManager** class, then just a device manager. So, we create a device manager and start the connection to the device:

```
DeviceManager deviceManager = new DeviceManager();
deviceManager.Connect();
```

Immediately after calling the **Connect** method, the device manager will contain empty references to instances of other classes, and properties with information about devices will be incorrect, since it takes some time to connect to the device, especially when connecting for the first time after the device is powered on. To notify the user about the end of the connection, the **OnConnect** event is provided in the device manager. In the handler for this event, you can perform actions that require information about devices or access instances of other classes in the library. An example of such a handler that starts a timer to update the graph and allocates memory for the data buffer is below:

```
deviceManager.OnConnect += OnConnectEventHandler;
private void OnConnectEventHandler(object sender, EventArgs e)
{
    PlotBuffer = new short[deviceManager.MaxSamplesToCapture];
    timer1.Start();
}
```

During the process of connecting to the device and when working with it, various errors may occur, for example, if the device's power is not turned on or the cable is not connected. The **OnError** event is provided to notify the user of errors in Device Manager. By subscribing to this event, you can display an error message if it occurs. The handler function must have two arguments: the first is a reference to the device manager, and the second is a reference to an instance of the **MessageEventArgs** class. The **MessageEventArgs** class has a **Message** property that contains a description of the error, and a **Source** property – a reference to an instance of the class that is the source:

```
deviceManager.OnError += OnErrorEventHandler;
private void OnErrorEventHandler(object sender, MessageEventArgs e)
```

```
{
    MessageBox.Show($"{e.Source.ToString()} error: {e.Message}");
}
```

After successfully connecting the ADC to the device, the next step is usually to display the ADC data on a graph. To do this, there is a method in the device manager

```
public int GetPlotData(ref short[] PlotBuffer, int PlotSamples,
    int DeviceId, int AdcNum, int ChannelNum);
```

The method returns the number of data samples copied to the **PlotBuffer** buffer. The requested number of **PlotSamples** must be less than or equal to the size of the **PlotBuffer**. **DeviceId**, **AdcNum** and **ChannelNum** – device identifier on the bus, ADC number and ADC channel number, respectively. These arguments are numbered from zero. The buffer size can be selected based on the maximum number of data samples that can be obtained from one ADC channel. To find out this number, just read the value of the **MaxSamplesToCapture** property of the Device Manager. If the return value of **GetPlotData** is 0, then there is no data yet. The **GetPlotData** method is intended only for data visualization in order to control data collection. The **DataLogger** class is intended to process ADC data in real time or to write this data to a file.

When you finish working with the device, you should disconnect from the device. The **Disconnect** method of the Device Manager is intended for this. When connected to a device, the device settings are automatically loaded from the configuration INI file and configured. And when disabled, the settings are automatically saved in the configuration file.

The table below shows the code from the sample Simple project from the examples\visual\SdkExamples\ folder that implements the above actions.

Table 2: An example Visual C# program for connecting to a device, collecting and visualizing data on a graph

```
using PhotoSoundClasses;
using System;
using System.Windows.Forms;

namespace Simple
{
    public partial class Simple : Form
    {
        public Simple()
        {
            InitializeComponent();
            chart1.Series.Clear();
            var series = chart1.Series.Add("ADC1/CH1");
            series.ChartType =
System.Windows.Forms.DataVisualization.Charting.SeriesChartType.FastLine;
        }

        private DeviceManager deviceManager = null;
        private short[] PlotBuffer = null;

        private void Form1_Load(object sender, EventArgs e)
```

```
        {
            deviceManager = new DeviceManager();
            deviceManager.OnConnect += OnConnectEventHandler;
            deviceManager.OnError += OnErrorEventHandler;
            deviceManager.Connect();
        }

        private void timer1_Tick(object sender, EventArgs e)
        {
            int samples = deviceManager.GetPlotData(PlotBuffer, PlotBuffer.Length, 0,
0, 0);
            if (samples > 0)
            {
                chart1.Series[0].Points.Clear();
                chart1.Series[0].Points.DataBindY(new ArraySegment<short>(PlotBuffer,
0, samples));
            }
        }

        private void OnConnectEventHandler(object sender, EventArgs e)
        {
            PlotBuffer = new short[deviceManager.MaxSamplesToCapture];
            timer1.Start();
        }

        private void OnErrorEventHandler(object sender, MessageEventArgs e)
        {
            MessageBox.Show($"{e.Source.ToString()} error: {e.Message}");
        }

        private void Form1_FormClosed(object sender, FormClosedEventArgs e)
        {
            deviceManager?.Disconnect();
        }
    }
}
```

Controlling Data Acquisition in MATLAB

ADC data acquisition is controlled by the **Capture**, **Trigger**, and **TriggerOutput** classes. Instances of these classes are not created by the user, but by the device manager after successfully connecting to devices. Links to created instances of classes are stored in the properties of the same name **Capture**, **Trigger** and **TriggerOutput** of the device manager (the **DeviceManager** class). Before connecting devices, these properties contain empty links (null).

The **Capture** class allows you to change data acquisition settings, such as the number of data samples per ADC channel or the flag to wait for a trigger event before starting data collection. Settings from the properties of the **Capture** class are passed to all connected devices simultaneously. The **Trigger** class defines the condition by which data collection begins, for example, whether triggering from an internal generator is allowed or the number of an input that receives an external trigger signal. Settings from the properties of the **Trigger** class are transferred to only one device, which is the master. If several masters are connected to the PC, the settings are only transferred to the first device on the system bus. The **TriggerOutput** class defines the parameters for the trigger output, such as pulse width and delay. The settings from the properties of the **TriggerOutput** class are also transferred only to the first master.

To change any parameter, you just need to assign a new value to the corresponding property of the class instance, for example **dev.Capture.WaitTrigger = true**. This value will be automatically transferred to the device via the system bus, for example USB, and also saved in memory for later writing the settings to the configuration file. This behavior is well suited for management through a graphical user interface - the user clicks a button and the settings change immediately. There is another method that is suitable for programmatically controlling data collection when many parameters are changed at the same time. In order to prohibit the automatic transfer of settings to the device, you need to assign the value **false** to the **AutoUpdate** property of the corresponding class. Next, you can assign new values to the properties of the class and call the **Configure** method of that class. The **Configure** method passes the settings to the device and sets the **AutoUpdate** property back to **true**.

An example of changing the properties of the **Capture** class:

```
dev.Capture.AutoUpdate = false;
dev.Capture.DecimationFactor = 1;
dev.Capture.EnabledAdcMask = 2^dev.MaxAdcPerDevice-1;
dev.Capture.FramesPerPacket = 1;
dev.Capture.SamplesToCapture = 1000;
dev.Capture.WaitTrigger = 1;
dev.Capture.Configure;
```

An example of changing the properties of the **Trigger** class:

```
dev.Trigger.AutoUpdate = false;
dev.Trigger.ConnectToGenerator = true;
dev.Trigger.InvertedInputsMask = 0;
dev.Trigger.EnabledInputsMask = 1;
dev.Trigger.GeneratorFrequency = 15;
```

```
dev.Trigger.InputNames(1) = 'OPT';
dev.Trigger.SlaveDelays(1) = 0;
dev.Trigger.InputsDelay = 3;
dev.Trigger.InputsGuard = 10;
dev.Trigger.Configure;
```

An example of changing the properties of the **TriggerOutput** class:

```
dev.Trigger.TriggerOutputs(1).AutoUpdate = false;
dev.Trigger.TriggerOutputs(1).ConnectToGenerator = true;
dev.Trigger.TriggerOutputs(1).PulseWidth = 10;
dev.Trigger.TriggerOutputs(1).SourcesMask = 0;
dev.Trigger.TriggerOutputs(1).Invert = false;
dev.Trigger.TriggerOutputs(1).Enable = true;
dev.Trigger.TriggerOutputs(1).Delay = 1;
dev.Trigger.TriggerOutputs(1).Configure;
```

Each class property has a certain range of valid values. When you assign a value to a property, it is validated and the property is changed only if the new value is in that range. Therefore, when creating a graphical user interface, you should read the property immediately after assignment and update the corresponding control with the read value. So, the user will be able to see that the value entered by him is incorrect and it was not saved and was not transferred to the device.

In addition to properties with settings, the **Trigger** class contains the **GetInputFrequencies** method. This method reads the current values of the frequency meters connected to the trigger inputs. After the call, you must wait for the **OnUpdateInputFrequencies** event, and then you can read the frequency values from the **Trigger . InputFrequencies** array. The handler function has two arguments, the first one is sent to the device manager, and the second is an empty one. An example of such a function:

```
function onupdatefreq(src,~)
    for n = 1:src.Trigger.InputFrequencies.Length
        disp(['Trigger input ' num2str(n) ' frequency is '
              num2str(src.Trigger.InputFrequencies(n))]);
    end
end
```

The table below shows the code of the captrig.m script from the examples\matlab\ folder, which implements the data collection control described above. And in the reference section of this tutorial, you can find a description of all the properties and methods of the **Capture, Trigger,** and **TriggerOutput** classes.

Table 3: Example of the MATLAB Script to Control Data Collection

```
filename = mfilename('fullpath');
app_path = fileparts(filename);
asm_path = fullfile(app_path,'..\..\x64\PhotoSoundClasses.dll');
asm = NET.addAssembly(asm_path);
dev = PhotoSoundClasses.DeviceManager;

disp('Connecting...');
```

```
    addlistener(dev,'OnError',@onerror);
    addlistener(dev,'OnUpdateInputFrequencies',@onupdatefreq);
    dev.Connect;
    while ~dev.Connected && ~dev.ConnectFailure
        pause(0.1);
    end

    if dev.Connected
        disp('Successfully connected to device');

        dev.Capture.AutoUpdate = false;
        dev.Capture.DecimationFactor = 1;
        dev.Capture.EnabledAdcMask = 2^dev.MaxAdcPerDevice-1;
        dev.Capture.FramesPerPacket = 1;
        dev.Capture.SamplesToCapture = 1000;
        dev.Capture.WaitTrigger = 1;
        dev.Capture.Configure;

        dev.Trigger.AutoUpdate = false;
        dev.Trigger.ConnectToGenerator = true;
        dev.Trigger.InvertedInputsMask = 0;
        dev.Trigger.EnabledInputsMask = 1;
        dev.Trigger.GeneratorFrequency = 15;
        dev.Trigger.InputNames(1) = 'OPT';
        dev.Trigger.SlaveDelays(1) = 0;
        dev.Trigger.InputsDelay = 0;
        dev.Trigger.InputsGuard = 10;
        dev.Trigger.Configure;

        dev.Trigger.TriggerOutputs(1).AutoUpdate = false;
        dev.Trigger.TriggerOutputs(1).ConnectToGenerator = true;
        dev.Trigger.TriggerOutputs(1).PulseWidth = 10;
        dev.Trigger.TriggerOutputs(1).SourcesMask = 0;
        dev.Trigger.TriggerOutputs(1).Invert = false;
        dev.Trigger.TriggerOutputs(1).Enable = true;
        dev.Trigger.TriggerOutputs(1).Delay = 1;
        dev.Trigger.TriggerOutputs(1).Configure;

        dev.Trigger.UpdateInputFrequencies;

        data = NET.createArray('System.Int16',dev.MaxSamplesToCapture);
        adc = 0;
        chan = 0;

        fig = figure('Name','Plot data example');
        while isvalid(fig)
            samples = dev.GetPlotData(data,data.Length,0,adc,chan);
            if samples > 0
                tmp = int16(data);
                plot(tmp(1:samples));
            end
            pause(0.1);
        end
    else
        disp('Failed to connect to device');
    end

    dev.Disconnect;

disp('Disconnected');
```

Controlling Data Acquisition in LabVIEW

ADC data acquisition is controlled by the **Capture**, **Trigger**, and **TriggerOutput** classes. Instances of these classes are not created by the user, but by the device manager after successfully connecting to devices. Links to created instances of classes are stored in the properties of the same name **Capture**, **Trigger** and **TriggerOutput** of the device manager (the **DeviceManager** class). Before connecting devices, these properties contain empty links (null).

The **Capture** class allows you to change data acquisition settings, such as the number of data samples per ADC channel or the flag to wait for a trigger event before starting data collection. Settings from the properties of the **Capture** class are passed to all connected devices simultaneously. The **Trigger** class defines the condition by which data collection begins, for example, whether triggering from an internal generator is allowed or the number of an input that receives an external trigger signal. Settings from the properties of the **Trigger** class are transferred to only one device, which is the master. If several masters are connected to the PC, the settings are only transferred to the first device on the system bus. The **TriggerOutput** class defines the parameters for the trigger output, such as pulse width and delay. The settings from the properties of the **TriggerOutput** class are also transferred only to the first master.

To change any parameter, you just need to assign a new value to the corresponding property of the class instance, as shown in the figures below. This value will be automatically transferred to the device via the system bus, for example USB, and also saved in memory for later writing the settings to the configuration file. In LabVIEW, instead of making your own value change handler for each control, you can update multiple properties in a common handler. Since the user can change the value of only one control at a time, there will be only one new value in the handler. An internal check in the class will reveal this new value and the settings will be transferred to the device via the system bus once.

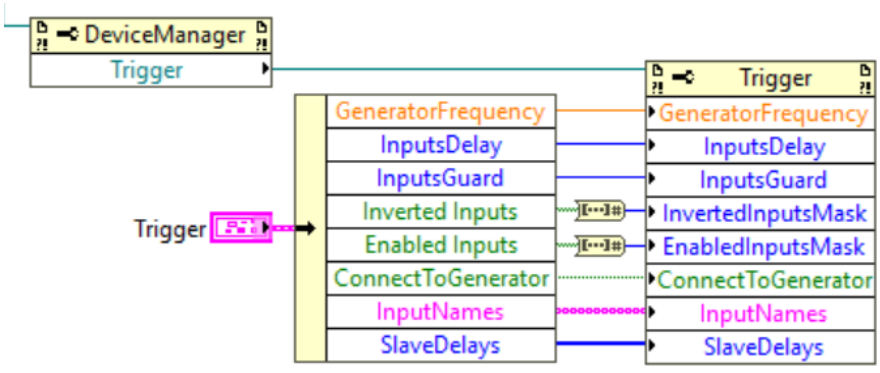

Figure 2: Changing properties of the Trigger class in LabVIEW

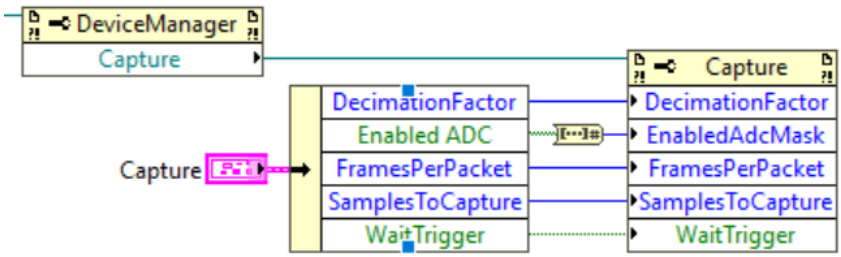

Figure 3 Changing Properties of the Capture Class in LabVIEW

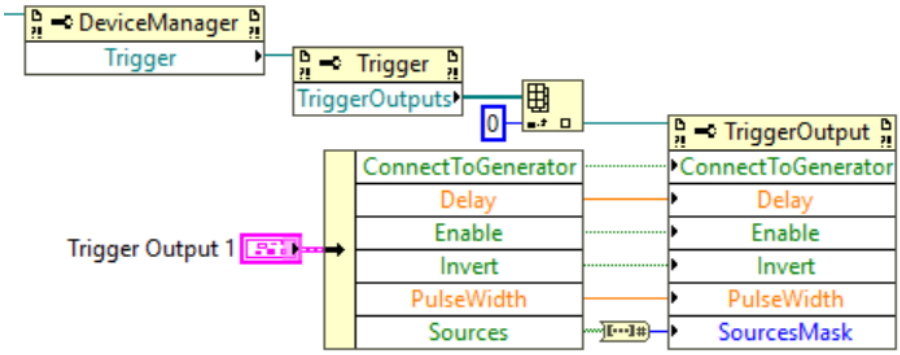

Figure 4 Changing Properties of the TriggerOutput Class in LabVIEW

Each class property has a certain range of valid values. When you assign a value to a property, it is validated and the property is changed only if the new value is in that range. Therefore, when creating a graphical user interface, you should read the property immediately after assignment and update the corresponding control with the read value. So, the user will be able to see that the value entered by him is incorrect and it was not saved and was not transferred to the device. The figures below show how you can read new property values for all three classes.

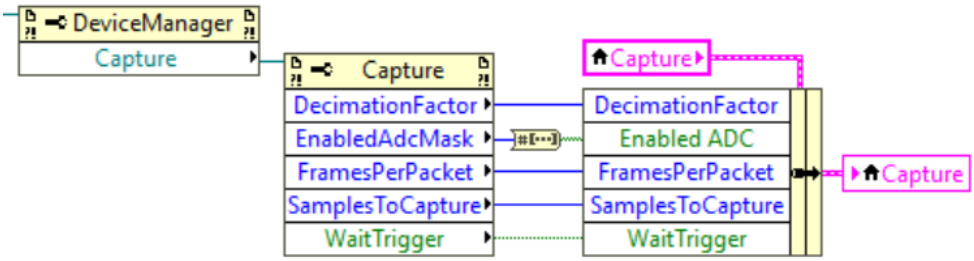

Figure 5: Reading Properties of the Capture Class in LabVIEW

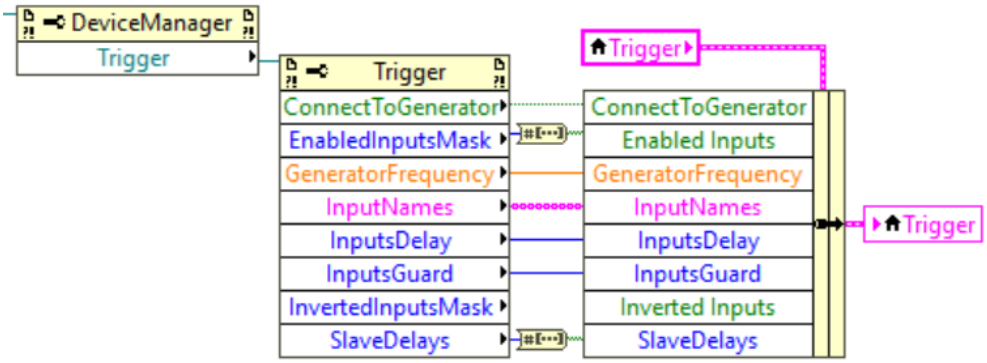

Figure 6: Reading Properties of the Trigger Class in LabVIEW

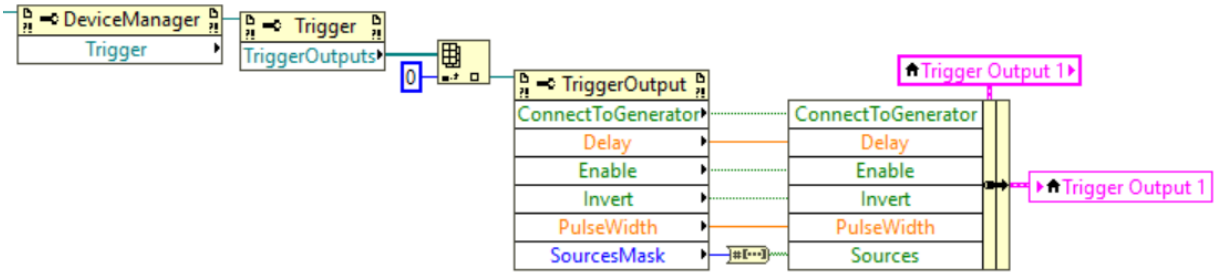

Figure 7: Reading the Properties of the TriggerOutput Class in LabVIEW

In addition to properties with settings, the **Trigger** class contains the **UpdateInputFrequencies** method. This method reads the current values of the counters connected to the trigger inputs. After calling the method, you must wait for the **OnUpdateInputFrequencies** event, and then you can read the frequency values from the **Trigger . InputFrequencies** array. The handler function has two arguments, the first is a reference to the device manager and the second is a null reference. An example of such a function

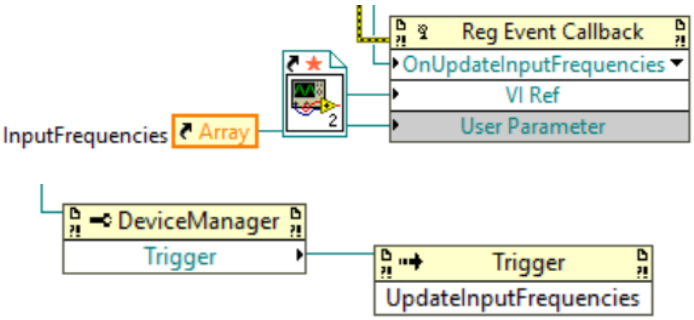

Figure 8: Measuring Trigger Input Frequencies in LabVIEW

The examples\labview\ folder contains an example captrig.vi that implements the data collection control described above. And in the reference section of this tutorial, you can find a description of all the properties and methods of the **Capture**, **Trigger**, and **TriggerOutput** classes.

Controlling Data Acquisition in Visual Studio C#

ADC data acquisition is controlled by the **Capture**, **Trigger**, and **TriggerOutput** classes. Instances of these classes are not created by the user, but by the device manager after successfully connecting to devices. Links to created instances of classes are stored in the properties of the same name **Capture**, **Trigger** and **TriggerOutput** of the device manager (the **DeviceManager** class). Before connecting devices, these properties contain empty links (null).

The **Capture** class allows you to change data acquisition settings, such as the number of data samples per ADC channel or the flag to wait for a trigger event before starting data collection. Settings from the properties of the **Capture** class are passed to all connected devices simultaneously. The **Trigger** class defines the condition by which data collection begins, for example, whether triggering from an internal generator is allowed or the number of an input that receives an external trigger signal. Settings from the properties of the **Trigger** class are transferred to only one device, which is the master. If several masters are connected to the PC, the settings are only transferred to the first device on the system bus. The **TriggerOutput** class defines the parameters for the trigger output, such as pulse width and delay. The settings from the properties of the **TriggerOutput** class are also transferred only to the first master.

To change any parameter, you just need to assign a new value to the corresponding property of the class instance, for example, **deviceManager.Capture.WaitTrigger = true**. This value will be automatically transferred to the device via the system bus, for example USB, and also saved in memory for later writing the settings to the configuration file. This behavior is well suited for management through a graphical user interface - the user clicks a button and the settings change immediately. There is another method that is suitable for programmatically controlling data collection when many parameters are changed at the same time. In order to prohibit the automatic transfer of settings to the device, you need to assign the value **false** to the **AutoUpdate** property of the corresponding class. Next, you can assign new values to the properties of the class and call the **Configure** method of that class. The **Configure** method passes the settings to the device and sets the **AutoUpdate** property back to **true**.

An example of changing the properties of the **Capture** class:

```
deviceManager.Capture.AutoUpdate = false;
deviceManager.Capture.DecimationFactor = 1;
deviceManager.Capture.EnabledAdcMask = (1u << deviceManager.MaxAdcPerDevice) - 1;
deviceManager.Capture.FramesPerPacket = 1;
deviceManager.Capture.SamplesToCapture = 1000;
deviceManager.Capture.WaitTrigger = true;
deviceManager.Capture.Configure();
```

An example of changing the properties of the **Trigger** class:

```
deviceManager.Trigger.AutoUpdate = false;
deviceManager.Trigger.ConnectToGenerator = true;
deviceManager.Trigger.InvertedInputsMask = 0;
deviceManager.Trigger.EnabledInputsMask = 0;
deviceManager.Trigger.GeneratorFrequency = 10.0;
```

```
deviceManager.Trigger.InputNames[0] = "OPT";
deviceManager.Trigger.SlaveDelays[0] = 0;
deviceManager.Trigger.InputsDelay = 0;
deviceManager.Trigger.InputsGuard = 10;
deviceManager.Trigger.Configure();
```

An example of changing the properties of the **TriggerOutput** class:

```
deviceManager.Trigger.TriggerOutputs[0].AutoUpdate = false;
deviceManager.Trigger.TriggerOutputs[0].ConnectToGenerator = true;
deviceManager.Trigger.TriggerOutputs[0].PulseWidth = 10.0;
deviceManager.Trigger.TriggerOutputs[0].SourcesMask = 0;
deviceManager.Trigger.TriggerOutputs[0].Invert = false;
deviceManager.Trigger.TriggerOutputs[0].Enable = true;
deviceManager.Trigger.TriggerOutputs[0].Delay = 0;
deviceManager.Trigger.TriggerOutputs[0].Configure();
```

Each class property has a certain range of valid values. When you assign a value to a property, it is validated and the property is changed only if the new value is in that range. Therefore, when creating a graphical user interface, you should read the property immediately after assignment and update the corresponding control with the read value. So, the user will be able to see that the value entered by him is incorrect and it was not saved and was not transferred to the device.

In addition to properties with settings, the **Trigger** class contains the **UpdateInputFrequencies** method. This method reads the current values of the counters connected to the trigger inputs. After calling the method, you must wait for the **OnUpdateInputFrequencies** event, and then you can read the frequency values from the **Trigger . InputFrequencies** array. The handler function has two arguments, the first is a reference to the device manager and the second is a null reference. An example of such a function:

```
private void OnUpdateInputFrequencies(object sender, EventArgs e)
{
   if (labels == null)
      labels = new Label[4] { labelFreq1, labelFreq2, labelFreq3, labelFreq4 };

      for (int i = 0; i < deviceManager.Trigger.InputFrequencies.Length; i++)
         labels[i].Text = $"Input {i} frequency:
{deviceManager.Trigger.InputFrequencies[i]:F1} Hz";
}
```

The table below shows the code from the CapTrig example project from the examples\visual\SdkExamples\ folder that implements the data collection control described above. And in the reference section of this tutorial, you can find a description of all the properties and methods of the **Capture**, **Trigger**, and **TriggerOutput** classes.

Table 4: Sample Visual C# Program for Managing Data Collection

```
using PhotoSoundClasses;
using System;
using System.Windows.Forms;

namespace CapTrig
```

```
    {
        public partial class CapTrig : Form
        {
            public CapTrig()
            {
                InitializeComponent();
                chart1.Series.Clear();
                var series = chart1.Series.Add("ADC1/CH1");
                series.ChartType =
System.Windows.Forms.DataVisualization.Charting.SeriesChartType.FastLine;
                this.Enabled = false;
            }

            private DeviceManager deviceManager = null;
            private short[] PlotBuffer = null;

            private void Form1_Load(object sender, EventArgs e)
            {
                deviceManager = new DeviceManager();
                deviceManager.OnUpdateInputFrequencies += OnUpdateInputFrequencies;
                deviceManager.OnConnect += OnConnectEventHandler;
                deviceManager.OnError += OnErrorEventHandler;
                deviceManager.Connect();
            }

            private void OnUpdateInputFrequencies(object sender, EventArgs e)
            {
                if (labels == null)
                    labels = new Label[4] { labelFreq1, labelFreq2, labelFreq3,
labelFreq4 };

                for (int i = 0; i < deviceManager.Trigger.InputFrequencies.Length;
i++)
                    labels[i].Text = $"Input {i} frequency:
{deviceManager.Trigger.InputFrequencies[i]:F1} Hz";
            }

            private void timer1_Tick(object sender, EventArgs e)
            {
                int samples = deviceManager.GetPlotData(PlotBuffer, PlotBuffer.Length,
0, 0, 0);
                if (samples > 0)
                {
                    chart1.Series[0].Points.Clear();
                    chart1.Series[0].Points.DataBindY(new
ArraySegment<short>(PlotBuffer, 0, samples));
                }
            }

            private void OnConnectEventHandler(object sender, EventArgs e)
            {
                PlotBuffer = new short[deviceManager.MaxSamplesToCapture];
                timer1.Start();
                this.Enabled = true;

                deviceManager.Capture.AutoUpdate = false;
                deviceManager.Capture.DecimationFactor = 1;
                deviceManager.Capture.EnabledAdcMask = (1u <<
deviceManager.MaxAdcPerDevice) - 1;
                deviceManager.Capture.FramesPerPacket = 1;
                deviceManager.Capture.SamplesToCapture = 1000;
                deviceManager.Capture.WaitTrigger = true;
                deviceManager.Capture.Configure();
```

```
                deviceManager.Trigger.AutoUpdate = false;
                deviceManager.Trigger.ConnectToGenerator = true;
                deviceManager.Trigger.InvertedInputsMask = 0;
                deviceManager.Trigger.EnabledInputsMask = 0;
                deviceManager.Trigger.GeneratorFrequency = 10.0;
                deviceManager.Trigger.InputNames[0] = "OPT";
                deviceManager.Trigger.SlaveDelays[0] = 0;
                deviceManager.Trigger.InputsDelay = 0;
                deviceManager.Trigger.InputsGuard = 10;
                deviceManager.Trigger.Configure();

                deviceManager.Trigger.TriggerOutputs[0].AutoUpdate = false;
                deviceManager.Trigger.TriggerOutputs[0].ConnectToGenerator = true;
                deviceManager.Trigger.TriggerOutputs[0].PulseWidth = 10.0;
                deviceManager.Trigger.TriggerOutputs[0].SourcesMask = 0;
                deviceManager.Trigger.TriggerOutputs[0].Invert = false;
                deviceManager.Trigger.TriggerOutputs[0].Enable = true;
                deviceManager.Trigger.TriggerOutputs[0].Delay = 0;
                deviceManager.Trigger.TriggerOutputs[0].Configure();

                udGeneratorFrequency.Value =
(decimal)deviceManager.Trigger.GeneratorFrequency;
                udSamplesToCapture.Value = deviceManager.Capture.SamplesToCapture;
                cbWaitForTrigger.Checked = true;
            }

            private void OnErrorEventHandler(object sender, MessageEventArgs e)
            {
                MessageBox.Show($"{e.Source.ToString()} error: {e.Message}");
            }

            private void Form1_FormClosed(object sender, FormClosedEventArgs e)
            {
                deviceManager.Disconnect();
            }

            private void udSamplesToCapture_ValueChanged(object sender, EventArgs e)
            {
                deviceManager.Capture.SamplesToCapture =
(int)udSamplesToCapture.Value;
                udSamplesToCapture.Value = deviceManager.Capture.SamplesToCapture;
            }

            private void udGeneratorFrequency_ValueChanged(object sender, EventArgs e)
            {
                deviceManager.Trigger.GeneratorFrequency =
(double)udGeneratorFrequency.Value;
                udGeneratorFrequency.Value =
(decimal)deviceManager.Trigger.GeneratorFrequency;
            }

            private void cbWaitForTrigger_CheckedChanged(object sender, EventArgs e)
            {
                deviceManager.Capture.WaitTrigger = cbWaitForTrigger.Checked;
            }

            private Label[] labels = null;

            private void buttonUpdate_Click(object sender, EventArgs e)
            {
                deviceManager.Trigger.UpdateInputFrequencies();
            }
```

```
        }
    }
```

Recording data to a File in MATLAB

Data is written to a file using the **DataLogger** class. Instances of this class (data loggers) are created by the user using the **CreateLogger** method of the device manager (the **DeviceManager** class):

```
logger = dev.CreateLogger('Matlab');
```

Call the **CreateLogger** method only after connecting to devices, otherwise the method returns an empty reference. The method returns a reference to the created data logger, and its arguments are the name of the created data logger and the length of the queue when writing to memory, which is discussed in the section. The name is used to save the settings of the logger in the configuration file. The user can create an arbitrary number of data loggers. Each logger can write data from one or several devices to binary files with the raw extension, and several loggers can receive data from the same device.

The logger starts writing ADC data to a file after calling its **StartLoggingToFile** method with a file name without an extension as an argument. The path to the file being written is determined by the **DataFolder** property of the logger:

```
logger.DataFolder = app_path;
logger.StartLoggingToFile('TestData');
```

Immediately after the successful start of the recording, the logger sets the value of the **Logging** property to **true**, and after the end of the recording – to **false**. The end of writing to the file occurs when the **StopLogging** method of the logger is called:

```
logger.StopLogging;
```

The logger can automatically end recording to the file if one of the restrictive conditions set before the start of recording is met. These conditions include exceeding the file size in megabytes, exceeding the file recording time, and exceeding the number of recorded frames. An example of setting these conditions through the properties of the logger is presented below:

```
logger.MaxFileSize = 100;
logger.MaxLoggedFrames = 100;
logger.LoggingTimeout = 60;
logger.LimitLoggingTime = true;
logger.LimitNumFrames = true;
logger.LimitFileSize = true;
```

Data recording to a file can be controlled using the properties-states of the logger: **Progress** – recording progress in percent, **FileSize** – the current size of the data file in megabytes,

NumLoggedFrames – the current number of recorded ADC data frames and **LoggingTime** – the current time from the beginning of the file recording in seconds. The Progress property shows the actual progress of the recording only if one of the restrictive conditions **MaxFileSize** or **MaxLoggedFrames** is specified, and the progress refers to the one closest to the fulfillment of the condition. The logging time **LoggingTime** is not used to calculate the logging progress, since the time control is intended only for an emergency stop of logging to a file as a result of some unforeseen situation, for example, due to non-receipt of data when the optical trigger signal is turned off. Below is an example of displaying the current state of the logger:

```
k = fprintf('Logging: %d%%, %6.2f MB, %d frames, %6.2f s',...
      logger.Progress,logger.FileSize,logger.NumLoggedFrames,...
      logger.LoggingTime);
```

The table below shows the filesave.m script code from the examples\matlab\ folder, which implements the data collection control described above. Also in the examples\matlab\RawConverter\ folder there is a Raw2Mat.m script for converting a RAW file to MAT format. In the reference section of this tutorial, you can find a description of all the properties and methods of the **DataLogger** class, as well as a description of the data file format.

Table 5: An example MATLAB script to write ADC data to a file

```
filename = mfilename('fullpath');
app_path = fileparts(filename);
asm_path = fullfile(app_path,'..\..\PhotoSoundSDK\PhotoSoundClasses.dll');
asm = NET.addAssembly(asm_path);
dev = PhotoSoundClasses.DeviceManager;

disp('Connecting...');
addlistener(dev,'OnError',@onerror);
dev.Connect;
while ~dev.Connected && ~dev.ConnectFailure
    pause(0.1);
end

if dev.Connected
    disp('Successfully connected to device');
    data = NET.createArray('System.Int16',dev.MaxSamplesToCapture);

    k = 0;
    adc = 0;
    chan = 0;
    fig = figure('Name','Plot data example');

    logger = dev.CreateLogger('Matlab');
    logger.DataFolder = app_path;
    logger.DevicesMask = 2^dev.DevicesCount-1;
    logger.MaxFileSize = 100;
    logger.MaxLoggedFrames = 100;
    logger.LoggingTimeout = 60;
    logger.LimitLoggingTime = true;
    logger.LimitNumFrames = true;
    logger.LimitFileSize = true;

    logger.StartLoggingToFile('TestData');
```

```
    logging = true;

    while isvalid(fig)
        samples = dev.GetPlotData(data,data.Length,0,adc,chan);
        if samples > 0
            tmp = int16(data);
            plot(tmp(1:samples));
        end
        for m=1:k
            fprintf('\b');
        end
        k = 0;
        if logger.Logging
            k = fprintf('Logging: %d%%, %6.2f MB, %d frames, %6.2f s',...
                logger.Progress,logger.FileSize,logger.NumLoggedFrames,...
                logger.LoggingTime);
        elseif logging
            logging = false;
            fprintf('Logging was finished\n');
        end
        pause(0.1);
    end

    logger.StopLogging;
else
    disp('Failed to connect to device');
end

dev.Disconnect;
fprintf('\nDisconnected\n');
```

Recording Data to a File in LabVIEW

Data is written to a file using the **DataLogger** class. Instances of this class (data loggers) are created by the user using the **CreateLogger** method of the **DeviceManager** (Figure 9). Call the **CreateLogger** method only after connecting to devices, otherwise the method returns an empty reference. The method returns a reference to the created data logger, and its arguments are the name of the created data logger and the length of the queue when writing to memory, which is discussed the DataLogger class below. The name is used to save the settings of the logger in the configuration file. The user can create an arbitrary number of data loggers. Each logger can write data from one or several devices to binary files with the RAW extension, and several loggers can receive data from the same device.

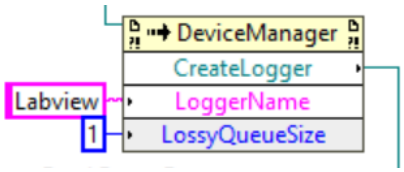

Figure 9 Creating a Data Logger in LabVIEW

The logger starts writing ADC data to a file after calling its **StartLoggingToFile** method with a file name without an extension as an argument. The path to the file being written is determined by the **DataFolder** property of the logger. The end of writing to the file occurs when the **StopLogging** method of the logger is called (Figure 10). Immediately after the successful start of

the recording, the logger sets the value of the **Logging** property to true, and after the end of the recording – to **false.**

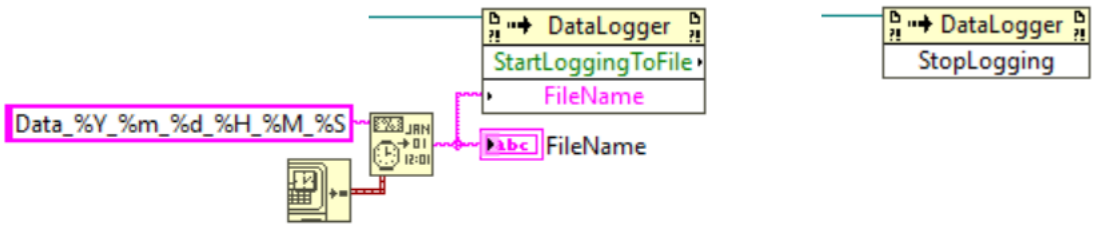

Figure 10: Starting and Stopping Writing Data to a File in LabVIEW

The logger can automatically end recording to the file if one of the restrictive conditions set before the start of recording is met. These conditions include exceeding the file size in megabytes, exceeding the file recording time, and exceeding the number of recorded frames. An example of setting these conditions through the properties of the registrar is presented below:

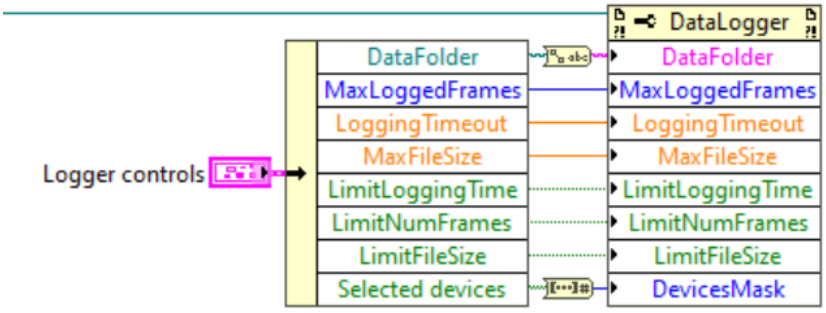

Figure 11: Configuring the Logger to Stop Conditional Recording in LabVIEW

Data recording to a file can be controlled using the properties-states of the logger: **Progress** – recording progress in percent, **FileSize** – the current size of the data file in megabytes, **NumLoggedFrames** – the current number of recorded ADC data frames and **LoggingTime** – the current time from the beginning of the file recording in seconds. The **Progress** property shows the actual progress of the recording only if one of the restrictive conditions **MaxFileSize** or **MaxLoggedFrames** is specified, and the progress refers to the one closest to the fulfillment of the condition. The recording time **LoggingTime** is not used to calculate the recording progress, since the time control is intended only for the emergency stop of recording to the file as a result of some unforeseen situation, for example, due to non-receipt of data when the optical trigger signal is turned off.

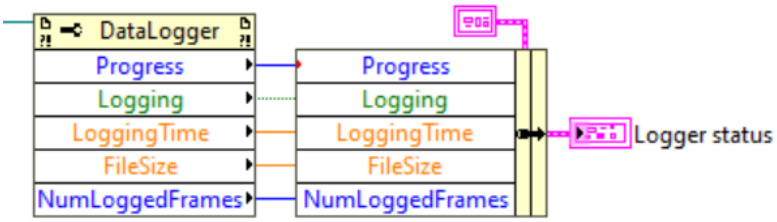

Figure 12: Checking the Status of Writing to a File in LabVIEW

The examples\labview\ folder contains an example filesave.vi that implements the above writing of ADC data to a file. Also in this folder is an example fileplay.vi, which reads data from a data file and displays it on a graph. In the reference section of this tutorial, you can find a description of all the properties and methods of the **DataLogger** class, as well as a description of the data file format.

Recording Data to a File in Visual Studio C#

Data is written to a file using the **DataLogger** class. Instances of this class (data loggers) are created by the user using the **CreateLogger** method of the device manager (the **DeviceManager** class):

```
logger = deviceManager.CreateLogger("FileSave");
```

Call the **CreateLogger** method only after connecting to devices, otherwise the method returns an empty reference. The method returns a reference to the created data logger, and its arguments are the name of the created data logger and the length of the queue when writing to memory, which is discussed in the section. The name is used to save the settings of the logger in the configuration file. The user can create an arbitrary number of data loggers. Each logger can write data from one or several devices to binary files with the RAW extension, and several loggers can receive data from the same device.

The logger starts writing ADC data to a file after calling its **StartLoggingToFile** method with a file name without extension as an argument:

```
logger.StartLoggingToFile("Data " + DateTime.Now.ToString("yyyy-MM-dd HH-mm-ss"));
```

The path to the file being written is determined by the **DataFolder** property of the logger. The end of writing to the file occurs when the **StopLogging** method of the logger is called:

```
logger.StopLogging();
```

When recording starts and stops, the logger generates the **OnStartLogging** and **OnStopLogging** events, respectively. Events have a standard signature and can be used to change the locking of control buttons.

The logger can automatically end recording to the file if one of the restrictive conditions set before the start of recording is met. These conditions include exceeding the file size in

megabytes, exceeding the file recording time, and exceeding the number of recorded frames. Below is an example of assigning these properties using the controls of the main window:

```
logger.LoggingTimeout = (double)udLoggingTimeout.Value;
logger.MaxLoggedFrames = (int)udNumLoggedFrames.Value;
logger.MaxFileSize = (double)udMaxFileSize.Value;
logger.LimitLoggingTime = cbLimitLoggingTime.Checked;
logger.LimitNumFrames = cbLimitNumFrames.Checked;
logger.LimitFileSize = cbLimitFileSize.Checked;
```

Data recording to a file can be controlled using the properties-states of the logger: **Progress** – recording progress in percent, **FileSize** – the current size of the data file in megabytes, **NumLoggedFrames** – the current number of recorded ADC data frames and **LoggingTime** – the current time from the beginning of the file recording in seconds. The **Progress** property shows the actual progress of the recording only if one of the restrictive conditions **MaxFileSize** or **MaxLoggedFrames** is specified, and the progress refers to the one closest to the fulfillment of the condition. The recording time **LoggingTime** is not used to calculate the recording progress, since the time control is intended only for the emergency stop of recording to the file as a result of some unforeseen situation, for example, due to non-receipt of data when the optical trigger signal is turned off. Below is the display of the status of the logger in the main window:

```
labelLoggedFrames.Text = $"Logged frames: {logger.NumLoggedFrames}";
labelLoggingTime.Text = $"Logging time: {logger.LoggingTime:F2} s";
labelFileSize.Text = $"File size: {logger.FileSize:F2} MB";
labelProgress.Text = $"Progress: {logger.Progress}%";
```

The table below shows the code from a sample FileSave project from the examples\visual\SdkExamples\ folder that implements the above writing data to a file. Also, in this folder is the FilePlay project, which reads data from the data file and displays it on the chart. In the reference section of this tutorial, you can find a description of all the properties and methods of the **DataLogger** class, as well as a description of the data file format.

The code in the table below shows an example of using the Device Manager **OnPropertyChanged** event. This event is triggered when one of the properties of the object that is the source of the event changes. In this case, you are only interested in changes after loading property values from the configuration file. The handler for this event has arguments: **object sender** (always a reference to the device manager) and a reference e to an instance of the **EmptyEventArgs** class, which has a **Source** property - the event source, in this case, **Source** should contain a reference to the user-created datalogger. Below is an example of how to initialize the controls of the main window with the values read from the configuration file after creating the data logger:

```
private void OnPropertyChangedEventHandler(object sender, EmptyEventArgs e)
{
   if (e.Source == logger)
   {
       udLoggingTimeout.Value = (decimal)logger.LoggingTimeout;
       udMaxFileSize.Value = (decimal)logger.MaxFileSize;
```

```
            udNumLoggedFrames.Value = logger.MaxLoggedFrames;
            cbLimitLoggingTime.Checked = logger.LimitLoggingTime;
            cbLimitNumFrames.Checked = logger.LimitNumFrames;
            cbLimitFileSize.Checked = logger.LimitFileSize;
            labelFolder.Text = logger.DataFolder;
        }
    }
```

Table 6: Visual C# programs to write data to a file

```
using PhotoSoundClasses;
using System;
using System.Windows.Forms;

namespace FileSave
{
  public partial class FileSave : Form
  {
    public FileSave()
    {
      InitializeComponent();
      chart1.Series.Clear();
      var series = chart1.Series.Add("ADC1/CH1");
      series.ChartType =
System.Windows.Forms.DataVisualization.Charting.SeriesChartType.FastLine;
      this.Enabled = false;
    }

    private DeviceManager deviceManager = null;
    private short[] PlotBuffer = null;

    private void FileSave_Load(object sender, EventArgs e)
    {
      deviceManager = new DeviceManager();
      deviceManager.OnConnect += OnConnectEventHandler;
      deviceManager.OnError += OnErrorEventHandler;
      deviceManager.Connect();
    }

    private void timer1_Tick(object sender, EventArgs e)
    {
      int samples = deviceManager.GetPlotData(PlotBuffer, PlotBuffer.Length, 0, 0, 0);
      if (samples > 0)
      {
        chart1.Series[0].Points.Clear();
        chart1.Series[0].Points.DataBindY(new ArraySegment<short>(PlotBuffer, 0,
samples));
      }

      labelLoggedFrames.Text = $"Logged frames: {logger.NumLoggedFrames}";
      labelLoggingTime.Text = $"Logging time: {logger.LoggingTime:F2} s";
      labelFileSize.Text = $"File size: {logger.FileSize:F2} MB";
      labelProgress.Text = $"Progress: {logger.Progress}%";
    }

    private DataLogger logger = null;

    private void OnConnectEventHandler(object sender, EventArgs e)
    {
      PlotBuffer = new short[deviceManager.MaxSamplesToCapture];
      logger = deviceManager.CreateLogger("FileSave");
      deviceManager.OnPropertyChanged += OnPropertyChangedEventHandler;
```

```
      timer1.Start();
      this.Enabled = true;
    }

    private void OnPropertyChangedEventHandler(object sender, EmptyEventArgs e)
    {
      if (e.Source == logger)
      {
        udLoggingTimeout.Value = (decimal)logger.LoggingTimeout;
        udMaxFileSize.Value = (decimal)logger.MaxFileSize;
        udNumLoggedFrames.Value = logger.MaxLoggedFrames;
        cbLimitLoggingTime.Checked = logger.LimitLoggingTime;
        cbLimitNumFrames.Checked = logger.LimitNumFrames;
        cbLimitFileSize.Checked = logger.LimitFileSize;
        labelFolder.Text = logger.DataFolder;
      }
    }

    private void OnErrorEventHandler(object sender, MessageEventArgs e)
    {
      MessageBox.Show($"{e.Source.ToString()} error: {e.Message}");
    }

    private void FileSave_FormClosed(object sender, FormClosedEventArgs e)
    {
      deviceManager.Disconnect();
    }

    private void buttonBrowse_Click(object sender, EventArgs e)
    {
      folderBrowserDialog1.SelectedPath = logger.DataFolder;
      if (folderBrowserDialog1.ShowDialog() == DialogResult.OK)
        logger.DataFolder = folderBrowserDialog1.SelectedPath;
    }

    private void buttonStart_Click(object sender, EventArgs e)
    {
      logger.StartLoggingToFile("Data " + DateTime.Now.ToString("yyyy-MM-dd HH-mm-
ss"));
    }

    private void buttonStop_Click(object sender, EventArgs e)
    {
      logger.StopLogging();
    }

    private void udMaxFileSize_ValueChanged(object sender, EventArgs e)
    {
      logger.MaxFileSize = (double)udMaxFileSize.Value;
      udMaxFileSize.Value = (decimal)logger.MaxFileSize;
    }

    private void udNumLoggedFrames_ValueChanged(object sender, EventArgs e)
    {
      logger.MaxLoggedFrames = (int)udNumLoggedFrames.Value;
      udNumLoggedFrames.Value = logger.MaxLoggedFrames;
    }

    private void udLoggingTimeout_ValueChanged(object sender, EventArgs e)
    {
      logger.LoggingTimeout = (double)udLoggingTimeout.Value;
      udLoggingTimeout.Value = (decimal)logger.LoggingTimeout;
    }
```

```
    private void cbLimitNumFrames_CheckedChanged(object sender, EventArgs e)
    {
      logger.LimitNumFrames = cbLimitNumFrames.Checked;
    }

    private void cbLimitLoggingTime_CheckedChanged(object sender, EventArgs e)
    {
      logger.LimitLoggingTime = cbLimitLoggingTime.Checked;
    }

    private void cbLimitFileSize_CheckedChanged(object sender, EventArgs e)
    {
      logger.LimitFileSize = cbLimitFileSize.Checked;
    }
  }
}
```

ADC AFE5818 setup in MATLAB

The **AFE5818** ADC is configured using the **AFE5818** and **AFE5818Vca** classes. Instances of these classes are not created by the user, but by the device manager after successfully connecting to devices. The link to the created instances is stored in the **AFE5818** property of the same name of the device manager (the DeviceManager class). Before connecting devices, this property contains an empty reference (null).

To change any parameter, you just need to assign a new value to the corresponding property of the class instance, for example **dev.AFE5818.LowNoiseMode = true**. This value will be automatically transferred to the device via the system bus, for example USB, and also saved in memory for later writing the settings to the configuration file. Some parameters are represented by enumerated (**enum**) properties, for example, the **PowerMode** property of the **AFE5818Vca** class. Only certain values can be assigned to such properties, which can be viewed using the MATLAB **enumeration** command. An example of using this command is below:

```
>> enumeration(dev.AFE5818.Vca1.PowerMode)

Enumeration members for class 'PhotoSoundClasses.AFE5818Vca+PowerModes':

    LowNoise
    LowPower
    MediumPower
```

In order to assign any value to the enum property in MATLAB, you must first obtain a list of values in a variable, and then use this variable with the index of the desired value. An example of obtaining such lists for the enum property of the **AFE5818Vca** class is presented below:

```
PowerMode = System.Enum.GetValues(dev.AFE5818.Vca1.PowerMode.GetType);
HpfCutoffFreq =
System.Enum.GetValues(dev.AFE5818.Vca1.HpfCutoffFreq.GetType);
```

```
LpfCutoffFreq =
System.Enum.GetValues(dev.AFE5818.Vca1.LpfCutoffFreq.GetType);
TgcAttenuation =
System.Enum.GetValues(dev.AFE5818.Vca1.TgcAttenuation.GetType);
LnaGlobalGain =
System.Enum.GetValues(dev.AFE5818.Vca1.LnaGlobalGain.GetType);
PgaGain = System.Enum.GetValues(dev.AFE5818.Vca1.PgaGain.GetType);
```

Below is an example of setting up the **AFE5818** ADC. In order not to transmit data to the device every time one parameter is changed, first you need to assign the value **false** to the **AutoUpdate** property, and after changing all the parameters, you need to call the **Configure** method of the **AFE5818** class.

```
dev.AFE5818.AutoUpdate = false;
dev.AFE5818.ConfiguredAdcMask = 2^dev.MaxAdcPerDevice-1;
dev.AFE5818.ConfiguredDevicesMask = 2^dev.DevicesCount-1;
dev.AFE5818.Vca1EqualsVca2 = true;
dev.AFE5818.Vca1.HpfCutoffDivided = false;
dev.AFE5818.Vca1.LowNoiseMode = true;
dev.AFE5818.Vca1.PgaHpfDisabled = false;
dev.AFE5818.Vca1.LnaHpfDisabled = false;
dev.AFE5818.Vca1.PgaClampEnabled = true;
dev.AFE5818.Vca1.F5MHzLpfEnabled = true;
dev.AFE5818.Vca1.TgcAttEnabled = true;
dev.AFE5818.Vca1.PowerMode = PowerMode(2);
dev.AFE5818.Vca1.HpfCutoffFreq = HpfCutoffFreq(2);
dev.AFE5818.Vca1.LpfCutoffFreq = LpfCutoffFreq(2);
dev.AFE5818.Vca1.TgcAttenuation = TgcAttenuation(2);
dev.AFE5818.Vca1.LnaGlobalGain = LnaGlobalGain(2);
dev.AFE5818.Vca1.PgaGain = PgaGain(2);
dev.AFE5818.Configure;
```

Only the basic parameters of the ADC can be changed using the properties of the **AFE5818** and **AFE5818Vca** classes. All other parameters can be edited in the AFE5818.xlsm file located in the doc SDK folder. After finishing editing, you need to click the "Create ini file" button on the "Result" page, and copy the generated AFE5818.ini to the PhotoSoundLibs\Device\ folder. When the device is turned on for the first time, a new file will be loaded into the device and the ADC will work with the new parameters.

The table below shows the script code afe5818.m from the examples\matlab\ folder, which implements the ADC setup described above. And in the reference section of this manual, you can find a description of all the properties and methods of the **AFE5818** and **AFE5818Vca** classes.

Table 7: Example of the AFE5818 ADC configuration in MATLAB

```
filename = mfilename('fullpath');
app_path = fileparts(filename);
asm_path = fullfile(app_path,'..\..\x64\PhotoSoundClasses.dll');
asm = NET.addAssembly(asm_path);
dev = PhotoSoundClasses.DeviceManager;

disp('Connecting...');
addlistener(dev,'OnError',@onerror);
dev.Connect;
```

```
while ~dev.Connected && ~dev.ConnectFailure
    pause(0.1);
end

if dev.Connected
    disp('Successfully connected to device');

    PowerMode = System.Enum.GetValues(dev.AFE5818.Vca1.PowerMode.GetType);
    HpfCutoffFreq =
System.Enum.GetValues(dev.AFE5818.Vca1.HpfCutoffFreq.GetType);
    LpfCutoffFreq =
System.Enum.GetValues(dev.AFE5818.Vca1.LpfCutoffFreq.GetType);
    TgcAttenuation =
System.Enum.GetValues(dev.AFE5818.Vca1.TgcAttenuation.GetType);
    LnaGlobalGain =
System.Enum.GetValues(dev.AFE5818.Vca1.LnaGlobalGain.GetType);
    PgaGain = System.Enum.GetValues(dev.AFE5818.Vca1.PgaGain.GetType);

    dev.AFE5818.AutoUpdate = false;
    dev.AFE5818.ConfiguredAdcMask = 2^dev.MaxAdcPerDevice-1;
    dev.AFE5818.ConfiguredDevicesMask = 2^dev.DevicesCount-1;
    dev.AFE5818.Vca1EqualsVca2 = true;
    dev.AFE5818.Vca1.HpfCutoffDivided = false;
    dev.AFE5818.Vca1.LowNoiseMode = true;
    dev.AFE5818.Vca1.PgaHpfDisabled = false;
    dev.AFE5818.Vca1.LnaHpfDisabled = false;
    dev.AFE5818.Vca1.PgaClampEnabled = true;
    dev.AFE5818.Vca1.F5MHzLpfEnabled = true;
    dev.AFE5818.Vca1.TgcAttEnabled = true;
    dev.AFE5818.Vca1.PowerMode = PowerMode(2);
    dev.AFE5818.Vca1.HpfCutoffFreq = HpfCutoffFreq(2);
    dev.AFE5818.Vca1.LpfCutoffFreq = LpfCutoffFreq(2);
    dev.AFE5818.Vca1.TgcAttenuation = TgcAttenuation(2);
    dev.AFE5818.Vca1.LnaGlobalGain = LnaGlobalGain(2);
    dev.AFE5818.Vca1.PgaGain = PgaGain(2);
    dev.AFE5818.Configure;

    data = NET.createArray('System.Int16',dev.MaxSamplesToCapture);
    adc = 0;
    chan = 0;

    fig = figure('Name','Plot data example');
    while isvalid(fig)
        samples = dev.GetPlotData(data,data.Length,0,adc,chan);
        if samples > 0
            tmp = int16(data);
            plot(tmp(1:samples));
        end
        pause(0.1);
    end
else
    disp('Failed to connect to device');
end

dev.Disconnect;
disp('Disconnected');
```

ADC AFE5818 setup in LabVIEW

The **AFE5818** ADC is configured using the **AFE5818** and **AFE5818Vca** classes. Instances of these classes are not created by the user, but by the device manager after successfully connecting to devices. The link to the created instances is stored in the **AFE5818** property of the same name of the device manager (the DeviceManager class). Before connecting devices, this property contains an empty reference (null).

To change any parameter, you just need to assign a new value to the corresponding property of the class instance, as shown in the figures below. This value will be automatically transferred to the device via the system bus, for example USB, and also saved in memory for later writing the settings to the configuration file. In LabVIEW, instead of making your own value change handler for each control, you can update multiple properties in a common handler. Since the user can change the value of only one control at a time, there will be only one new value in the handler. An internal check in the class will reveal this new value and the settings will be transferred to the device via the system bus once.

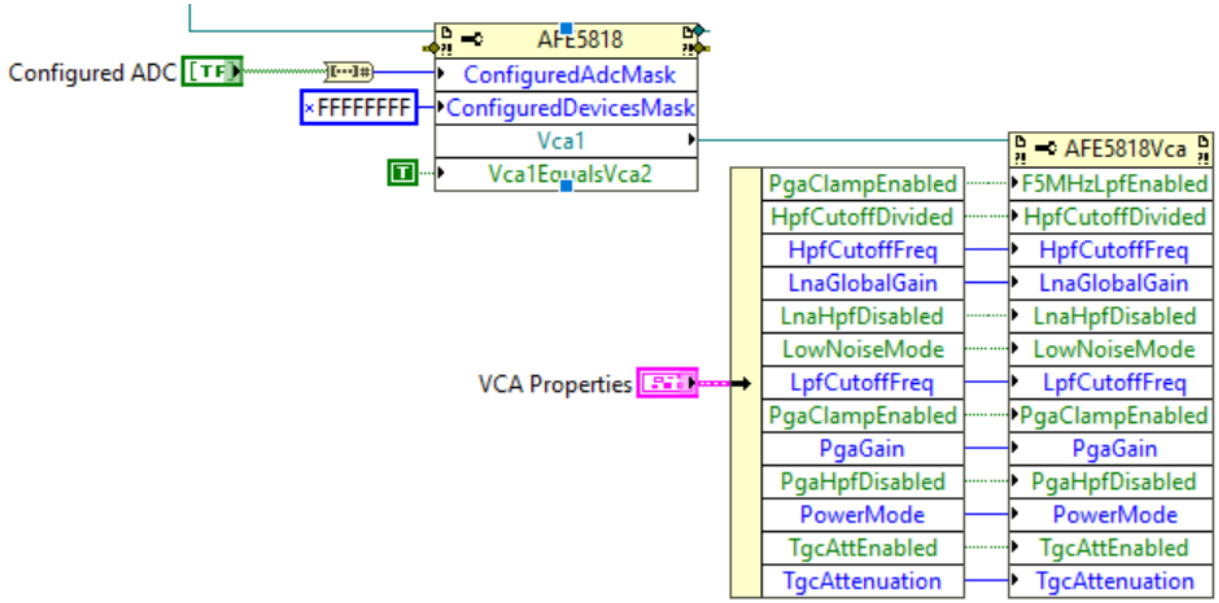

Figure 13 Changing the Properties of the AFE6818 and AFE5818Vca Classes in LabVIEW

Each class property has a certain range of valid values. When you assign a value to a property, it is validated and the property is changed only if the new value is in that range. Therefore, when creating a graphical user interface, you should read the property immediately after assignment and update the corresponding control with the read value. So, the user will be able to see that the value entered by him is incorrect and it was not saved and was not transferred to the device. The pictures below show how new property values can be read.

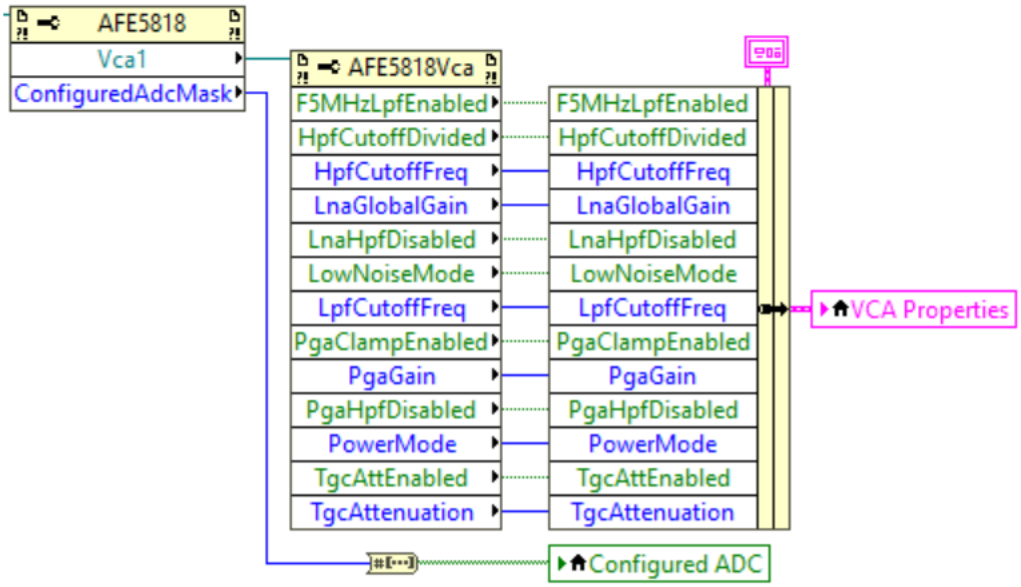

Figure 14. Reading Properties of the AFE5818 and AFE5818Vca Classes in LabVIEW.

Only the basic parameters of the ADC can be changed using the properties of the **AFE5818** and **AFE5818Vca** classes. All other parameters can be edited in the AFE5818.xlsm file located in the doc SDK folder. After finishing editing, you need to click the "Create ini file" button on the "Result" page, and copy the generated AFE5818.ini to the PhotoSoundLibs\Device\ folder. When the device is turned on for the first time, a new file will be loaded into the device and the ADC will work with the new parameters.

The examples\labview\ folder contains an example afe5818.vi that implements the ADC setup described above. And in the reference section of this manual, you can find a description of all the properties and methods of the **AFE5818** and **AFE5818Vca** classes.

ADC AFE5818 setup in Visual Studio C#

The **AFE5818** ADC is configured using the **AFE5818** and **AFE5818Vca** classes. Instances of these classes are not created by the user, but by the device manager after successfully connecting to devices. The link to the created instances is stored in the **AFE5818** property of the same name of the device manager (the DeviceManager class). Before connecting devices, this property contains an empty reference (null).

To change any parameter, you just need to assign a new value to the corresponding property of the class instance, for example **deviceManager.AFE5818.LowNoiseMode = true**. This value will be automatically transferred to the device via the system bus, for example USB, and also saved in memory for later writing the settings to the configuration file. To make it easier to work with class properties, you can use the **PropertyGrid** control. If you assign it to the **SelectedObject** property a reference to the **AFE5818** class: **propertyGrid1.SelectedObject = deviceManager.AFE5818**, then you can edit all the properties of this class and the **AFE5818Vca** class for the **Vca1** and **Vca2** properties. After the user has assigned a new value to a property, the **PropertyGrid** control writes the property and then reads it back and displays the read value

in the window. Thus, the check for the range of valid values is performed automatically. In addition, the **PropertyGrid** generates lists for enumerated properties from which the user can select the desired value.

Only the basic parameters of the ADC can be changed using the properties of the **AFE5818** and **AFE5818Vca** classes. All other parameters can be edited in the AFE5818.xlsm file located in the doc SDK folder. After finishing editing, you need to click the "Create ini file" button on the "Result" page, and copy the generated AFE5818.ini to the PhotoSoundLibs\Device\ folder. When the device is turned on for the first time, a new file will be loaded into the device and the ADC will work with the new parameters.

The table below shows the code from the example project AFE5818_AFE5832 from the examples\visual\SdkExamples\ folder, which implements the ADC setup described above. And in the reference section of this guide, you can find a description of all the properties and methods of the **AFE5818** and **AFE5818Vca** classes.

Table 8: Sample Visual C# Program for Managing Data Collection

```
using PhotoSoundClasses;
using System;
using System.Windows.Forms;

namespace AFE5818_AFE5832
{
    public partial class AFE5818_AFE5832 : Form
    {
        public AFE5818_AFE5832()
        {
            InitializeComponent();
            chart1.Series.Clear();
            var series = chart1.Series.Add("ADC1/CH1");
            series.ChartType =
System.Windows.Forms.DataVisualization.Charting.SeriesChartType.FastLine;
        }

        private DeviceManager deviceManager = null;
        private short[] PlotBuffer = null;

        private void AFE5818_AFE5832_Load(object sender, EventArgs e)
        {
```

```
            deviceManager = new DeviceManager();
            deviceManager.OnConnect += OnConnectEventHandler;
            deviceManager.OnError += OnErrorEventHandler;
            deviceManager.Connect();
        }

        private void timer1_Tick(object sender, EventArgs e)
        {
            int samples = deviceManager.GetPlotData(PlotBuffer, PlotBuffer.Length, 0,
0, 0);
            if (samples > 0)
            {
                chart1.Series[0].Points.Clear();
                chart1.Series[0].Points.DataBindY(new ArraySegment<short>(PlotBuffer,
0, samples));
            }
        }

        private void OnConnectEventHandler(object sender, EventArgs e)
        {
            PlotBuffer = new short[deviceManager.MaxSamplesToCapture];
            timer1.Start();
            propertyGrid1.SelectedObject = deviceManager.AFE5818;
            rbAFE5818.Checked = true;
        }

        private void OnErrorEventHandler(object sender, MessageEventArgs e)
        {
            MessageBox.Show($"{e.Source.ToString()} error: {e.Message}");
        }

        private void AFE5818_AFE5832_FormClosed(object sender, FormClosedEventArgs e)
        {
            deviceManager?.Disconnect();
        }

        private void rbAFE5818_CheckedChanged(object sender, EventArgs e)
```

```
        {
            if (deviceManager.Connected)
            {
                RadioButton rb = sender as RadioButton;
                if (rb == rbAFE5818)
                    propertyGrid1.SelectedObject = deviceManager.AFE5818;
                else if (rb == rbAFE5832)
                    propertyGrid1.SelectedObject = deviceManager.AFE5832;
            }
        }
    }
}
```

ADC AFE5832 setup in Matlab

The AFE5832 ADC is configured using the **AFE5832** and **AFE5832Die** classes. Instances of these classes are not created by the user, but by the device manager after successfully connecting to devices. The link to the created instances is stored in the **AFE5832** property of the same name of the device manager (the **DeviceManager** class). Before connecting devices, this property contains a null reference.

To change any parameter, you just need to assign a new value to the corresponding property of the class instance, for example **dev.AFE5832. EnableAttenuatorHpf= true**. This value will be automatically transferred to the device via the system bus, for example USB, and also saved in memory for later writing the settings to the configuration file. Some parameters are represented by enumerated properties, for example, the **AttenuatorHpfCorner** property of the **AFE5832** class. These properties can only be assigned specific values, which can be viewed using the MATLAB **enumeration** command. An example of using this command is below:

```
>> enumeration(dev.AFE5832.AttenuatorHpfCorner)

Enumeration members for class 'PhotoSoundClasses.AFE5832+HpfCorners':

    C2
    C3
    C4
    C5
    C6
    C7
    C8
    C9
    C10
```

In order to assign any value to the enum property in Matlab, you must first get a list of values in a variable, and then use this variable with the index of the desired value. An example of obtaining such lists for enum properties of the **AFE5832** and **AFE5832die** classes is presented below:

```
AttenuatorHpfCorner =
 System.Enum.GetValues(dev.AFE5832.AttenuatorHpfCorner.GetType);
LpfCutoffFreq =
 System.Enum.GetValues(dev.AFE5832.Odd.LpfCutoffFreq.GetType);
HpfCutoffFreq =
 System.Enum.GetValues(dev.AFE5832.Odd.HpfCutoffFreq.GetType);
```

Below is an example of setting up the **AFE5832** ADC. In order not to transfer data to the device every time one parameter is changed, first you need to set the **AutoUpdate** property to **false**, and after changing all the parameters, you need to call the **Configure** method of the **AFE5818** class.

```
dev.AFE5832.AutoUpdate = false;
dev.AFE5832.ConfiguredAdcMask = 2^dev.MaxAdcPerDevice-1;
dev.AFE5832.ConfiguredDevicesMask = 2^dev.DevicesCount-1;
dev.AFE5832.EnableAttenuatorHpf = true;
dev.AFE5832.AttenuatorHpfCorner = AttenuatorHpfCorner(1);
dev.AFE5832.OddEqualEven = true;
dev.AFE5832.Odd.LpfCutoffFreq = LpfCutoffFreq(1);
dev.AFE5832.Odd.HpfCutoffFreq = HpfCutoffFreq(1);
dev.AFE5832.Odd.DtgcGain = 30;
dev.AFE5832.Odd.EnableLnaHpf = true;
dev.AFE5832.Odd.LowPowerMode = false;
dev.AFE5832.Odd.EnableDtgcAttenuator = true;
dev.AFE5832.Configure;
```

Only the basic parameters of the ADC can be changed using the properties of the **AFE5832** and **AFE5832Die** classes. All other parameters can be edited in the AFE5832.xlsm file located in the doc SDK folder. After finishing editing, you need to click the "Create ini file" button on the "Result" page, and copy the generated AFE5832.ini to the PhotoSoundLibs\Device\ folder. When

the device is turned on for the first time, a new file will be loaded into the device and the ADC will work with the new parameters.

The table below shows the afe5832.m script code from the examples\matlab\ folder, which implements the ADC setup described above. And in the reference section of this guide, you can find a description of all the properties and methods of the **AFE5832** and **AFE5832Die** classes.

Table 9: Example of MATLAB Script for AFE5832 ADC setup

```
filename = mfilename('fullpath');
app_path = fileparts(filename);
asm_path = fullfile(app_path,'..\..\x64\PhotoSoundClasses.dll');
asm = NET.addAssembly(asm_path);
dev = PhotoSoundClasses.DeviceManager;

disp('Connecting...');
addlistener(dev,'OnError',@onerror);
dev.Connect;
while ~dev.Connected && ~dev.ConnectFailure
    pause(0.1);
end

if dev.Connected
    disp('Successfully connected to device');

    AttenuatorHpfCorner =
System.Enum.GetValues(dev.AFE5832.AttenuatorHpfCorner.GetType);
    LpfCutoffFreq =
System.Enum.GetValues(dev.AFE5832.Odd.LpfCutoffFreq.GetType);
    HpfCutoffFreq =
System.Enum.GetValues(dev.AFE5832.Odd.HpfCutoffFreq.GetType);

    dev.AFE5832.AutoUpdate = false;
    dev.AFE5832.ConfiguredAdcMask = 2^dev.MaxAdcPerDevice-1;
    dev.AFE5832.ConfiguredDevicesMask = 2^dev.DevicesCount-1;
    dev.AFE5832.EnableAttenuatorHpf = true;
    dev.AFE5832.AttenuatorHpfCorner = AttenuatorHpfCorner(1);
    dev.AFE5832.OddEqualEven = true;
    dev.AFE5832.Odd.LpfCutoffFreq = LpfCutoffFreq(1);
    dev.AFE5832.Odd.HpfCutoffFreq = HpfCutoffFreq(1);
```

```
    dev.AFE5832.Odd.DtgcGain = 30;
    dev.AFE5832.Odd.EnableLnaHpf = true;
    dev.AFE5832.Odd.LowPowerMode = false;
    dev.AFE5832.Odd.EnableDtgcAttenuator = true;
    dev.AFE5832.Configure;

    data = NET.createArray('System.Int16',dev.MaxSamplesToCapture);
    adc = 0;
    chan = 0;

    fig = figure('Name','Plot data example');
    while isvalid(fig)
        samples = dev.GetPlotData(data,data.Length,0,adc,chan);
        if samples > 0
            tmp = int16(data);
            plot(tmp(1:samples));
        end
        pause(0.1);
    end
else
    disp('Failed to connect to device');
end

dev.Disconnect;
disp('Disconnected');
```

ADC AFE5832 setup in LabVIEW

The AFE5832 ADC is configured using the **AFE5832** and **AFE5832Die** classes. Instances of these classes are not created by the user, but by the device manager after successfully connecting to devices. The link to the created instances is stored in the **AFE5832** property of the same name of the device manager (the **DeviceManager** class). Before connecting devices, this property contains a null reference.

To change any parameter, you just need to assign a new value to the corresponding property of the class instance, as shown in the figures below. This value will be automatically transferred to the device via the system bus, for example USB, and also saved in memory for later writing the settings to the configuration file. In LabVIEW, instead of making your own value change handler for each control, you can update multiple properties in a common handler. Since the user can change the value of only one control at a time, there will be only one new value in the handler. An internal check in the class will reveal this new value and the settings will be transferred to the device via the system bus once.

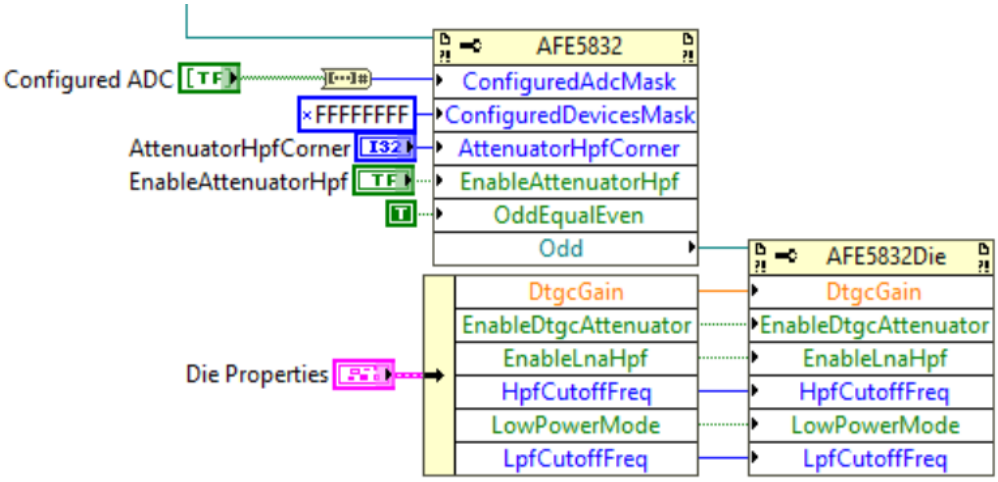

Figure 15: Modifying the Properties of the AFE5832 and AFE5832Die Classes in LabVIEW

Each class property has a certain range of valid values. When you assign a value to a property, it is validated and the property is changed only if the new value is in that range. Therefore, when creating a graphical user interface, you should read the property immediately after assignment and update the corresponding control with the read value. So, the user will be able to see that the value entered by him is incorrect and it was not saved and was not transferred to the device. The pictures below show how new property values can be read.

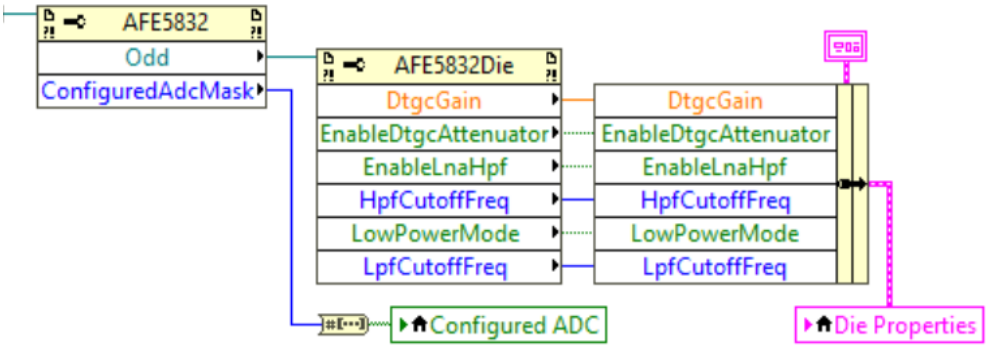

Figure 16 Reading Properties of the AFE5832 and AFE5832Die Classes in LabVIEW

Only the basic parameters of the ADC can be changed using the properties of the **AFE5832** and **AFE5832Die** classes. All other parameters can be edited in the AFE5832.xlsm file located in the doc SDK folder. After finishing editing, you need to click the "Create ini file" button on the "Result" page, and copy the generated AFE5832.ini to the PhotoSoundLibs\Device\ folder. When the device is turned on for the first time, a new file will be loaded into the device and the ADC will work with the new parameters.

The examples\labview\ folder contains an example afe5832.vi that implements the ADC setup described above. And in the reference section of this guide, you can find a description of all the properties and methods of the **AFE5832** and **AFE5832Die** classes.

ADC AFE5832 setup in Visual Studio C#

The AFE5832 ADC is configured using the **AFE5832** and **AFE5832Die** classes. Instances of these classes are not created by the user, but by the device manager after successfully connecting to devices. The link to the created instances is stored in the **AFE5832** property of the same name of the device manager (the **DeviceManager** class). Before connecting devices, this property contains a null reference.

To change any parameter, you just need to assign a new value to the corresponding property of the class instance, for example **dev.AFE5832. EnableAttenuatorHpf= true**. This value will be automatically transferred to the device via the system bus, for example USB, and also saved in memory for later writing the settings to the configuration file. To make it easier to work with class properties, you can use the **PropertyGrid** control. If you assign it to the **SelectedObject** property a reference to the **AFE5832** class: **propertyGrid1.SelectedObject = deviceManager.AFE5832**, then you can edit all the properties of this class and the **AFE5832Die** class for the **Odd** and **Even** properties. After the user has assigned a new value to a property, the **PropertyGrid** control writes the property and then reads it back and displays the read value in the window. Thus, the check for the range of valid values is performed automatically. In addition, the **PropertyGrid** generates lists for enumerated properties from which the user can select the desired value.

Only the basic parameters of the ADC can be changed using the properties of the **AFE5832** and **AFE5832Die** classes. All other parameters can be edited in the AFE5832.xlsm file located in the doc SDK folder. After finishing editing, you need to click the "Create ini file" button on the "Result" page, and copy the generated AFE5832.ini to the PhotoSoundLibs\Device\ folder. When the device is turned on for the first time, a new file will be loaded into the device and the ADC will work with the new parameters.

The table () shows the code from the example project AFE5818_AFE5832 from the examples\visual\SdkExamples\ folder, which implements the ADC setup described above. In the reference section of this guide, you can find a description of all the properties and methods of the **AFE5832** and **AFE5832Die** classes.

Real-time data processing in MATLAB

Data processing in real time in the MATLAB environment is discussed below using the example of constructing a sonogram. As a result of processing, the user can see a video image on the screen displaying information about the signal in the sensors of the connected ultrasound sensor.

Receiving ADC data for processing is carried out using the **DataLogger** class. Instances of this class (data loggers) are created by the user using the **CreateLogger** method of the device manager (the **DeviceManager** class):

```
logger = dev.CreateLogger('RealTime',1);
```

Call the **CreateLogger** method only after connecting to devices, otherwise the method returns an empty reference. The method returns a reference to the created data logger, and its arguments are the name of the created data logger and the length of the queue when writing to memory. The name is used to save the settings of the recorder in the configuration file. The queue length is measured in ADC data frames and can be 1 or more. For a queue with losses, as a rule, 1 is sufficient, and for queues without losses, this value should be selected experimentally so that there are no data gaps during processing.

Next, you should set the properties of the created recorder, which determine the duration of data entry and the device from which the data is read:

```
logger.LimitLoggingTime = false;
logger.LimitNumFrames = false;
logger.DevicesMask = 1;
```

The logger starts writing ADC data to a queue in memory immediately after calling its **StartLoggingToMemory(LossyQueue)** method, where **LossyQueue = true** for lossy queues:

```
logger.StartLoggingToMemory(true);
```

Before retrieving data from the queue, you need to prepare a buffer for them in memory. This can be done using the Matlab **NET.createArray** command. The amount of allocated memory can be set to the maximum, and then the actual amount of data can be determined:

```
FrameBuffer =
NET.createArray('System.Int16',dev.MaxSamplesToCapture*dev.MaxChannelsToCaptu
re);
```

The **GetFrame** method is directly involved in retrieving data from memory:

```
[valid,channels,samples,frame_num,trig_time,trig_src,sample_rate] =
logger.GetFrame(FrameBuffer,false);
```

The input arguments of the method are the allocated data buffer in memory and the sign of frame transposition. If it is **true**, then the first index (row) specifies the sample number (time), and the second (column) specifies the channel number. Otherwise, the line defines the channel, and the column defines the time. When called, the method expects data and, if the timeout has not expired, then it returns **valid = true** and fills the rest of the output arguments with the parameters of the data frame. The output parameters of the data frame are described in more detail in the reference section.

The construction of a sinogram is reduced to the permutation of the data in accordance with the channel map. The channel map is an ordered array with ADC channel numbers, the array index corresponds to the channel number of the ultrasound sensor:

```
    tmpData = single(FrameBuffer);
    frame = reshape(tmpData(1:(channels*samples)),samples,channels);
    mapped = frame(:,chmap);
```

At the end of processing, the result is scaled by the range of values and displayed as an image with a specified color palette on the screen:

```
    image = imagesc('XData',1:channels,'YData',1:samples,'CData',
mapped/max(max(mapped)), [-1 1]);
```

The table below shows the realtime.m script code from the examples \ matlab folder, which implements the data processing described above. In the reference section of this tutorial, you can find a description of all the properties and methods of the **DataLogger** class.

Table 10: An example MATLAB script for real-time processing of ADC data

```
filename = mfilename('fullpath');
app_path = fileparts(filename);
asm_path = fullfile(app_path,'..\..\x64\PhotoSoundClasses.dll');
asm = NET.addAssembly(asm_path);
dev = PhotoSoundClasses.DeviceManager;

disp('Connecting...');
addlistener(dev,'OnError',@onerror);
dev.Connect;
```

```
while ~dev.Connected && ~dev.ConnectFailure
    pause(0.1);
end

if dev.Connected
    disp('Successfully connected to device');
    image = [];

    if dev.Devices(1).SensorsMapLoaded
        fig = figure('Name','Plot data example');
        set(gca,'nextplot','replacechildren','YDir','reverse');
        chmap = double(dev.Devices(1).ChannelsMap)+1;

        logger = dev.CreateLogger('RealTime',1);
        logger.DevicesMask = 1;
        logger.LimitLoggingTime = false;
        logger.LimitNumFrames = false;
        FrameBuffer =
NET.createArray('System.Int16',dev.MaxSamplesToCapture*dev.MaxChannelsToCap
ture);
        logger.StartLoggingToMemory(true);

        while isvalid(fig)

[valid,channels,samples,frame_num,trig_time,trig_src,sample_rate] =
logger.GetFrame(FrameBuffer,false);
            if valid
                tmpData = single(FrameBuffer);
                frame =
reshape(tmpData(1:(channels*samples)),samples,channels);
                mapped = frame(:,chmap);
                xlim([0 channels])
                ylim([0 samples])
                colorbar
                xlabel('Channels')
                ylabel('Samples')
                if isempty(image)
                    image =
imagesc('XData',1:channels,'YData',1:samples,'CData',
mapped/max(max(mapped)), [-1 1]);
                else
                    set(image,'CData',mapped/max(max(mapped)));
                end
            end
            pause(0.1);
        end
    else
        disp('Sensors map was not assigned!');
    end
else
    disp('Failed to connect to device');
end

dev.Disconnect;
fprintf('\nDisconnected\n');
```

Real-time data processing in LabVIEW

Data processing in real time in the Labview environment is discussed below using the example of building a sonogram. As a result of processing, the user can see a video image on the screen displaying information about the signal in the sensors of the connected ultrasound sensor.

Receiving ADC data for processing is carried out using the **DataLogger** class. Instances of this class (data loggers) are created by the user using the **CreateLogger** method of the **DeviceManager** as shown in the figure below.

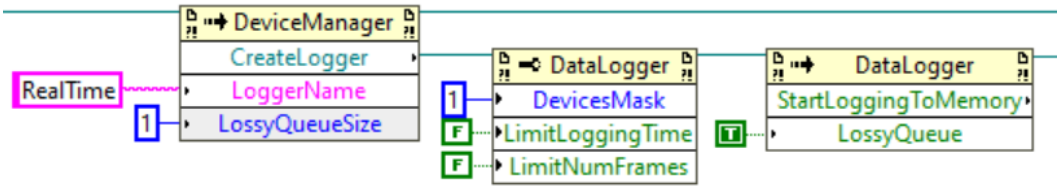

Figure 17: Creating and Configuring a Data Logger in LabVIEW

Call the **CreateLogger** method only after connecting to devices, otherwise the method returns an empty reference. The method returns a reference to the created data logger, and its arguments are the name of the created data logger and the length of the queue when writing to memory. The name is used to save the settings of the recorder in the configuration file. The queue length is measured in ADC data frames and can be 1 or more. For a queue with losses, as a rule, 1 is sufficient, and for queues without losses, this value should be selected experimentally so that there are no data gaps during processing.

Next, you should set the properties of the created recorder, which determine the duration of data entry and the device from which the data is read. After that, you can start writing ADC data to the memory queue using the **StartLoggingToMemory(LossyQueue)** method, where **LossyQueue = true** for lossy queues (Figure 17).

Before retrieving data from the queue, you need to prepare a buffer for them in memory. The amount of allocated memory can be set to the maximum, and then the actual amount of data can be determined (Figure 18).

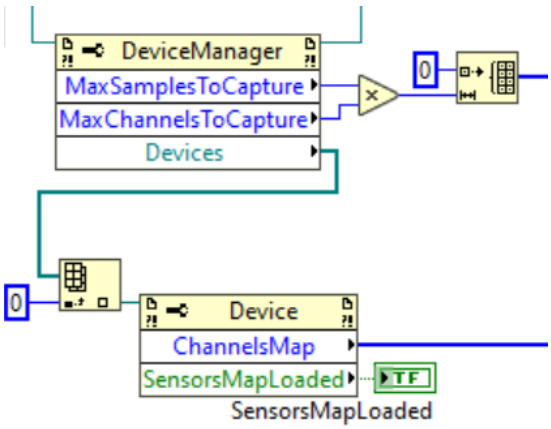

Figure 18: Preparing to process data in LabVIEW

The **GetFrame** method is directly involved in retrieving data from memory. The input arguments of the method are the allocated data buffer in memory and the sign of frame transposition. If it is **true**, then the first index (row) specifies the sample number (time), and the second (column) specifies the channel number. Otherwise, the line defines the channel, and the column defines the time. When called, the method expects data and, if the timeout has not expired, then it returns **valid = true** and fills the rest of the output arguments with the parameters of the data frame. The output parameters of the data frame are described in more detail in the reference section. The data copied to the data buffer should be limited in length according to the output arguments **FrameChannels** and **FrameSamples** and converted to a two-dimensional array for further processing (Figure 19).

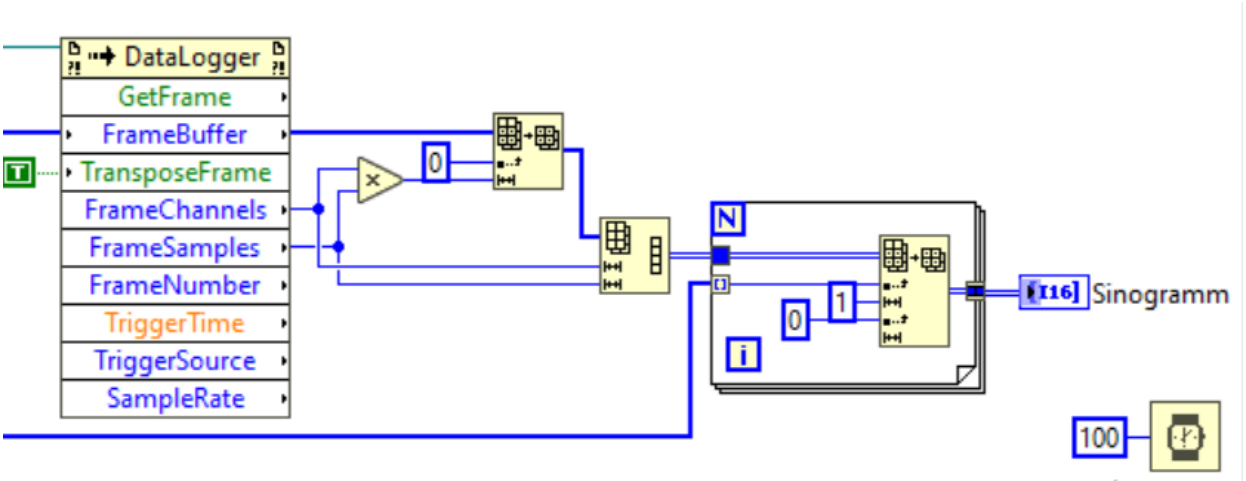

Figure 19: Retrieving data from memory in LabVIEW

The construction of a sinogram is reduced to the permutation of the data in accordance with the channel map. The channel map is an ordered array with ADC channel numbers, the array index corresponds to the channel number of the ultrasound sensor (Figure 18). The result of the permutation is scaled by the range of values and displayed as an image with a specified color palette on the screen as shown in the figure above.

The \examples\labview\ folder contains a realtime.vi example that implements the above data processing. In the reference section of this tutorial, you can find a description of all the properties and methods of the **DataLogger** class.

Real-time data processing in Visual C#

Data processing in real time in the Visual C# environment is discussed below using the example of building a sonogram. As a result of processing, the user can see a video image on the screen displaying information about the signal in the sensors of the connected ultrasound sensor.

Receiving ADC data for processing is carried out using the **DataLogger** class. Instances of this class (data loggers) are created by the user using the **CreateLogger** method of the device manager (the **DeviceManager** class):

```
logger = deviceManager.CreateLogger("RealTime",1);
```

Call the **CreateLogger** method only after connecting to devices, otherwise the method returns an empty reference. The method returns a reference to the created data logger, and its arguments are the name of the created data logger and the length of the queue when writing to memory. The name is used to save the settings of the recorder in the configuration file. The queue length is measured in ADC data frames and can be 1 or more. For a queue with losses, as a rule, 1 is sufficient, and for queues without losses, this value should be selected experimentally so that there are no data gaps during processing.

Next, you should set the properties of the created recorder, which determine the duration of data entry and the device from which data is read:

```
logger.LimitLoggingTime = false;
logger.LimitNumFrames = false;
logger.DevicesMask = 1;
```

The logger starts writing ADC data to a queue in memory immediately after calling its **StartLoggingToMemory(LossyQueue)** method, where **LossyQueue = true** for lossy queues:

```
logger.StartLoggingToMemory(true);
```

Before retrieving data from the queue, you need to allocate a buffer for them in memory. The amount of allocated memory can be set to the maximum, and then the actual amount of data can be determined:

```
frameBuffer = new short[deviceManager.MaxSamplesToCapture *
deviceManager.MaxChannelsToCapture];
```

The **GetFrame** method is directly involved in retrieving data from memory:

```
logger.GetFrame(frameBuffer, transposeFrame, out int frameChannles, out int
frameSamples, out uint frameNumber, out double triggerTime, out int triggerSource, out
int sampleRate)
```

The method input arguments are the allocated data buffer in memory frameBuffer and the **transposeFrame** flag. If it is **true**, then the first index (row) specifies the sample number (time), and the second (column) specifies the channel number. Otherwise, the line defines the channel, and the column defines the time. When called, the method expects data and if the timeout has not expired, then it returns **true** and fills the output arguments with the parameters of the data frame. The output parameters of the data frame are described in more detail in the reference section.

The construction of a sinogram is reduced to the permutation of the data in accordance with the channel map. The channel map is an ordered array with ADC channel numbers, the array index corresponds to the channel number of the ultrasound sensor:

```
int[] map = deviceManager.Devices[0].ChannelsMap;
```

The result of the permutation is scaled by the range of values and displayed as an image with a specified color palette on the screen. Since data processing requires performance, the processing code is enclosed in an **unsafe** block:

```
unsafe
{
   byte* row = (byte*)bmpData.Scan0;
   int n = 0;
   for (int f = 0; f < bmp.Height; f++)
   {
       for (int w = 0; w < bmp.Width; w++)
         row[w] = (byte)Math.Round((frameBuffer[f*frameChannles+map[w]]-min)*scale);
       row += stride;
    }
}
```

To avoid freezing of the user interface, data processing is carried out in a separate thread of the **BackgroundWorker** class. The methods and properties of the **DataLogger** class are ThreadSafe, but they can raise events, such as the Device Manager **OnError** event. If the event handler uses graphical interface elements, then they should be accessed through the **Invoke** method of these elements. Below is an example of displaying an error message using an element of the **Label** type:

```
private void OnErrorEventHandler(object sender, MessageEventArgs e)
{
   labelFPS.Invoke((MethodInvoker) delegate { labelFPS.Text = $"{e.Source.ToString()}
error: {e.Message}"; });
}
```

The examples \ visual \ SdkExamples folder contains the RealTime project, which implements the data processing described above. In the reference section of this tutorial, you can find a description of all the properties and methods of the **DataLogger** class.

PhotoSoundClasses.dll Class Library Reference

Capture class

Table 1 Capture Properties and Methods

Name	Type	Description
Configure	method	Restarts ADC data collection with parameters from class properties
SamplesToCapture	Integer, 32 bit	Number of data collection counts for the ADC channel
FramesPerPacket	Integer, 32 bit	Number of ADC data frames in one data bus package
DecimationFactor	Integer, 32 bit	ADC thinning factor
WaitTrigger	Boolean	Sign of waiting for trigger event before data collection starts
EnabledAdcMask	Unsigned, 32 bit	ADC mask allowed to collect data
AutoUpdate	Boolean	A mark of automatic transfer of settings to the device when the class properties change

Configure

The **Configure** method of the **Capture** class disables the collection of ADC data, then transfers the settings calculated from the properties of the **Capture** class to the device via the system bus, and then enables the collection of ADC data again. The method also sets the **AutoUpdate** property of the **Capture** class to true.

SamplesToCapture

The **SamplesToCapture** property of the **Capture** class sets the number of data samples per ADC channel, which are written to the data frame buffer in the device after the acquisition is started, and then transferred to the PC via the system bus. The maximum number of samples depends on the size of the device framebuffer. To find out the maximum number of samples for a specific device, you need to read the value of the **MaxSamples** property of the corresponding instance of the **Device** class.

FramesPerPacket

The **FramesPerPacket** property of the **Capture** class specifies the number of ADC data frames in one packet transmitted over the system data bus. Since the PC interrupt system has a limited frequency of operation, the ADC data frames are combined into a packet to reduce the interrupt frequency. This allows you to increase the data transfer rate up to the bandwidth of the system bus. On the other hand, the frequency of receiving frames by the user program is reduced by FramesPerPacket times. This can result in too low the refresh rate on the graph if the acquisition is triggered by a trigger event with a low repetition rate. Therefore, in such cases, the

FramesPerPacket value should be set equal to 1. In other cases, **FramesPerPacket** can be left equal to 10.

DecimationFactor

The **DecimationFactor** property of the Capture class controls the decimation of ADC data samples. If it is equal to 1, then the samples are written to the device frame buffer without gaps. If **DecimationFactor** = 2, then count 1 is written, then skip recording, then count 3 is written, etc. If **DecimationFactor** = 3, then only every third sample is recorded. Thus, the sampling rate of the output data transmitted to the PC is equal to the sampling rate of the ADC divided by the **DecimationFactor**. To find out the ADC sampling rate for a specific device, you need to read the value of the **MaxSampleRate** property of the corresponding instance of the **Device** class. This data decimation can result in aliasing if the input bandwidth of the ADC is greater than half the sample rate of the output data. To eliminate this effect, you can adjust the bandwidth of the ADC low-pass filter (see description of ADC classes below).

WaitTrigger

The **WaitTrigger** property of the **Capture** class enables or disables waiting for a trigger event before starting data collection. If waiting is disabled, then a new start of data collection follows immediately after the end of the transfer of the previous ADC data frame from the device buffer to the PC. If enabled, after the end of the data transfer, a trigger event is first expected and then followed by a trigger. If the repetition rate of the trigger events is too high, then the trigger event may occur before the end of the data transfer to the PC. In this case, the data collection will not start, but the event counter from the trigger will be incremented, and the event skip will be recorded. The number of missed events can be read from the **LostEvents** property of the **Device** class.

EnabledAdcMask

The **EnabledAdcMask** property of the **Capture** class sets the ADC chips allowed for data collection. This property is common to all devices, so the number of ADC chips is determined by the device with the most installed chips. This number can be read from the **MaxAdcPerDevice** property of the Device Manager (**DeviceManager** class). Each bit of **EnabledAdcMask** corresponds to one ADC chip, bit 0 to chip # 1, bit 1 to chip # 2, and so on. If the bit value is 1, then the ADC is enabled for data acquisition, otherwise it is disabled. A reduced number of ADC chips may be required to reduce the amount of data transferred to the PC. This allows for faster transfer times and higher trigger event repetition rates, as well as reducing the size of data files when written to disk.

AutoUpdate

The **AutoUpdate** property of the **Capture** class enables or disables automatic transmission of settings to the device when the properties of the **Capture** class are changed. If the property value is **true**, then when writing a new value to any of the properties of the **Capture** class, the updated settings are automatically transferred to the device. If the property value is **false**, then

you can assign new values to several properties of the **Capture** class, and then call the **Configure** method, which will transfer the settings to the device and restore the **AutoUpdate** property to true.

Trigger class

Table 2 Properties and methods of the Trigger class

Name	Type	Description
Configure	method	Takes settings from class properties to a system bus
GetInputFrequencies	method	Reads from the device on the system bus the frequency of signals at trigger inputs
TriggerOutputs	array TriggerOutput	TriggerOutput Class Array
InputNames	Massive string	An array with trigger input names
SlaveDelays	Integer Array, 32 bit	Array of data acquisition start delays in relation to the signal from the slave HDMI connector
GeneratorFrequency	Double	Internal generator frequency, Hz
ConnectToGenerator	Boolean	Sign of permission to use an internal generator to start data collection
InputsDelay	Integer, 32 bit	Delaying the start of data collection in relation to the signal from the selected trigger or generator input at the clock of the ADC sampling frequency
InputsGuard	Integer, 32 bit	Interval of protection against noise at the trigger inputs in clock cycles of the ADC sampling frequency.
EnabledInputsMask	Unsigned, 32 bit	Trigger input mask allowed to start data collection
InvertedInputsMask	Unsigned, 32 bit	Trigger input mask with negative input polarity
AutoUpdate	Boolean	Sign of automatic transmission of settings to the device when class properties change

Configure

The **Configure** method of the **Trigger** class transfers the settings calculated from the properties of the **Trigger** class to the device via the system bus, and also sets the AutoUpdate property to **true**. Although the **Trigger** class contains an array of instances of the **TriggerOutput** class, settings from the properties of the **TriggerOutput** class are not passed to the device. To do this, the **TriggerOutput** class has its own **Configure** method and each trigger output is configured separately.

GetInputFrequencies

The **UpdateInputFrequencies** method of the **Trigger** class reads the signal frequencies at the trigger inputs from the device via the system bus and returns an array of numbers with the signal frequencies in hertz.

TriggerOutputs

The **TriggerOutputs** property of the **Trigger** class is an array of instances of the **TriggerOutputs** class. The length of the array is equal to the number of trigger outputs and each element of the array corresponds to its own trigger output.

InputNames

The **InputNames** property of the **Trigger** class is an array of strings with custom trigger input names. Strings can contain any descriptive text for the convenience of labeling inputs.

SlaveDelays

The **SlaveDelays** property of the **Trigger** class is an array of integers with delays in triggering data collection for each slave. The array length is 14, the maximum number of slaves. Array element 0 corresponds to the first slave device and element 13 to the last in the HDMI cable chain. The delay is calibrated against the analog signal input from a single source, fed to the ADC inputs using cables of equal length. Delay values must be between 0 and the value of the InputsDelay property of the master.

GeneratorFrequency

The **GeneratorFrequency** property of the **Trigger** class sets the frequency of the internal oscillator in hertz. The internal oscillator can serve as a trigger signal source for both triggering data acquisition and for any of the trigger outputs.

ConnectToGenerator

The **ConnectToGenerator** property of the **Trigger** class enables or disables the use of an internal generator to trigger data collection. To start collecting data from the generator, you must also enable waiting for the trigger event (the **WaitTrigger** property of the **Capture** class).

InputsDelay

The **InputsDelay** property of the **Trigger** class sets the delay in starting data acquisition in relation to the trigger signal in ADC sampling clock cycles. The trigger signal can come from one or more trigger inputs or from an internal oscillator.

InputsGuard

The **InputsGuard** property of the **Trigger** class sets the noise protection interval at the trigger inputs in ADC sampling clock cycles. The trigger event is latched on the rising edge of the trigger signal for positive signal polarity, or on the falling edge for negative polarity. Pulse noise at the

trigger input can cause multiple edges to be detected. Although the acquisition will start on the first edge, the remaining edges can lead to incorrect trigger event counts and a false number of trigger events to be missed.

EnabledInputsMask

The **EnabledInputsMask** property of the **Trigger** class specifies the trigger inputs allowed to start data collection. Each bit of **EnabledInputsMask** corresponds to one trigger input, bit 0 to input # 1, bit 1 to input # 2, and so on. If the bit value is 1, then the signal from the trigger input starts data collection, if 0 - then no. If multiple inputs are allowed to be triggered, then data collection will be triggered by a signal from any of these inputs. The number of the input, on the signal from which the data collection was started, is memorized by the device and transmitted to the PC in the ADC data frame. To start collecting data from a trigger input, you must also enable waiting for a trigger event (the **WaitTrigger** property of the **Capture** class).

InvertedInputsMask

The **InvertedInputsMask** property of the **Trigger** class specifies the trigger inputs with negative polarity of the input signal. Each bit of **InvertedInputsMask** corresponds to one trigger input, bit 0 - input # 1, bit 1 - input # 2, etc. If the bit value is 1, then the trigger input has a negative polarity of the input signal, if 0 - then positive.

AutoUpdate

The **AutoUpdate** property of the **Trigger** class enables or disables automatic transmission of settings to the device when the properties of the **Trigger** class are changed. If the property value is true, then when writing a new value to any of the property of the **Trigger** class, the updated settings are automatically transferred to the device. If the property value is **false**, then you can assign new values to several properties of the **Trigger** class, and then call the **Configure** method, which will transfer the settings to the device and restore the **AutoUpdate** property to true.

TriggerOutput class

Table 3: TriggerOutput class properties and methods

Name	Type	Description
Configure	method	Takes settings from class properties to a system bus
PulseWidth	Double	The duration of the trigger release in microseconds
Delay	Double	Signal delay at trigger output in microseconds
SourcesMask	Unsigned, 32 bit	Trigger input mask connected to trigger exit
ConnectToGenerator	Boolean	Sign of an internal generator connecting to the trigger exit

Enable	Boolean	Signal resolution sign at trigger input
InvertInputsDelay	Boolean	Sign of negative polarity of the signal at the exit of the trigger
AutoUpdate	Boolean	A sign of automatic transfer of settings to the device when the class properties change

Configure

The **Configure** method of the **TriggerOutput** class transfers the settings calculated from the properties of the **TriggerOutput** class to the device via the system bus, and also sets the **AutoUpdate** property to true.

PulseWidth

The **PulseWidth** property of the **TriggerOutput** class sets the pulse width at the trigger output in microseconds. The pulse duration should not exceed the pulse repetition period at the trigger output.

Delay

The **Delay** property of the **TriggerOutput** class sets the delay of the signal at the trigger output in relation to the signal at the connected trigger input or to the signal from the internal generator in microseconds.

SourcesMask

The **SourcesMask** property of the **TriggerOutput** class determines which trigger inputs are connected to the trigger's output. Each bit of the **SourcesMask** corresponds to one trigger input, bit 0 to input # 1, bit 1 to input # 2, and so on. If the bit value is 1, then the trigger input is connected to the trigger output, otherwise it is not connected. If several inputs are connected to the output, then the output signal is a logical OR function of signals from the trigger inputs.

ConnectToGenerator

The **ConnectToGenerator** property of the **TriggerOutput** class connects or disables the internal generator to the trigger output. The internal oscillator can be connected in conjunction with one or more trigger inputs. In this case, the signal at the trigger output is a logical OR function of signals from the trigger inputs.

Enable

The **Enable** property of the **TriggerOutput** class enables or disables the pulse signal at the trigger output.

Invert

The **Invert** property of the **TriggerOutput** class determines the polarity of the signal at the trigger output. If it is **true**, then the signal polarity is negative, if **false** – positive.

AutoUpdate

The **AutoUpdate** property of the **TriggerOutput** class enables or disables automatic transmission of settings to the device when the properties of the **Trigger** class are changed. If the property value is **true**, then when writing a new value to any of the property of the **Trigger** class, the updated settings are automatically transferred to the device. If the property value is **false**, then you can assign new values to several properties of the **Trigger** class, and then call the Configure method, which will transfer the settings to the device and restore the **AutoUpdate** property to true.

DataLogger class

Table 4: DataLogger class properties and methods

Name	Type	Description
Configure	method	Calls for OnPropertyChanged device manager DeviceManager
StartLoggingToFile	method	Starts recording data in a file
StartLoggingToMemory	method	Launches memory record
StopLogging	method	Stops recording data
GetFrame	method	Extracts ADC data frame from memory queue
OnStartLogging	event	Event is called when you start recording data
OnStopLogging	event	Event is triggered when data records stop
LimitNumFrames	Boolean	Sign of limiting the number of frames recorded
LimitLoggingTime	Boolean	Sign of time limit
LimitFileSize	Boolean	Sign of limiting the size of the data file
DataFolder	String	Full way to the data file folder
DevicesMask	Unsigned, 32 bit	A mask of devices whose data is recorded in memory or file
MaxLoggedFramesInputsDelay	Integer, 32 bit	Maximum number of recorded data frames
MaxFileSize	Integer, 32 bit	Maximum data file size in megabytes
LoggingTimeout	Integer, 32 bit	Maximum time to write in a file or memory in seconds
Logging	Boolean	Sign of active data recording
Progress	Integer, 32 bit	Percentage data record progress
NumLoggedFrames	Integer, 32 bit	Current number of recorded data frames
LoggingTime	Double	Current memory or file time
FileSize	Double	Current data file size in megabytes
AutoUpdate	Boolean	A sign of automatic transfer of settings to the device when the class properties change

Configure

The **Configure** method for the **DataLogger** class does nothing and is reserved for future reference.

StartLoggingToFile

The **StartLoggingToFile**(**FileName**) method of the **DataLogger** class starts writing ADC data to a file. The **FileName** argument of type **String** passes the name of the file without the extension and without the file path. The method returns **true** if recording started without errors.

StartLoggingToMemory

The **StartLoggingToMemory(LossyQueue)** method of the **DataLogger** class starts writing ADC data to a queue in memory. A **LossyQueue** of type Boolean indicates to create a lossy queue, otherwise a lossless queue will be created. The queue length is specified when you instantiate the **DataLogger** class in the **CreateLogger** method of the **DeviceManager**. The method returns **true** if recording started without errors.

StopLogging

The **StopLoggin** method of the **DataLogger** class stops writing ADC data to a file or memory. Since it takes some time to complete writing to the file, you should wait until the end of writing by checking the **Logging** flag of the **DataLogger** class or wait for the **OnStopLogging** event of the **DataLogger** class.

GetFrame

The **GetFrame(FrameBuffer, TransposeFrame, FrameChannels, FrameSamples, FrameNumber, TriggerTime, TriggerSource, SampleRate)** method of the **DataLogger** class waits for one ADC data frame, fetches it from the queue in memory, and copies it to the provided **FrameBuffer**. The **TransposeFrame** argument must be **true** if the two-dimensional data array is to have row feeds (first index) and column-wise channels, and must be **false** if the two-dimensional data array must have row feeds and column feeds. The rest of the arguments are links for receiving the parameters of the data frame: **FrameChannels** - the number of ADC channels in the frame, **FrameSamples** - the number of ADC samples in the frame, **FrameNumber** - the sequence number of the frame, **TriggerTime** - the countdown of the trigger event for this frame in milliseconds, **SampleRate** – the sampling rate of the frame data in hertz. The method returns **true** if the data was retrieved from the queue successfully and **false** if the data timed out the frame.

OnStartLogging

The **OnStartLogging** event of the **DataLogger** class is called immediately after the successful start of writing data to a file or memory. The event handler must have standard arguments of type **object** and **EventArgs**.

OnStopLogging

The **OnStopLogging** event of the **DataLogger** class is called upon automatic or forced completion of writing data to a file or memory. The event handler must have standard arguments of type **object** and **EventArgs**.

LimitNumFrames

The **LimitNumFrames** property of the **DataLogger** class enables or disables automatic stopping of data writing to a file or memory if the number of ADC data frames written is equal to the maximum value set by the **MaxLoggedFrames** property of the **DataLogger** class.

LimitLoggingTime

The **LimitLoggingTime** property of the **DataLogger** class enables or disables automatic stopping of data logging to a file or memory if the logging time exceeds the maximum value set by the **LoggingTimeout** t property of the **DataLogger** class.

LimitFileSize

The **LimitFileSize** property of the **DataLogger** class enables or disables automatic stopping of writing data to a file or memory if the file size has exceeded the maximum value set by the **MaxFileSize** property of the **DataLogger** class.

DataFolder

The **DataFolder** property of the **DataLogger** class specifies the full path to the folder where the logged data files are stored.

DevicesMask

The **DevicesMask** property of the **DataLogger** class defines the devices from which data is written to a file or memory. Each DevicesMask bit corresponds to one device, bit 0 - to a device with **Id** = 0, bit 1 - to a device with **Id** = 1, etc. If the bit value is 1, then data from the device is written, otherwise it is not.

MaxLoggedFrames

The **MaxLoggedFrames** property of the **DataLogger** class specifies the maximum number of ADC data frames to write. To stop recording when the maximum number of recorded frames is exceeded, you must also set the **LimitNumFrames** property of the **DataLogger** class to **true**.

MaxFileSize

The **MaxFileSize** property of the **DataLogger** class specifies the maximum size of a data file in megabytes. To stop recording when the maximum file size is exceeded, you must also set the **LimitFileSize** property of the **DataLogger** class to **true**.

LoggingTimeout

The **LoggingTimeout** property of the **DataLogger** class specifies the maximum time for writing data to a file or memory in seconds. To stop recording when the maximum recording time is exceeded, you must also set the **LimitLoggingTime** property of the **DataLogger** class to **true**.

Logging

The **Logging** property of the **DataLogger** class indicates the state of the data log and is a read-only property. If the property value is **true**, then the recording is made, if **false** – then no.

Progress

The **Progress** property of the **DataLogger** class shows the current progress of writing data as a percentage and is a read-only property. The recording progress is calculated either to end the recording by exceeding the number of frames, or by exceeding the file size. In this case, the progress value is displayed for the condition that will be fulfilled earlier. Stopping recording at least one of the conditions must be allowed by the **LimitFileSize** or **LimitNumFrames** properties of the **DataLogger** class.

NumLoggedFrames

The **NumLoggedFrames** property of the **DataLogger** class shows the current number of ADC data frames written to file or memory and is a read-only property.

LoggingTime

The **LoggingTime** property of the **DataLogger** class shows the current time in seconds since the start of writing data to a file or memory and is a read-only property.

FileSize

The **FileSize** property of the **DataLogger** class shows the current size of the data file in megabytes and is a read-only property.

AutoUpdate

The **AutoUpdate** property of the **DataLogger** class enables or disables notification through the Device Manager **OnPropertyChanged** event when the properties of the **DataLogger** class change. Before starting to batch change the properties of the **DataLogger** class, this property can be set to **false**, and when finished, **true**. In this case, the **OnPropertyChanged** event will be called only once.

Class AFE5818

Table 5: Properties and Methods of AFE5818 class

Name	Type	Description
Configure	method	Takes settings from class properties to a system bus
ConfiguredDevicesMask	Unsigned, 32 bit	Device mask for configuration
ConfiguredAdcMask	Unsigned, 32 bit	Mask of ADC chips for configuration
Vca1EqualsVca2	Boolean	If true, then the settings for VCA #2 are taken from the Vca1 property, if false, then from Vca2
Vca1, Vca2	Class AFE5818Vca	References to class AFE5818Vca with settings for VCA #1 and for VCA #2
AutoUpdate	Boolean	Sign of automatic transmission of settings to the device when class properties change

Configure

The **Configure** method of the **AFE5818** class transfers the settings calculated from the properties of the **AFE5818** class to the device via the system bus, and also sets the **AutoUpdate** property to **true**.

ConfiguredDevicesMask

The **ConfiguredDevicesMask** property of the **AFE5818** class sets the device mask for configuration. Each bit of the mask corresponds to one device: bit 0 - to a device with Id = 0, bit 1 - to a device with Id = 1, etc., where Id is an identifier of a device on the system bus. When changing the mask, the settings are not automatically transferred to the device, but the new mask is taken into account when changing other properties. So, if any property is assigned the same value, then if the mask has not been changed, then the settings will not be transferred to the device, and if they were changed, they will be. The number of connected devices can be read from the **DevicesCount** property of the device manager.

ConfiguredAdcMask

The **ConfiguredAdcMask** property of the **AFE5818** class sets the mask of the ADC chips for configuration. Each bit of the mask corresponds to one chip: bit 0 - chip #1, bit 1 - chip #2, etc. When changing the mask, the settings are not automatically transferred to the device, but the new mask is taken into account when changing other properties. So, if any property is assigned the same value, then if the mask has not been changed, then the settings will not be transferred to the device, and if it was changed, then they will be transferred to the device.

Vca1EqualsVca2

The **Vca1EqualsVca2** property of the **AFE5818** class determines whether the settings for VCA #1 and VCA #2 are the same. If the property value is **true**, then the settings for VCA #2 are taken from the **Vca1** property of the **AFE5818** class, if **false**, then from **Vca2**.

AutoUpdate

The **AutoUpdate** property of the **AFE5818** class enables or disables automatic transmission of settings to the device when the properties of the **AFE5818** class are changed. If the property value is **true**, then when writing a new value to any of the properties of the **AFE5818** class, the updated settings are automatically transferred to the device. If the property value is **false**, then you can assign new values to several properties of the **AFE5818** class, and then call the Configure method, which will transfer the settings to the device and restore the **AutoUpdate** property to **true**.

Vca1, Vca2

The **Vca1** and **Vca2** properties of the **AFE5818** class are references to the **AFE5818Vca** class. This class is described below. Changing the properties of the **AFE5818Vca** class will automatically call the Configure method of the **AFE5818** class if the **AutoUpdate** property of the **AFE5818** class is **true**.

Table 6: Properties and Methods of AFE5818Vca class

Name	Type	Description
HpfCutoffDivided	Boolean	If true, the cutoff frequency of the LNA block's high-pass filter is reduced by three times.
LowNoiseMode	Boolean	Flag of VCA Low Noise Mode Enabled
PgaHpfDisabled	Boolean	Flag of PGA block high pass filter disabled
LnaHpfDisabled	Boolean	Flag of LNA High Pass Filter Disabled
PgaClampEnabled	Boolean	Flag of PGA voltage limiter enabled
F5MHzLpfEnabled	Boolean	Flag of Low pass filter Enabled
TgcAttEnabled	Boolean	Flag of Connecting the attenuator in the TGC block
PowerMode	Enum	VCA power consumption mode
HpfCutoffFreq	Enum	High Pass Filter Cutoff Frequency
LpfCutoffFreq	Enum	Low Pass Filter Cutoff Frequency
TgcAttenuation	Enum	Attenuator gain in TGC unit
LnaGlobalGain	Enum	LNA block gain
PgaGain	Enum	PGA block gain

HpfCutoffDivided

The **HpfCutoffDivided** property of the **AFE5818Vc** class enables (**true**) or disables (**false**) the LNA block's high-pass filter cutoff frequency to be reduced threefold.

LowNoiseMode

The **LowNoiseMode** property of the **AFE5818Vc** class enables (**true**) or disables (**false**) the VCA low noise mode for high impedance sensors.

PgaHpfDisabled

The **PgaHpfDisabled** property of the **AFE5818Vc** class enables (**false**) or disables (**true**) the high-pass filter of the PGA.

LnaHpfDisabled

The **LnaHpfDisabled** property of the **AFE5818Vc** class enables (**false**) or disables (**true**) the high pass filter of the LNA block.

PgaClampEnabled

The **PgaClampEnabled** property of the **AFE5818Vc** class enables (**true**) or disables (**false**) the voltage limiter in the PGA.

F5MHzLpfEnabled

The **F5MHzLpfEnabled** property of the **AFE5818Vc** class enables (**true**) or disables (**false**) a first-order low-pass filter with a 5 MHz bandwidth.

TgcAttEnabled

The **TgcAttEnabled** property of the **AFE5818Vc** class enables (**true**) or disables (**false**) the attenuator in the TGC block.

PowerMode

The **PowerMode** property of the **AFE5818Vca** class determines the power consumption mode of the VCA unit, is an enumerable property and can be assigned three values: **LowNoise**, **LowPower**, **MediumPower**.

HpfCutoffFreq

The **HpfCutoffFreq** property of the **AFE5818Vca** class defines the cutoff frequency of the high pass filter, is an enumerable property and can be assigned the following values: **_50_kHz**, **_100_kHz**, **_150_kHz**, **_200_kHz**.

LpfCutoffFreq

The **LpfCutoffFreq** property of the **AFE5818Vca** class defines the cutoff frequency of the low-pass filter, is an enumerable property and can be assigned the following values: **_10_MHz**, **_15_MHz**, **_20_MHz**, **_30_MHz**, **_35_MHz**, **_50_MHz**.

TgcAttenuation

The **TgcAttenuation** property of the **AFE5818Vca** class sets the gain of the attenuator in the TGC block, is an enumerable property and can be assigned the following values: **_0_dB**, **_6_dB**, **_12_dB**, **_18_dB**, **_24_dB**, **_30_dB**, **_36_dB.**

LnaGlobalGain

The **LnaGlobalGain** property of the **AFE5818Vca** class defines the gain of the LNA block, is an enumerable property and can be assigned the following values: **_12_dB**, **_18_dB**, **_24_dB**.

PgaGain

The **PgaGain** property of the **AFE5818Vca** class defines the gain of the PGA block, is an enumerable property and can be assigned the following values: **_24_dB**, **_30_dB**.

Class AFE5832

Table 7: AFE5832 Class Properties and Methods

Name	Type	Description
Configure	method	Transfers settings from class properties to a device via the system bus
ConfiguredDevicesMask	Unsigned, 32 bit	Device mask for configuration
ConfiguredAdcMask	Unsigned, 32 bit	Mask of ADC chips for configuration
EnableAttenuatorHpf	Boolean	Sign of connecting the high-pass filter of the attenuator
AttenuatorHpfCorner	Enum	Attenuator high pass filter slope
OddEqualEven	Boolean	If true, then the settings for Even die are taken from the Odd property, if false, then from Even
Odd, Even	Class AFE5832Die	References to the AFE5832Die class with settings for Odd die and for Even die
AutoUpdate	Boolean	Sign of automatic transmission of settings to the device when class properties change

Configure

The **Configure** method of the **AFE5832** class transfers the settings calculated from the properties of the **AFE5832** class to the device via the system bus, and also sets the **AutoUpdate** property to **true**.

ConfiguredDevicesMask

The **ConfiguredDevicesMask** property of the **AFE5832** class sets the device mask for configuration. Each bit of the mask corresponds to one device: bit 0 - to a device with Id = 0, bit 1 - to a device with Id = 1, etc., where Id is an identifier of a device on the system bus. When changing the mask, the settings are not automatically transferred to the device, but the new mask is taken into account when changing other properties. So if any property is assigned the same value, then if the mask has not been changed, then the settings will not be transferred to the device, and if they were changed, they will be. The number of connected devices can be read from the **DevicesCount** property of the Device Manager.

ConfiguredAdcMask

The **ConfiguredAdcMask** property of the **AFE5832** class sets the mask of the ADC chips for configuration. Each bit of the mask corresponds to one chip: bit 0 - chip # 1, bit 1 - chip # 2, etc. When changing the mask, the settings are not automatically transferred to the device, but the new mask is taken into account when changing other properties. So if any property is assigned the same value, then if the mask has not been changed, then the settings will not be transferred to the device, and if it was changed, they will be.

EnableAttenuatorHpf

The **EnableAttenuatorHpf** property of the **AFE5832** class enables (**true**) or disables (**false**) the high-pass filter of the attenuator block.

AttenuatorHpfCorner

The **AttenuatorHpfCorner** property of the **AFE5832** class determines the slope of the high-pass filter of the attenuator, is an enumerable property and can be assigned the following values: **C2**, **C3**, **C4**, **C5**, **C6**, **C7**, **C8**, **C9**, **C10**.

OddEqualEven

The **OddEqualEven** property of the **AFE5832** class determines whether the Even die and Odd die settings are the same. If the property value is **true**, then the settings for Even die are taken from the **Odd** property of the **AFE5832** class, if **false**, then from **Even**.

AutoUpdate

The **AutoUpdate** property of the **AFE5832** class enables or disables automatic transmission of settings to the device when the properties of the **AFE5832** class are changed. If the property value is **true**, then when writing a new value to any of the properties of the **AFE5832** class, the

updated settings are automatically transferred to the device. If the property value is **false**, then you can assign new values to several properties of the **AFE5832** class, and then call the **Configure** method, which will transfer the settings to the device and restore the **AutoUpdate** property to **true**.

Odd, Even

The **Odd** and **Even** properties of the **AFE5832** class are references to the **AFE5832Die** class. This class is described below. When changing the properties of the **AFE5832Die** class, the **Configure** method of the **AFE5832** class will be automatically called if the **AutoUpdate** property of the **AFE5832** class is **true**.

Table 8: AFE5832Die Class Properties and Methods

Name	Type	Description
LpfCutoffFreq	Enum	LNA low pass filter cutoff frequency
HpfCutoffFreq	Enum	LNA High Pass Filter Cutoff Frequency
DtgcGain	Double	Digital TGC gain, dB
EnableLnaHpf	Boolean	Sign of connecting the high-pass filter of the LNA unit
LowPowerMode	Boolean	Flag of VCA unit low power consumption mode activation
EnableDtgcAttenuator	Boolean	Flag of connecting the attenuator in the digital TGC block

LpfCutoffFreq

The **LpfCutoffFreq** property of the **AFE5832Die** class defines the cutoff frequency of the LNA block low pass filter, is an enumerable property and can be assigned the following values: **_10_MHz**, **_15_MHz**, **_20_MHz**, **_30_MHz**. If the low power mode is enabled (the **LowPowerMode** property of the **AFE5832Die** class is **true**), then the frequency values must be divided by two.

HpfCutoffFreq

The **HpfCutoffFreq** property of the **AFE5832Die** class defines the cutoff frequency of the LNA block high pass filter, is an enumerable property and can be assigned the following values: **_75_kHz**, **_150_kH**.

DtgcGain

The **DtgcGain** property of the **AFE5832Die** class defines the digital TGC gain in decibels.

EnableLnaHpf

The **EnableLnaHpf** property of the **AFE5832Die** class enables (**true**) or disables (**false**) the high-pass filter in the LNA block.

LowPowerMode

The **LowPowerMode** property of the **AFE5832Die** class enables (**true**) or disables (**false**) the low power mode of the VCA.

EnableDtgcAttenuator

The **EnableDtgcAttenuator** property of the **AFE5832Die** class enables (**true**) or disables (**false**) the attenuator in the digital TGC block.

Class AFE5832LP

Table 9 AFE5832LP Class Properties and Methods

Name	Type	Description
Configure	Method	Transfers settings from class properties to a device via the system bus
ConfiguredDevicesMask	Unsigned, 32 bit	Device mask for configuration
ConfiguredAdcMask	Unsigned, 32 bit	Mask of ADC chips for configuration
HpfCornerFreq	Enum	Attenuator high pass filter slope
LpfCutoffFreqs	Enum	Low pass filter cutoff frequency
PgaGainOddEqualEven	Enum	PGA gain
LnaGainOdd, Even	Enum	LNA gain
LowPowerMode	Boolean	Flag of low power consumption mode
LowLatencyEnable	Boolean	Flag of the mode with low signal delay and disabled digital postprocessing
Attenuator	Double	Digitally controlled attenuator attenuation range 0 to 36 dB
AutoUpdate	Boolean	Flag for automatic transmission of settings to the device when class properties are changed

Configure

The **Configure** method of the **AFE5832LP** class transfers the settings calculated from the properties of the **AFE5832LP** class to the device via the system bus, and also sets the **AutoUpdate** property to **true**.

ConfiguredDevicesMask

The **ConfiguredDevicesMask** property of the **AFE5832LP** class sets the device mask for configuration. Each bit of the mask corresponds to one device: bit 0 - to a device with Id = 0, bit 1 - to a device with Id = 1, etc., where Id is an identifier of a device on the system bus. When changing the mask, the settings are not automatically transferred to the device, but the new mask is taken into account when changing other properties. So if any property is assigned the same value, then if the mask has not been changed, then the settings will not be transferred to

the device, and if they were changed, they will be. The number of connected devices can be read from the **DevicesCount** property of the Device Manager.

ConfiguredAdcMask

The **ConfiguredAdcMask** property of the **AFE5832LP** class sets the mask of the ADC chips for configuration. Each bit of the mask corresponds to one chip: bit 0 - chip # 1, bit 1 - chip # 2, etc. When changing the mask, the settings are not automatically transferred to the device, but the new mask is taken into account when changing other properties. So, if any property is assigned the same value, then if the mask has not been changed, then the settings will not be transferred to the device, and if it was changed, they will be.

HpfCornerFreq

The **HpfCornerFreq** property of the **AFE5832LP** class determines the steepness of the high pass filter, is an enumerated property, and can be assigned values: **_100_kHz, _110_kHz, _120_kHz, _130_kHz, _140_kHz, _150_kHz, _160_kHz, _170_kHz, _20_kHz, _30_kHz, _40_kHz, _50_kHz, _60_kHz, _70_kHz, _80_kHz, _90_kHz, _270_kHz, _280_kH, _290_kHz, _300_kHz, _310_kHz, _180_kHz, _190_kHz, _200_kHz, _210_kHz, _220_kHz, _230_kHz, _240_kHz.**

LpfCutoffFreq

The **LpfCutoffFreq** property of the **AFE5832LP** class determines the cutoff frequency of the low-pass filter of the block, is an enumerated property and can be assigned values: **_10_MHz**, **_15_MHz**, **_20_MHz**, **_25_MHz**. If low power mode is enabled (the **LowPowerMode** property of the **AFE5832LP** class is **true**), then the **_25_MHz** value will correspond to a frequency of 20 MHz, and the remaining values will correspond to the frequencies specified in them.

PgaGain

The **PgaGain** property of the **AFE5832LP** class determines the gain of the PGA block, is an enumerated property, and can be assigned values: **_21_dB, _24_dB, _27_dB**.

LnaGain

The **LnaGain** property of the **AFE5832LP** class determines the gain of the LNA block, is an enumerated property, and can be assigned values: **_15_dB, _18_dB, _21_dB**.

LowPowerMode

The **LowPowerMode** property of the **AFE5832LP** class enables (**true**) or disables (**false**) the low power mode.

LowLatencyEnable

The **LowLatencyEnable** property of the **AFE5832LP** class enables (**true**) or disables (**false**) the low latency mode with digital processing disabled.

Attenuator

The **Attenuator** property of the **AFE5832LP** class sets the attenuation factor of the digital attenuator in decibels in the range from 0 to 36 in 0.125 dB steps.

AutoUpdate

The **AutoUpdate** property of the **AFE5832LP** class enables or disables automatic transmission of settings to the device when the properties of th **AFE5832LP** class are changed. If the property value is **true**, then when writing a new value to any of the properties of the **AFE5832LP** class, the updated settings are automatically transferred to the device. If the property value is **false**, then you can assign new values to several properties of the **AFE5832LP** class, and then call the **Configure** method, which will transfer the settings to the device and restore the **AutoUpdate** property to **true**.

Data file format

ADC data is saved in a binary RAW file. The file consists of a file header (

Table 10) and N ADC data frames. The number of data frames is written in the file header. Each frame also contains a header and data (Table 11). Each frame corresponds to one device, the frames are written to the file strictly sequentially from the device with a lower serial number to the device with a higher serial number. The serial number of the device is determined by its position in the chain of devices connected by HDMI cable. The master has sequence number 0 and the last sequence number has the last connected device. To determine which device a particular frame belongs to, analyze the Boards Mask field in the file header.

The data in each frame is arranged sequentially - first the 1st sample of channel # 1, then the 1st sample of channel # 2 and so on until the last channel, then the 2nd sample of channel # 1, then the 2nd sample of channel # 2 and so on until the last channel. Then the sequence is repeated until the last count of the last channel. The file header contains the total number of channels recorded in the file, and the same number of data samples per ADC channel for all data frames. The frame header indicates the number of channels for the corresponding device. The number of channels is determined by the number of allowed ADCs and the number of channels per ADC. In order to determine which channel belongs to which ADC, you should analyze the mask of allowed ADCs **ADC Mask** for the current frame. Channels are always arranged in ascending order by ADC numbers.

Table 10 Data file header format

Field	Type	Description
Format version	Double	File version
Number of frames	Integer, 32 bit	File format version
Header length	Integer, 32 bit	File header size in bytes
Frame length	Integer, 32 bit	Data frame size in bytes
Sample rate	Integer, 32 bit	The frequency of data sampling in Hertz
Number of channels	Integer, 32 bit	Number of ADC channels in the file
Number of samples	Integer, 32 bit	Number of data counts on the ADC channel in each frame
Number of boards	Integer,32 bit	The number of devices, data from which are recorded in the file
Boards mask	Unsigned, 32 bit	The device mask is in order from the master (bit #0) to the last slave (bit #31). If data from the device with the sequence number N, starting from the master device with the number 0, are written to the file, then the N bit is 1, otherwise 0

Table 11 Data Frame Format

Field	Type	Description
Number of channels	Integer, 32 bit	Number of ADC channels in a frame
Number of samples	Integer, 32 bit	Number of samples per channel
Sample rate	Integer, 32 bit	Data sampling rate in Hz
Trigger source	Integer, 32 bit	Frame data trigger input mask. Bit 0 of the mask corresponds to input #1, bit 1 of the mask – to input #2, and so on.
Trigger time	Double	Timing of the frame data trigger signal in milliseconds
Frame number	Unsigned, 32 bit	Numeric frame label
ADC Mask	Unsigned, 32 bit	Mask of allowed ADC chips. Each bit of the mask corresponds to one ADC chip, bit 0 to chip #1, bit 1 to chip #2, and so on. If the bit value is 1, then the ADC is allowed to collect data, otherwise it is prohibited.
ADC data	Integer Array, 16 bit	ADC data array, data are arranged sequentially by channels: first, all channels for counting #1, then all channels for counting #2, etc.