# PhotoSound SDK 1.1.2

Руководство пользователя

## Оглавление

## Введение

Пакет разработчика (SDK) предназначен для разработки приложений для устройств PhotoSound в средах MATLAB, LabVIEW и Visual Studio C#. Пакет состоит из нескольких программных уровней, пример которых для устройства ADC256 Legion показан на рисунке (Рисунок 1). Первый уровень – системный и в него входят драйверы, обеспечивающие обмен данными с устройствами по системной шине. Второй уровень – промежуточный и он состоит из 32 или 64 разрядных библиотек (LIB или DLL), реализующих интерфейс взаимодействия драйверов и программного обеспечения верхнего уровня. Компоненты первого уровня и в некоторых случаях второго уровня копируются на ПК пользователя в процессе установки драйверов и хранятся отдельно от остальных компонентов SDK в папке Windows. Третий уровень – функциональный, это 32 или 64 разрядные DLL, обеспечивающие функционирование устройств – загрузку прошивок (firmware), инициализацию аппаратной части, реализацию протокола передачи команд управления и данных. Четвёртый уровень – библиотечный, это .NET сборка, так же имеющая расширение DLL, в которой содержатся классы для сбора и сохранения данных, конфигурирования устройств, хранения настроек конфигураций, сохранения настроек в INI файлах и чтения настроек из INI файлов. .NET сборка этого уровня уже может использоваться в программных средах MATLAB, LabVIEW и Visual Studio C#. Пятый уровень – прикладной, он так же представлен .NET сборкой с расширением DLL, которая содержит графические элементы управления, посредством которых конечный пользователь приложения работает с устройствами. Эти графические элементы можно помещать на форму приложения Windows Forms C# или LabVIEW front panel через .NET контейнер. Шестой уровень – пользовательский, это самостоятельное приложение с расширением EXE, так же являющееся .NET сборкой. С помощью этого приложения пользователь может не только выполнять основные операции с устройством, но и использовать его как диалоговое окно в составе более сложного приложения, написанного в любой из вышеперечисленных программных сред.


*Рисунок 1 - Программные уровни SDK*

В SDK имеется два набора файлов папках sdk\x86 и sdk\x64. Наборы различаются разрядностью машинного кода динамических библиотек 3-го уровня. Для сборки приложений используется только один из наборов: x86 – для 32 разрядных приложений и x64 – для 64 разрядных. Компоненты для создания пользовательских приложений находятся в подпапке PhotoSoundSDK. Перечень и описание файлов, находящихся в этой подпапке представлен в таблице ниже.

| Имя файла/папки | Описание |
|---|---|
| PhotoSoundClasses.dll | Главная .NET сборка с библиотекой классов для работы с устройствами |
| PhotoSoundControls.dll | .NET сборка с графическими элементами управления |
| PhotoSoundDAQ.exe | .NET сборка с приложением для выполнения основных операций с устройствами |
| PhotoSoundDAQ.exe.config | Конфигурационный файл для Photo-SoundDAQ.exe |
| PhotoSoundLibs | Папка с программными компонентами 3-го уровня, прошивками и конфигурационными INI файлами с настройками |
| PhotoSoundLibs\Device\PhotoSoundDevice.dll | Функциональная динамическая библиотека для работы с устройствами АЦП |
| PhotoSoundLibs\Device\PhotoSoundDevice.img | Файл прошивки контроллера Cypress FX3 шины USB 3.0 |
| PhotoSoundLibs\Device\AFE5832.ini | Конфигурационный файл со значениями регистров АЦП Texas Instruments AFE5832, загружаемыми при первом запуске программного обеспечения |
| PhotoSoundLibs\Device\AFE5818.ini | Конфигурационный файл со значениями регистров АЦП Texas Instruments AFE5818, загружаемыми при первом запуске программного обеспечения |
| PhotoSoundLibs\Device\adc*.bin | Файл прошивки ПЛИС. Полное имя файла определяется типом устройства и ревизией его печатной платы. Запустите SetRevision2.1\GetFirmwareName.exe для получения имени прошивки |
| Config | Папка с конфигурационными INI файлами |

| | |
|---|---|
| Config\Default.ini | Конфигурационный файл с настройками устройств по умолчанию. Если файл не существует, то он создаётся автоматически при первом запуске программного обеспечения |
| Config*.ini | Альтернативные конфигурационные файлы с настройками устройств |
| Data | Папка по умолчанию для сохранения данных, захваченных с АЦП |
| Maps | Папка для хранения файлов с картами сенсоров. Карта представляет собой столбец с номерами сенсоров в порядке от первого канала первого АЦП до последнего канала последнего АЦП на плате |
| Maps*.map | Файлы с картой сенсоров. Для каждого типа устройства существует свой файл с картой сенсоров |

Помимо программных компонентов PhotoSoundSDK в SDK входят:

- папка doc – содержит это руководство пользователя и Excel файлы – калькуляторы конфигурационных INI файлов для АЦП: AFE5832.xlsm и AFE5818.xlsm;
- папка examples – содержит примеры кода для программных сред MATLAB, LabVIEW и Visual Studio C#.

# Работа с библиотекой классов PhotoSoundClasses.dll

## Начало работы в Matlab

Для начала работы с библиотекой классов необходимо загрузить сборку по команде **NET.addAssembly(asm_path)**, где asm_path – полный путь к файлу PhotoSoundClasses.dll. Далее можно получить список классов в библиотеке по команде **disp(asm.Classes)**. Пользователь может использовать экземпляры (instances) этих классов для управления устройствами PhotoSound и сбора данных, но создавать сам может только экземпляры классов **DeviceManager** и **Settings**. Экземпляры остальных классов создаются автоматически при подключении к устройству и доступны как свойства экземпляра класса **DeviceManager**, далее просто диспетчера устройств. Итак создаём диспетчер устройств по команде **dev = PhotoSoundClasses.DeviceManager** и запускаем подключение к устройству по команде **dev.Connect**. Поскольку подключение к устройству занимает некоторое время, особенно при первом подключении после включения питания устройства, то далее можно заняться другими задачами, а после перейти к циклу ожидания завершения подключения.

Ожидать подключения следует до тех пор, пока значение одного из свойств **Connected** или **ConnectFailure** диспетчера устройств не станет равным 1.

В процессе написания программного кода часто требуется знать список методов и свойств того или иного класса, а также событий, которые он может генерировать. Для этого в среде Matlab есть команды **methods**, **events** и **properties**. Так, если выполнить **methods(dev)**, то получим

```
>> methods(dev)

Methods for class PhotoSoundClasses.DeviceManager:

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

Методы **Equals**, **GetType**, **GetHashCode** и **ToString** являются стандартными для всех NET классов. Описание остальных методов, свойств и событий можно найти в таблицах в конце раздела. Кроме того, если после имени экземпляра класса набрать . и нажать клавишу Tab, то появится список методов и свойств из которого можно выбрать нужное.

В процессе подключения к устройству и при работе с ним могут возникать различные ошибки, например, если питание устройства не включено или не подключен кабель. Для уведомления пользователя об ошибках в диспетчере устройств предусмотрено событие **OnError**. Подписавшись на это событие по команде **addlistener(dev,'OnError',@onerror)**, можно вывести на экран сообщение об ошибке в случае её возникновения. Функция-обработчик **onerror** здесь принимает два аргумента: первый – источник события (всегда диспетчер устройств), второй – ссылка на экземпляр класса **MessageEventArgs**. Класс **MessageEventArgs** имеет свойство **Message**, которое содержит описание ошибки, и свойство **Source** – ссылку на экземпляр класса, который является источником ошибки. Пример кода такой функции представлен ниже:

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

Следующим шагом после успешного подключения к устройству АЦП, как правило, является отображение данных АЦП на графике. Для этого в диспетчере устройств есть метод **GetPlotData**. По команде **num_samples = dev.GetPlotData(buffer, buffer_length, device_id, adc_num, chan_num)** в буфер данных в памяти **buffer** будут скопированы отсчёты данных канала **chan_num** для АЦП **adc_num** и устройства **device_id**. Нумерация в аргументах этого метода идёт с нуля. Метода возвращает число отсчётов **num_samples**, скопированных в буфер. Оно может быть меньше, чем число запрошенных отсчётов или длины буфера **buffer_length**, если сбор данных производится для меньшего числа отсчётов. Для выделения памяти для буфера данных в среде Matlab есть команда **NET.createArray**.

Размер буфера можно выбрать исходя из максимального числа отсчётов данных, которое можно получить с одного канала АЦП. Чтобы узнать это число достаточно прочитать значение свойства **MaxSamplesToCapture** диспетчера устройств. Если **num_samples** равно 0, то данные пока отсутствуют. Метод **GetPlotData** предназначен только для визуализации данных в целях контроля сбора данных. Для обработки данных АЦП в реальном масштабе времени или для записи этих данных в файл предназначен класс **DataLogger**.

По окончанию работы с устройством следует отключиться от устройства. Для этого нужно выполнить команду **dev.Disconnect**. При подключении к устройству из конфигурационного INI файла автоматически загружаются настройки устройства и производится его настройка. А при отключении настройки автоматически сохраняются в конфигурационном файле.

В таблице ниже приведён код скрипта simple.m из папки examples\matlab\, который состоит из перечисленных выше команд.

*Table 1 Пример скрипта Matlab для подключения к устройству, сбора и визуализации данных на графике*

```
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');
```

## Начало работы в LabVIEW

Начинать работу с библиотекой классов в LabVIEW можно сразу с создания экземпляра класса **DeviceManager**, далее просто диспетчера устройств. Для этого кликаем правой кнопкой мыши (ПКМ) на диаграмме, в окне Functions выбираем **Connectivity\.NET\Constructor Node**. Далее кликаем левой кнопкой мыши на диаграмме и в появившемся диалоговом окне делаем обзор по кнопке **Browse…** и находим PhotoSoundClasses.dll. После чего в окне появляется список классов **Objects** и конструкторов **Constructors** из которого выбираем **DeviceManager** и **DeviceManager(String appPath)** (Figure 1). В списке классов **Objects** можно увидеть много других классов. Пользователь может использовать экземпляры (instances) этих классов для управления устройствами PhotoSound и сбора данных, но создавать сам может только экземпляры классов **DeviceManager** и **Settings**. Экземпляры остальных классов создаются автоматически при подключении к устройству и доступны как свойства диспетчера устройств. После размещения конструктора диспетчера устройств на диаграмме запускаем подключение к устройству. Для этого через ПКМ добавляем на диаграмму **Connectivity\.NET\Invoke Node (.NET)**, соединяем вход ссылки с конструктором и через меню **Method** выбираем метод **Connect(Boolean autoApdate)** (Figure 2).

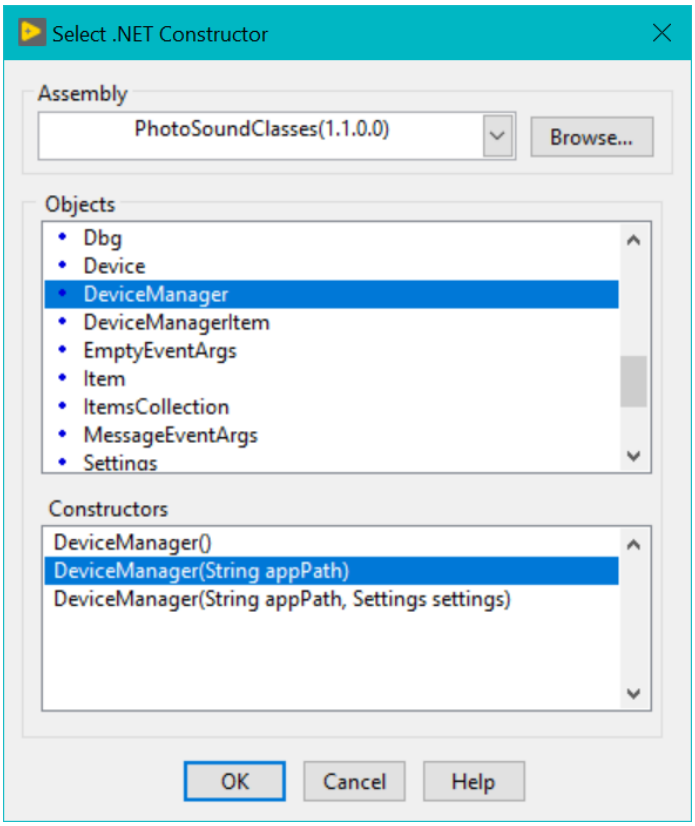


*Figure 1 Создание диспетчера устройств в Labview*

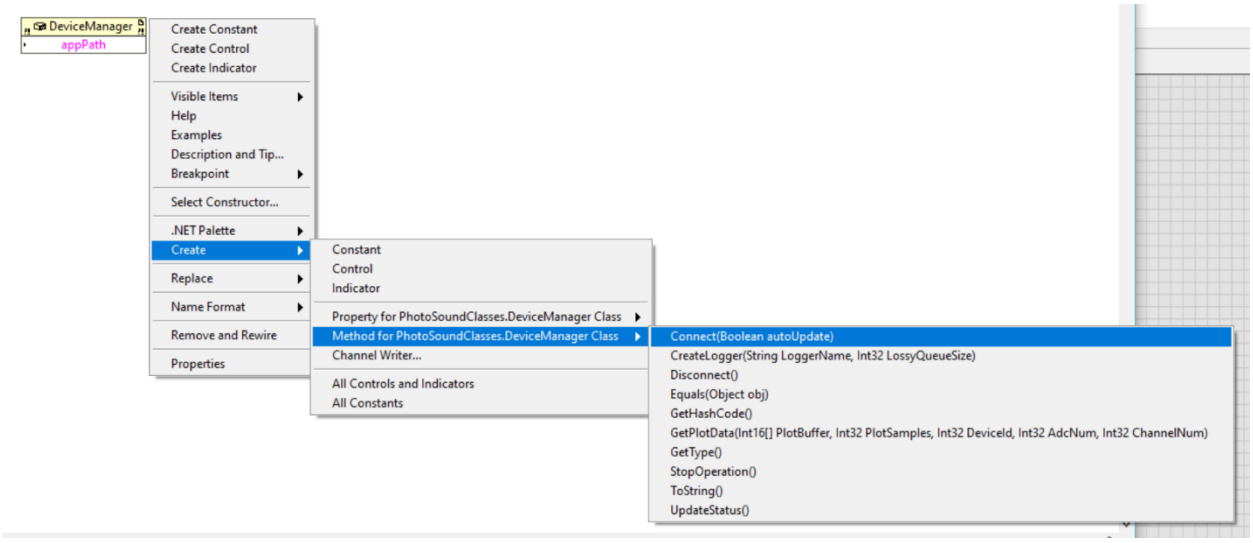


*Figure 2 Вызов метода Connect в LabVIEW*

Поскольку подключение к устройству занимает некоторое время, особенно при первом подключении после включения питания устройства, то далее можно заняться другими задачами, а после перейти к циклу ожидания завершения подключения. Ожидать подключения следует до тех пор, пока значение одного из свойств **Connected** или **ConnectFailure** диспетчера устройств не станет равным True.Для чтения свойства диспетчера устройств нужно через ПКМ добавить на диаграмму **Connectivity\.NET\Property Node (.NET)**, соединить вход ссылки с конструктором и через меню **Property** выбрать конкретное свойство, например **Connected** (Figure 3).

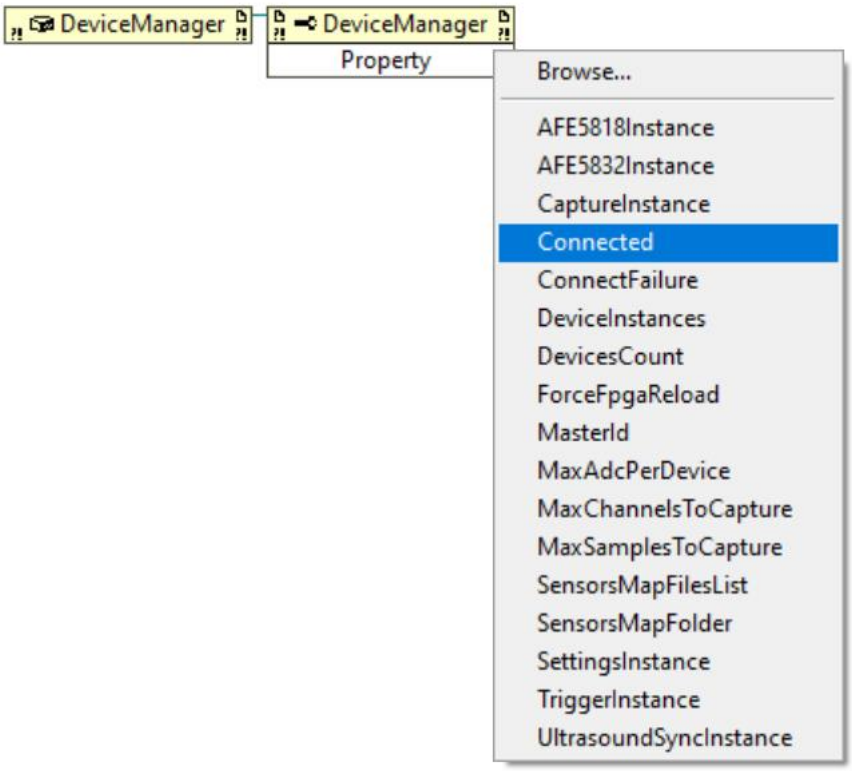


*Figure 3 Чтение свойства диспетчера устройств в Labview*

В процессе подключения к устройству и при работе с ним могут возникать различные ошибки, например, если питание устройства не включено или не подключен кабель. Для уведомления пользователя об ошибках в диспетчере устройств предусмотрено событие **OnError**. Если зарегистрировать обработчик этого события, то можно вывести на экран сообщение об ошибке в случае её возникновения. Для этого через ПКМ добавляем на диаграмму **Connectivity\.NET\Register Event Callback**, соединяем вход **Event** с конструктором диспетчера устройств и в меню **Event** выбираем **OnError** (Figure 4). Теперь через ПКМ на входе **VI Ref** выбираем **Create Callback VI**. Labview создаст новый Vi с нужным интерфейсом, на диаграмму которого можно добавить вывод диалогового окна с сообщением об ошибке (Figure 5). Диаграмма обработчика имеет вход **Event Data** – кластер с двумя полями: **sender** – источник события (всегда диспетчер устройств), **e** – ссылка на экземпляр класса **MessageEventArgs**. Класс **MessageEventArgs** имеет свойство **Message**, которое содержит описание ошибки, и свойство **Source** – ссылку на экземпляр класса, который является источником ошибки.

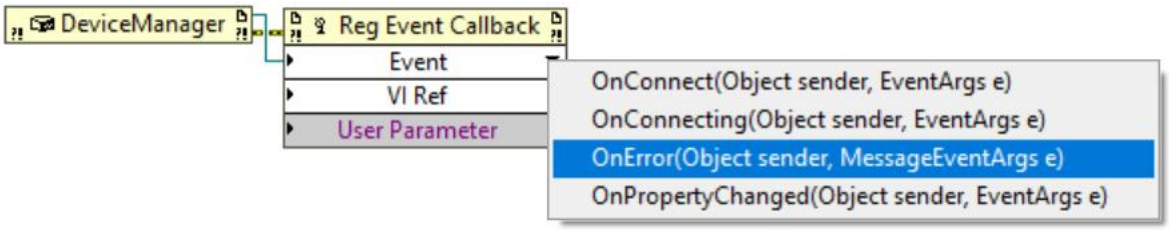


*Figure 4 Создание обработчика события OnError в Labview*

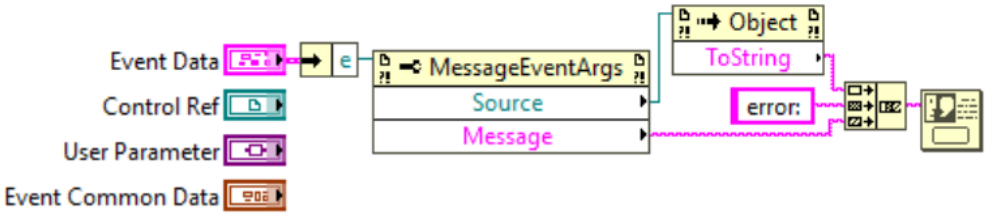


*Figure 5 Диаграмма обработчика события OnError*

Следующим шагом после успешного подключения к устройству АЦП, как правило, является отображение данных АЦП на графике. Для этого в диспетчере устройств есть метод **GetPlotData**, который копирует в буфер данных в памяти **PlotBuffer** отсчёты данных канала **ChannelNum** для АЦП **AdcNum** и устройства **DeviceId** (). Нумерация в аргументах этого метода идёт с нуля. Размер буфера можно выбрать исходя из максимального числа отсчётов данных, которое можно получить с одного канала АЦП. Чтобы узнать это число достаточно прочитать значение свойства **MaxSamplesToCapture** диспетчера устройств. На выходе **GetPlotData** метод возвращает число отсчётов, скопированных в буфер. Оно может быть меньше, чем число запрошенных отсчётов или длины буфера **PlotSamples**, если сбор данных производится для меньшего числа отсчётов. Если на выходе **GetPlotData** 0, то

данные пока отсутствуют. Метод **GetPlotData** предназначен только для визуализации данных в целях контроля сбора данных. Для обработки данных АЦП в реальном масштабе времени или для записи этих данных в файл в формате пользователя предназначен класс **DataLogger**.

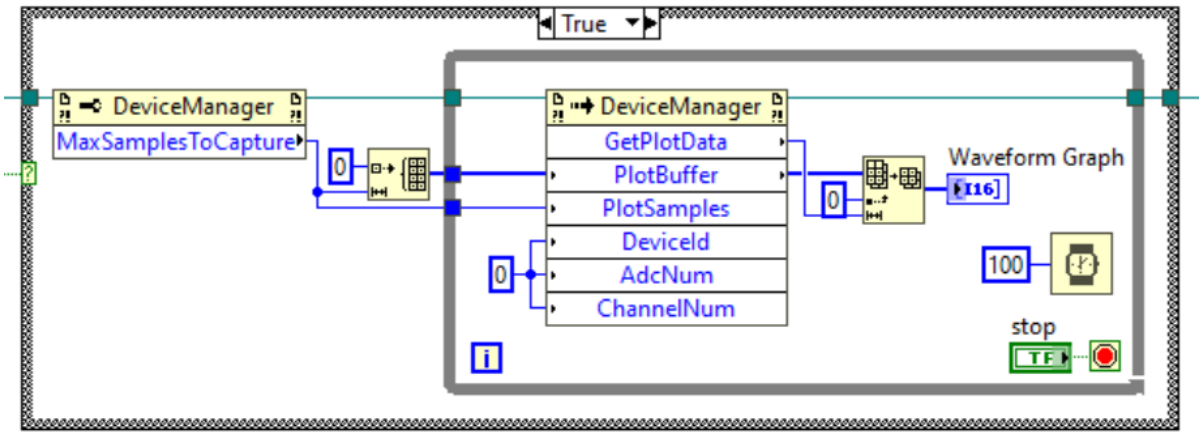


*Figure 6 Отображение данных АЦП на графике*

По окончанию работы с устройством следует отключиться от устройства. Для этого предназначен метод **Disconnect**. При подключении к устройству из конфигурационного INI файла автоматически загружаются настройки устройства и производится его настройка. А при отключении настройки автоматически сохраняются в конфигурационном файле.

В папке examples\labview находится пример simple.vi, реализующий описанные выше действия.

## Начало работы в Visual Studio C#

Для начала работы с библиотекой классов необходимо добавить ссылку на сборку PhotoSoundClasses.dll в проекте Visual Studio. Далее следует выбрать целевую платформу сборки проекта – x64 или x86 и скопировать файлы из папки x64\PhotoSoundSDK или x86\PhotoSoundSDK в выходную папку проекта. Библиотека классов содержит много классов, экземпляры (instances) которых пользователь может использовать для управления устройствами PhotoSound и сбора данных, но создавать сам может только экземпляры классов **DeviceManager** и **Settings**. Экземпляры остальных классов создаются автоматически при подключении к устройству и доступны как свойства экземпляра класса **DeviceManager**, далее просто диспетчера устройств. Итак, создаём диспетчер устройств и запускаем подключение к устройству:

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

Сразу после вызова метода **Connect** диспетчер устройств будет содержать пустые ссылки на экземпляры других классов, а свойства с информацией об устройствах будут

неверными, так как подключение к устройству занимает некоторое время, особенно при первом подключении после включения питания устройства. Для уведомления пользователя об окончании подключения в диспетчере устройств предусмотрено событие **OnConnect**. В обработчике этого события можно выполнять действия, требующие информации об устройствах или обращающиеся к экземплярам других классов библиотеки. Пример такого обработчика, запускающего таймер для обновления графика и выделяющего память для буфера данных ниже:

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

В процессе подключения к устройству и при работе с ним могут возникать различные ошибки, например, если питание устройства не включено или не подключен кабель. Для уведомления пользователя об ошибках в диспетчере устройств предусмотрено событие **OnError**. Подписавшись на это событие, можно вывести на экран сообщение об ошибке в случае её возникновения. Функция-обработчик должна иметь два аргумента: первый – ссылка на диспетчер устройств, второй – ссылка на экземпляр класса **MessageEventArgs**. Класс **MessageEventArgs** имеет свойство **Message**, которое содержит описание ошибки, и свойство **Source** – ссылку на экземпляр класса, который является источником:

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

Следующим шагом после успешного подключения к устройству АЦП, как правило, является отображение данных АЦП на графике. Для этого в диспетчере устройств есть метод

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

Метод возвращает число отсчётов данных, скопированных в буфер **PlotBuffer**. Запрашиваемое число отсчётов **PlotSamples** должно быть меньше или равно размеру буфера **PlotBuffer**. **DeviceId**, **AdcNum** и **ChannelNum** – идентификатор устройства на шине, номер АЦП и номер канала АЦП соответственно. Нумерация этих аргументов идёт с нуля. Размер буфера можно выбрать исходя из максимального числа отсчётов данных, которое можно получить с одного канала АЦП. Чтобы узнать это число достаточно прочитать значение свойства **MaxSamplesToCapture** диспетчера устройств. Если возвращаемое **GetPlotData** значение равно 0, то данные пока отсутствуют. Метод **GetPlotData** предназначен только для визуализации данных в целях контроля сбора данных. Для

обработки данных АЦП в реальном масштабе времени или для записи этих данных в файл предназначен класс **DataLogger**.

По окончанию работы с устройством следует отключиться от устройства. Для этого предназначен метод **Disconnect** диспетчера устройств. При подключении к устройству из конфигурационного INI файла автоматически загружаются настройки устройства и производится его настройка. А при отключении настройки автоматически сохраняются в конфигурационном файле.

В таблице ниже приведён код из примера проекта Simple из папки examples\visual\SdkExamples, который реализует описанные выше действия.

*Table 2 Пример программы Visual C# для подключения к устройству, сбора и визуализации данных на графике*

```
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();
        }
    }
}
```

## Управление сбором данных в Matlab

Управление сбором данных АЦП осуществляется при помощи классов **Capture**, **Trigger** и **TriggerOutput**. Экземпляры этих классов создаются не пользователем, а диспетчером устройств после успешного подключения к устройствам. Ссылки на созданные экземпляры

классов хранятся в одноименных свойствах **Capture**, **Trigger** и **TriggerOutput** диспетчера устройств (класс **DeviceManager**). До подключения устройств в этих свойствах находятся пустые ссылки (null).

Класс **Capture** позволяет изменить настройки сбора данных, такие как число отсчётов данных на канал АЦП или признак ожидания события триггера перед запуском сбора данных. Настройки из свойств класс **Capture** передаются всем подключенным устройствам одновременно. Класс **Trigger** определяет условие, по которому начинается сбор данных, например, разрешён ли запуск от внутреннего генератора или номер входа, на который поступает внешний сигнал триггера. Настройки из свойств класса **Trigger** передаются только в одно устройство, которое является ведущим. Если к ПК подключено несколько ведущих устройств, то настройки передаются только в то устройство, которое является первым на системной шине. Класс **TriggerOutput** задаёт параметры выходного сигнала триггера, например, длительность импульса и задержку. Настройки из свойств класса **TriggerOutput** также передаются только в первое ведущее устройство.

Для изменения какого-либо параметра нужно просто присвоить новое значение соответствующему свойству экземпляра класса, например **dev.Capture.WaitTrigger = true**. Это значение будет автоматически передано в устройство по системной шине, например USB, а также сохранено в памяти для последующей записи настроек в конфигурационный файл. Такое поведение хорошо подходит для управления через графический интерфейс пользователя – пользователь нажимает кнопку и сразу происходит изменение настроек. Существует и другой способ, который подходит для программного управления сбором данных, когда изменяется множество параметров одновременно. Для того, чтобы запретить автоматическую передачу настроек в устройство нужно присвоить значение **false** свойству **AutoUpdate** соответствующего класса. Далее можно назначить новые значения свойствам класса и вызвать метод **Configure** этого класса. Метод **Configure** передаёт настройки в устройство и устанавливает обратно значение **true** для свойства **AutoUpdate**.

Пример изменения свойств класса **Capture**:

```
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;
```

Пример изменения свойств класса **Trigger**:

```
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;
```

Пример изменения свойств класса **TriggerOutput**:

```
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;
```

Каждое свойство класса имеет некоторый диапазон допустимых значений. При присвоении свойству значения производится его проверка и свойство изменяется, только если новое значение находится в этом диапазоне. Поэтому при создании графического интерфейса пользователя следует прочитать свойство сразу после присвоения и обновить соответствующий элемент управления прочитанным значением. Так пользователь сможет увидеть, что введённое им значение неверно и оно не было сохранено и не было передано в устройство.

Помимо свойств с настройками, класс **Trigger** содержит метод **UpdateInputFrequencies**. Этот метод считывает текущие значения частотомеров, подключенных ко входам триггера. После вызова методы необходимо ожидать события **OnUpdateInputFrequencies**, а затем можно считать значения частот из массива **Trigger . InputFrequencies**. Функция обработчик имеет два аргумента, первый – ссылку на диспетчер устройств, а второй – пустую ссылку. Пример такой функции:

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

В таблице ниже приведён код скрипта captrig.m из папки examples\matlab, который реализует описанное выше управление сбором данных. А в справочном разделе этого руководства можно найти описание всех свойств и методов классов **Capture**, **Trigger** и **TriggerOutput**.

*Table 3 Пример скрипта MATLAB для управления сбором данных*

```
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');
```

## Управление сбором данных в LabVIEW

Управление сбором данных АЦП осуществляется при помощи классов **Capture**, **Trigger** и **TriggerOutput**. Экземпляры этих классов создаются не пользователем, а диспетчером устройств после успешного подключения к устройствам. Ссылки на созданные экземпляры классов хранятся в одноименных свойствах **Capture**, **Trigger** и **TriggerOutput** диспетчера устройств (класс **DeviceManager**). До подключения устройств в этих свойствах находятся пустые ссылки (null).

Класс **Capture** позволяет изменить настройки сбора данных, такие как число отсчётов данных на канал АЦП или признак ожидания события триггера перед запуском сбора данных. Настройки из свойств класс **Capture** передаются всем подключенным устройствам одновременно. Класс **Trigger** определяет условие, по которому начинается сбор данных, например, разрешён ли запуск от внутреннего генератора или номер входа, на который поступает внешний сигнал триггера. Настройки из свойств класса **Trigger** передаются только в одно устройство, которое является ведущим. Если к ПК подключено несколько ведущих устройств, то настройки передаются только в то устройство, которое является первым на системной шине. Класс **TriggerOutput** задаёт параметры выходного сигнала триггера, например, длительность импульса и задержку. Настройки из свойств класса **TriggerOutput** также передаются только в первое ведущее устройство.

Для изменения какого-либо параметра нужно просто присвоить новое значение соответствующему свойству экземпляра класса, как показано на рисунках ниже. Это значение будет автоматически передано в устройство по системной шине, например USB, а также сохранено в памяти для последующей записи настроек в конфигурационный файл. В Labview можно не делать свой обработчик изменения значения для каждого элемента управления, а обновлять несколько свойств в общем обработчике. Поскольку пользователь может изменить значение только одного элемента управления за раз, то и в обработчике будет только одно новое значение. Внутренняя проверка в классе выявит это новое значение и настройки будут переданы в устройство по системной шине один раз.

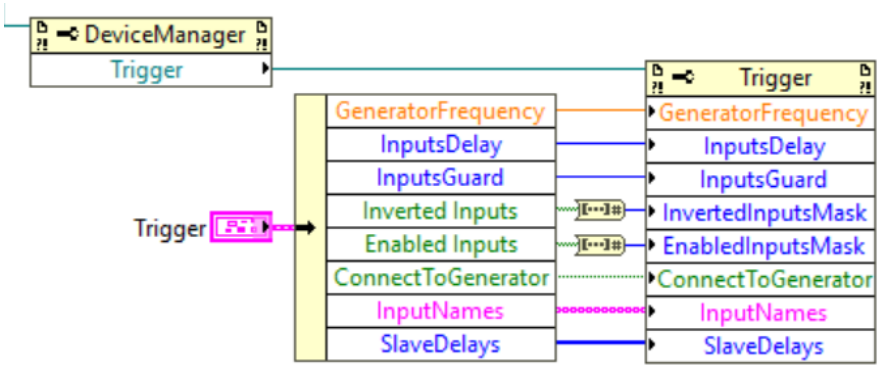


*Рисунок 2 Изменение свойств класса Trigger в LabVIEW*

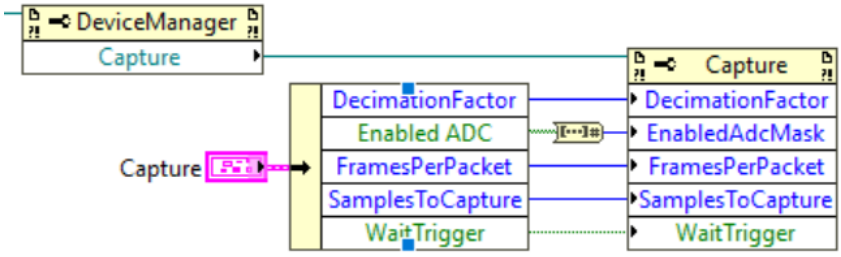


*Рисунок 3 Изменение свойств класса Capture в LabVIEW*

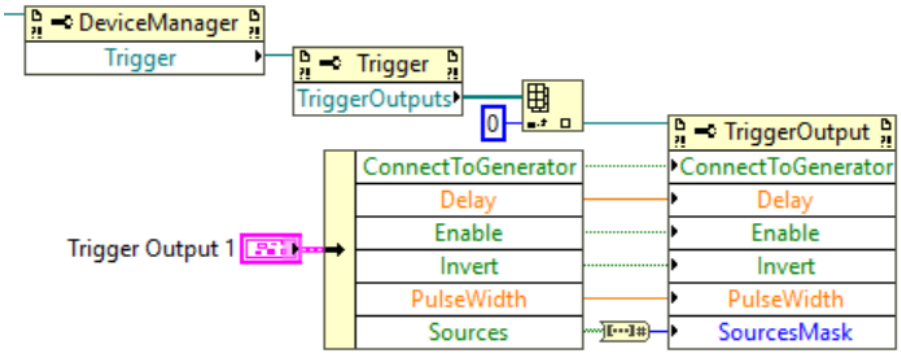


*Рисунок 4 Изменение свойств класса TriggerOutput в LabVIEW*

Каждое свойство класса имеет некоторый диапазон допустимых значений. При присвоении свойству значения производится его проверка и свойство изменяется, только если новое значение находится в этом диапазоне. Поэтому при создании графического интерфейса пользователя следует прочитать свойство сразу после присвоения и обновить соответствующий элемент управления прочитанным значением. Так пользователь сможет увидеть, что введённое им значение неверно и оно не было сохранено и не было передано в устройство. На рисунках ниже показано, как можно считывать новые значения свойств для всех трёх классов.

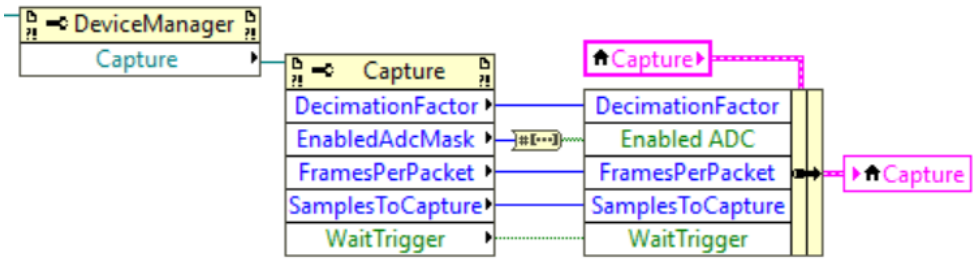


*Рисунок 5 Чтение свойств класса Capture в LabVIEW*

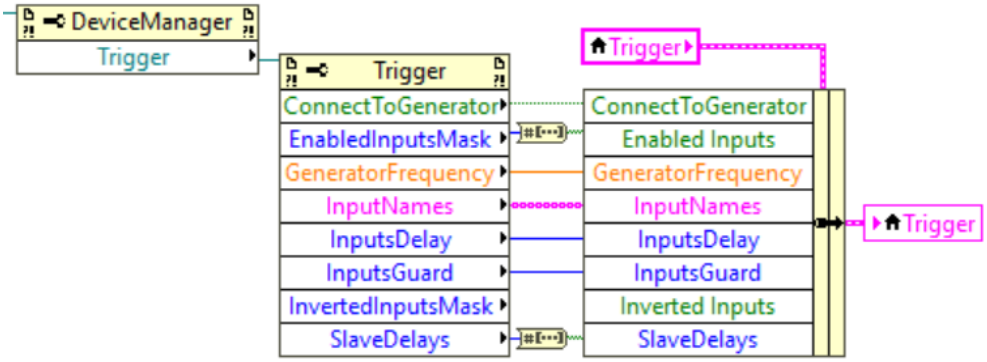


*Рисунок 6 Чтение свойств класса Trigger в LabVIEW*

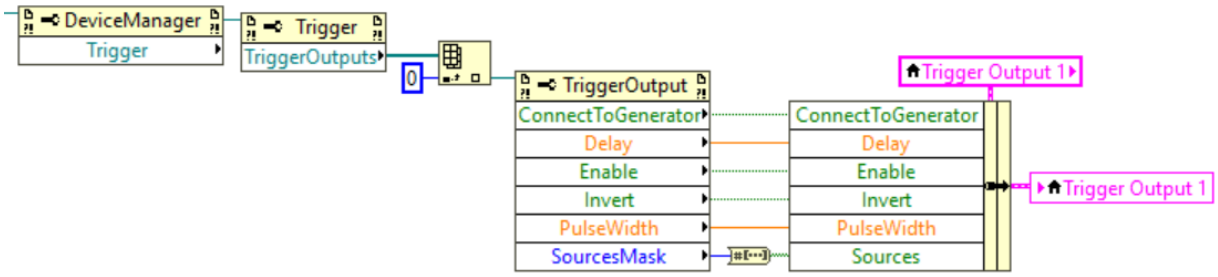


*Рисунок 7 Чтение свойств класса TriggerOutput в LabVIEW*

Помимо свойств с настройками, класс **Trigger** содержит метод **UpdateInputFrequencies** класса **Trigger**. Этот метод считывает текущие значения частотомеров, подключенных ко входам триггера. Пример получения текущего значения частоты сигнала на входах триггера представлен на рисунке ниже.

Помимо свойств с настройками, класс **Trigger** содержит метод **UpdateInputFrequencies**. Этот метод считывает текущие значения частотомеров, подключенных ко входам триггера. После вызова методы необходимо ожидать события **OnUpdateInputFrequencies**, а затем можно считать значения частот из массива **Trigger . InputFrequencies**. Функция обработчик имеет два аргумента, первый – ссылку на диспетчер устройств, а второй – пустую ссылку. Пример такой функции

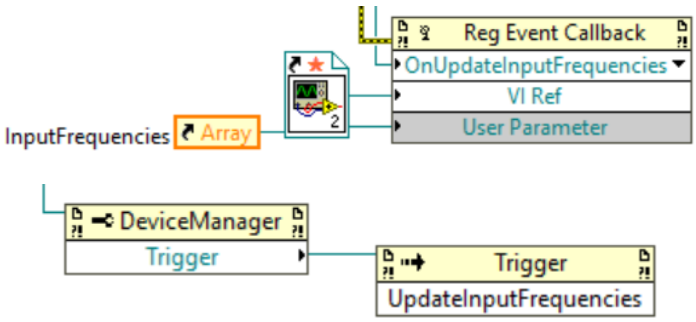


*Рисунок 8 Измерение частот на входах триггера в LabVIEW*

В папке examples\labview находится пример captrig.vi, который реализует описанное выше управление сбором данных. А в справочном разделе этого руководства можно найти описание всех свойств и методов классов **Capture**, **Trigger** и **TriggerOutput**.

### Управление сбором данных в Visual Studio C#

Управление сбором данных АЦП осуществляется при помощи классов **Capture**, **Trigger** и **TriggerOutput**. Экземпляры этих классов создаются не пользователем, а диспетчером устройств после успешного подключения к устройствам. Ссылки на созданные экземпляры классов хранятся в одноименных свойствах **Capture**, **Trigger** и **TriggerOutput** диспетчера устройств (класс **DeviceManager**). До подключения устройств в этих свойствах находятся пустые ссылки (null).

Класс **Capture** позволяет изменить настройки сбора данных, такие как число отсчётов данных на канал АЦП или признак ожидания события триггера перед запуском сбора данных. Настройки из свойств класс **Capture** передаются всем подключенным устройствам одновременно. Класс **Trigger** определяет условие, по которому начинается сбор данных, например, разрешён ли запуск от внутреннего генератора или номер входа, на который поступает внешний сигнал триггера. Настройки из свойств класса **Trigger** передаются только в одно устройство, которое является ведущим. Если к ПК подключено несколько ведущих устройств, то настройки передаются только в то устройство, которое является первым на системной шине. Класс **TriggerOutput** задаёт параметры выходного сигнала триггера, например, длительность импульса и задержку. Настройки из свойств класса **TriggerOutput** также передаются только в первое ведущее устройство.

Для изменения какого-либо параметра нужно просто присвоить новое значение соответствующему свойству экземпляра класса, например **deviceManager.Capture.WaitTrigger = true**. Это значение будет автоматически передано в устройство по системной шине, например USB, а также сохранено в памяти для последующей записи настроек в конфигурационный файл. Такое поведение хорошо подходит для управления через графический интерфейс пользователя – пользователь нажимает кнопку и сразу происходит изменение настроек. Существует и другой способ, который подходит для программного управления сбором данных, когда изменяется множество параметров

одновременно. Для того, чтобы запретить автоматическую передачу настроек в устройство нужно присвоить значение **false** свойству **AutoUpdate** соответствующего класса. Далее можно назначить новые значения свойствам класса и вызвать метод **Configure** этого класса. Метод **Configure** передаёт настройки в устройство и устанавливает обратно значение **true** для свойства **AutoUpdate**.

Пример изменения свойств класса **Capture**:

```
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();
```

Пример изменения свойств класса **Trigger**:

```
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();
```

Пример изменения свойств класса **TriggerOutput**:

```
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();
```

Каждое свойство класса имеет некоторый диапазон допустимых значений. При присвоении свойству значения производится его проверка и свойство изменяется, только если новое значение находится в этом диапазоне. Поэтому при создании графического интерфейса пользователя следует прочитать свойство сразу после присвоения и обновить соответствующий элемент управления прочитанным значением. Так пользователь сможет увидеть, что введённое им значение неверно и оно не было сохранено и не было передано в устройство.

Помимо свойств с настройками, класс **Trigger** содержит метод **UpdateInputFrequencies**. Этот метод считывает текущие значения частотомеров, подключенных ко входам триггера. После вызова методы необходимо ожидать события **OnUpdateInputFrequencies**, а затем можно считать значения частот из массива **Trigger . InputFrequencies**. Функция обработчик

имеет два аргумента, первый – ссылку на диспетчер устройств, а второй – пустую ссылку. Пример такой функции

```
    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.InputFrequen-
cies[i]:F1} Hz";
    }
```

В таблице ниже приведён код из примера проекта CapTrig из папки examples\visual\SdkExamples, который реализует описанное выше управление сбором данных. А в справочном разделе этого руководства можно найти описание всех свойств и методов классов **Capture**, **Trigger** и **TriggerOutput**.

*Table 4 Пример программы Visual C# для управления сбором данных*

```
    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.Se-
riesChartType.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, label-
Freq4 };

                for (int i = 0; i < deviceManager.Trigger.InputFrequencies.Length;
i++)
```

```
                    labels[i].Text = $"Input {i} frequency: {deviceManager.Trigger.In-
putFrequencies[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>(Plot-
Buffer, 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.MaxAdcPer-
Device) - 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.Genera-
torFrequency;
                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)udSamplesToCap-
ture.Value;
                udSamplesToCapture.Value = deviceManager.Capture.SamplesToCapture;
            }

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

            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();
            }
        }
    }
```

## Запись данных в файл в MATLAB

Запись данных в файл осуществляется при помощи класса **DataLogger**. Экземпляры этого класса (регистраторы данных) создаются пользователем при помощи метода **Create-Logger** диспетчера устройств (класс **DeviceManager**):

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

Вызывать метод **CreateLogger** следует только после подключения к устройствам, в противном случае метод возвращает пустую ссылку. Метод возвращает ссылку на созданный регистратор данных, а его аргументами являются название создаваемого регистратора и длина очереди при записи в память, что рассмотрено в разделе . Название используется для сохранения настроек регистратора в конфигурационном файле. Пользователь может создавать произвольное количество регистраторов данных. Каждый регистратор может записывать данные с одного или несколько устройств в бинарные файлы с расширением raw, причём несколько регистраторов могут получать данные с одного и того же устройства.

Регистратор начинает запись данных АЦП в файл после вызова его метода **StartLoggingToFile** с именем файла без расширения в качестве аргумента. Путь к записываемому файлу определяется свойством **DataFolder** регистратора:

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

Сразу после успешного запуска записи регистратор устанавливает значение свойства **Logging** в **true**, а по окончанию записи – в **false**. Завершение записи в файл происходит при вызове метода **StopLogging** регистратора:

```
logger.StopLogging;
```

Регистратор может автоматически заканчивать запись в файл, если будет выполнено одно из ограничительных условий, заданных перед началом записи. К этим условиям относятся – превышение размера файла в мегабайтах, превышение времени записи файла и превышение числа записанных кадров. Пример задания этих условий через свойства регистратора представлен ниже:

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

Запись данных в файл можно контролировать при помощи свойств-состояний регистратора: **Progress** – прогресс записи в процентах, **FileSize** – текущий размер файла данных в мегабайтах, **NumLoggedFrames** – текущее количество записанных кадров данных АЦП и **LoggingTime** – текущее время с начала записи файла в секундах. Свойство **Progress** показывает действительный прогресс записи только если задано одно из ограничительных условий **MaxFileSize** или **MaxLoggedFrames**, причём прогресс относится к ближайшему по выполнения условию. Время записи **LoggingTime** не используется для вычисления прогресса записи, так как контроль времени предназначен только для аварийной остановки записи в файл в результате какой-либо непредвиденной ситуации, например, из-за непоступления данных при отключении сигнала оптического триггера. Ниже приведён пример вывода на экран текущего состояния регистратора:

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

В таблице ниже представлен код скрипта filesave.m из папки examples\matlab, который реализует описанное выше управление сбором данных. Также в папке examples\matlab\RawConverter находится скрипт Raw2Mat.m для конвертации RAW файла в MAT

формат. В справочном разделе этого руководства можно найти описание всех свойств и методов класса **DataLogger**, а также описание формата файла данных.

*Table 5 Пример скрипта MATLAB для записи данных АЦП в файл*

```
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');
```

## Запись данных в файл в LabVIEW

Запись данных в файл осуществляется при помощи класса **DataLogger**. Экземпляры этого класса (регистраторы данных) создаются пользователем при помощи метода **CreateLogger** диспетчера устройств **DeviceManager** (Рисунок 9). Вызывать метод **CreateLogger** следует только после подключения к устройствам, в противном случае метод возвращает пустую ссылку. Метод возвращает ссылку на созданный регистратор данных, а его аргументами являются название создаваемого регистратора и длина очереди при записи в память, что рассмотрено в Класс DataLogger. Название используется для сохранения настроек регистратора в конфигурационном файле. Пользователь может создавать произвольное количество регистраторов данных. Каждый регистратор может записывать данные с одного или несколько устройств в бинарные файлы с расширением raw, причём несколько регистраторов могут получать данные с одного и того же устройства.

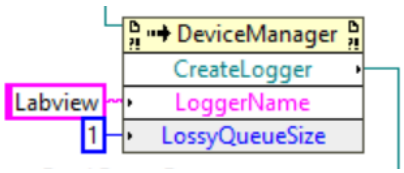


*Рисунок 9 Создание регистратора данных в LabVIEW*

Регистратор начинает запись данных АЦП в файл после вызова его метода **StartLoggingToFile** с именем файла без расширения в качестве аргумента. Путь к записываемому файлу определяется свойством **DataFolder** регистратора. Завершение записи в файл происходит при вызове метода **StopLogging** регистратора (Рисунок 10). Сразу после успешного запуска записи регистратор устанавливает значение свойства **Logging** в **true**, а по окончанию записи – в **false**.

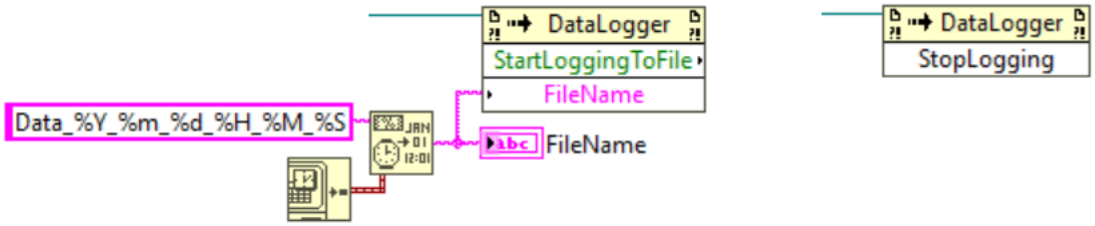


*Рисунок 10 Запуск и остановка записи данных в файл в LabVIEW*

Регистратор может автоматически заканчивать запись в файл, если будет выполнено одно из ограничительных условий, заданных перед началом записи. К этим условиям относятся – превышение размера файла в мегабайтах, превышение времени записи файла и превышение числа записанных кадров. Пример задания этих условий через свойства регистратора представлен ниже:

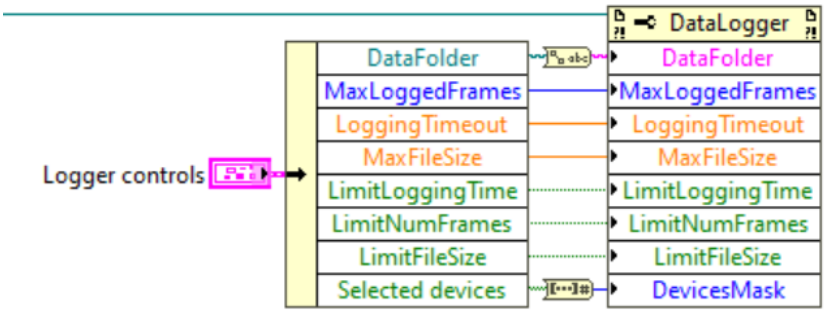


*Рисунок 11 Настройка регистратора для остановки записи по условию в LabVIEW*

Запись данных в файл можно контролировать при помощи свойств-состояний регистратора: **Progress** – прогресс записи в процентах, **FileSize** – текущий размер файла данных в мегабайтах, **NumLoggedFrames** – текущее количество записанных кадров данных АЦП и **LoggingTime** – текущее время с начала записи файла в секундах. Свойство **Progress** показывает действительный прогресс записи только если задано одно из ограничительных условий **MaxFileSize** или **MaxLoggedFrames**, причём прогресс относится к ближайшему по выполнения условию. Время записи **LoggingTime** не используется для вычисления прогресса записи, так как контроль времени предназначен только для аварийной остановки записи в файл в результате какой-либо непредвиденной ситуации, например, из-за непоступления данных при отключении сигнала оптического триггера.

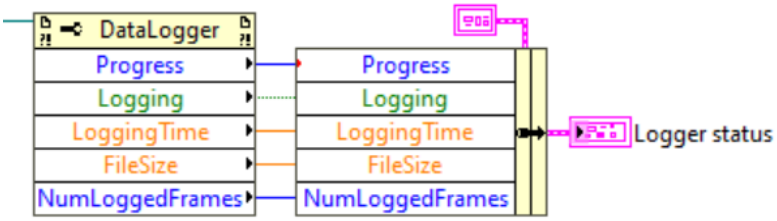


*Рисунок 12 Проверка состояния записи в файл в LabVIEW*

В папке examples\labview находится пример filesave.vi, который реализует описанную выше запись данных АЦП в файл. Также в этой папке находится пример fileplay.vi, который считывает данные из файла данных и выводит их на график. В справочном разделе этого руководства можно найти описание всех свойств и методов класса **DataLogger**, а также описание формата файла данных.

### Запись данных в файл в Visual Studio C#

Запись данных в файл осуществляется при помощи класса **DataLogger**. Экземпляры этого класса (регистраторы данных) создаются пользователем при помощи метода **Create-Logger** диспетчера устройств (класс **DeviceManager**):

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

Вызывать метод **CreateLogger** следует только после подключения к устройствам, в противном случае метод возвращает пустую ссылку. Метод возвращает ссылку на созданный регистратор данных, а его аргументами являются название создаваемого регистратора и опциональная длина очереди при записи в память, что рассмотрено в разделе . Название используется для сохранения настроек регистратора в конфигурационном файле. Пользователь может создавать произвольное количество регистраторов данных. Каждый регистратор может записывать данные с одного или несколько устройств в бинарные файлы с расширением raw, причём несколько регистраторов могут получать данные с одного и того же устройства.

Регистратор начинает запись данных АЦП в файл после вызова его метода **StartLoggingToFile** с именем файла без расширения в качестве аргумента:

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

Путь к записываемому файлу определяется свойством **DataFolder** регистратора. Завершение записи в файл происходит при вызове метода **StopLogging** регистратора:

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

При запуске и при остановке записи регистратор генерирует события **OnStartLogging** и **OnStopLogging** соответственно. События имеют стандартную сигнатуру и могут быть использованы для изменения блокировки кнопок управления.

Регистратор может автоматически заканчивать запись в файл, если будет выполнено одно из ограничительных условий, заданных перед началом записи. К этим условиям относятся – превышение размера файла в мегабайтах, превышение времени записи файла и превышение числа записанных кадров. Ниже приведён пример назначения этих свойств при помощи элементов управления главного окна:

```
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;
```

Запись данных в файл можно контролировать при помощи свойств-состояний регистратора: **Progress** – прогресс записи в процентах, **FileSize** – текущий размер файла данных в мегабайтах, **NumLoggedFrames** – текущее количество записанных кадров данных АЦП и **LoggingTime** – текущее время с начала записи файла в секундах. Свойство **Progress** показывает действительный прогресс записи только если задано одно из ограничительных условий **MaxFileSize** или **MaxLoggedFrames**, причём прогресс относится к ближайшему по выполнения условию. Время записи **LoggingTime** не используется для вычисления прогресса записи, так как контроль времени предназначен только для аварийной остановки записи в файл в результате какой-либо непредвиденной ситуации, например, из-

за непоступления данных при отключении сигнала оптического триггера. Ниже приведён отображения состояния регистратора в главном окне:

```
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}%";
```

В таблице ниже приведён код из примера проекта FileSave из папки examples\visual\SdkExamples, который реализует описанную выше запись данных в файл. Также в этой папке находится проект FilePlay, который считывает данные из файла данных и выводит их на график. В справочном разделе этого руководства можно найти описание всех свойств и методов класса **DataLogger**, а также описание формата файла данных.

В коде из таблицы ниже показан пример использования события **OnPropertyChanged** диспетчера устройств. Это событие вызывается, когда происходит изменение одного из свойств объекта – источника события. В данном случае интересуют только изменения после загрузки значений свойств из конфигурационного файла. Обработчик этого события имеет аргументы: **object sender** (всегда ссылка на диспетчер устройств) и ссылку e на экземпляр класса **EmptyEventArgs**, который имеет свойство **Source** – источник события, в данном случае **Source** должен содержать ссылку на созданный пользователем регистратор данных. Ниже показан пример как проинициализировать элементы управления главного окна значениями, считанными из конфигурационного файла после создания регистратора данных:

```
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 программы Visual C# для записи данных в файл*

```
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.SeriesChart-
Type.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, sam-
ples));
      }

      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;
    }
  }
}
```

## Настройка АЦП AFE5818 в MATLAB

Настройка АЦП **AFE5818** осуществляется при помощи классов **AFE5818** и **AFE5818Vca**. Экземпляры этих классов создаются не пользователем, а диспетчером устройств после успешного подключения к устройствам. Ссылка на созданные экземпляры хранятся в одноименном свойстве **AFE5818** диспетчера устройств (класс **DeviceManager**). До подключения устройств в этом свойстве находится пустая ссылка (null).

Для изменения какого-либо параметра нужно просто присвоить новое значение соответствующему свойству экземпляра класса, например **dev.AFE5818.LowNoiseMode = true**. Это значение будет автоматически передано в устройство по системной шине, например USB, а также сохранено в памяти для последующей записи настроек в конфигурационный файл. Некоторые параметры представлены перечисляемыми (**enum**) свойствами, например, свойство **PowerMode** класса **AFE5818Vca**. Таким свойствам можно присвоить только определенные значения, которые можно посмотреть при помощи команды **enumeration** MATLAB. Пример использования этой команды ниже:

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

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

    LowNoise
    LowPower
    MediumPower
```

Для того, чтобы в MATLAB присвоить какое-либо значение свойству **enum** необходимо сначала получить список значений в переменной, а затем использовать эту переменную с индексом нужного значения. Пример получения таких списков для **enum** свойства класса **AFE5818Vca** представлен ниже:

```
PowerMode = System.Enum.GetValues(dev.AFE5818.Vca1.PowerMode.GetType);
HpfCutoffFreq = System.Enum.GetValues(dev.AFE5818.Vca1.HpfCutoffFreq.Get-
Type);
LpfCutoffFreq = System.Enum.GetValues(dev.AFE5818.Vca1.LpfCutoffFreq.Get-
Type);
TgcAttenuation = System.Enum.GetValues(dev.AFE5818.Vca1.TgcAttenua-
tion.GetType);
LnaGlobalGain = System.Enum.GetValues(dev.AFE5818.Vca1.LnaGlobalGain.Get-
Type);
PgaGain = System.Enum.GetValues(dev.AFE5818.Vca1.PgaGain.GetType);
```

Ниже представлен пример настройки АЦП **AFE5818**. Для того, чтобы не передавать данные в устройство каждый раз при изменении одного параметра, сначала нужно присвоить значение **false** свойству **AutoUpdate**, а после изменения всех параметров нужно вызвать метод **Configure** класса **AFE5818**.

```
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;
```

Посредством свойств классов **AFE5818** и **AFE5818Vca** можно изменить только основные параметры работы АЦП. Все остальные параметры можно редактировать в файле AFE5818.xlsm, который находится в папке doc SDK. После завершения редактирования нужно нажать кнопку «Create ini file» на странице «Result», и сгенерированный AFE5818.ini

скопировать в папку PhotoSoundLibs\Device. При первом включении устройства новый файл загрузится в устройство и АЦП будет работать с новыми параметрами.

В таблице ниже приведён код скрипта afe5818.m из папки examples\matlab, который реализует описанную выше настройку АЦП. А в справочном разделе этого руководства можно найти описание всех свойств и методов классов **AFE5818** и **AFE5818Vca**.

*Table 7 Пример скрипта MATLAB для настройки АЦП AFE5818*

```
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.Hpf-
CutoffFreq.GetType);
    LpfCutoffFreq = System.Enum.GetValues(dev.AFE5818.Vca1.Lpf-
CutoffFreq.GetType);
    TgcAttenuation = System.Enum.GetValues(dev.AFE5818.Vca1.TgcAttenua-
tion.GetType);
    LnaGlobalGain = System.Enum.GetValues(dev.AFE5818.Vca1.LnaGlobal-
Gain.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');
```

## Настройка АЦП AFE5818 в LabVIEW

Настройка АЦП **AFE5818** осуществляется при помощи классов **AFE5818** и **AFE5818Vca**. Экземпляры этих классов создаются не пользователем, а диспетчером устройств после успешного подключения к устройствам. Ссылка на созданные экземпляры хранятся в одноименном свойстве **AFE5818** диспетчера устройств (класс **DeviceManager**). До подключения устройств в этом свойстве находится пустая ссылка (null).

Для изменения какого-либо параметра нужно просто присвоить новое значение соответствующему свойству экземпляра класса, как показано на рисунках ниже. Это значение будет автоматически передано в устройство по системной шине, например USB, а также сохранено в памяти для последующей записи настроек в конфигурационный файл. В LabVIEW можно не делать свой обработчик изменения значения для каждого элемента управления, а обновлять несколько свойств в общем обработчике. Поскольку пользователь может изменить значение только одного элемента управления за раз, то и в обработчике будет только одно новое значение. Внутренняя проверка в классе выявит это новое значение и настройки будут переданы в устройство по системной шине один раз.

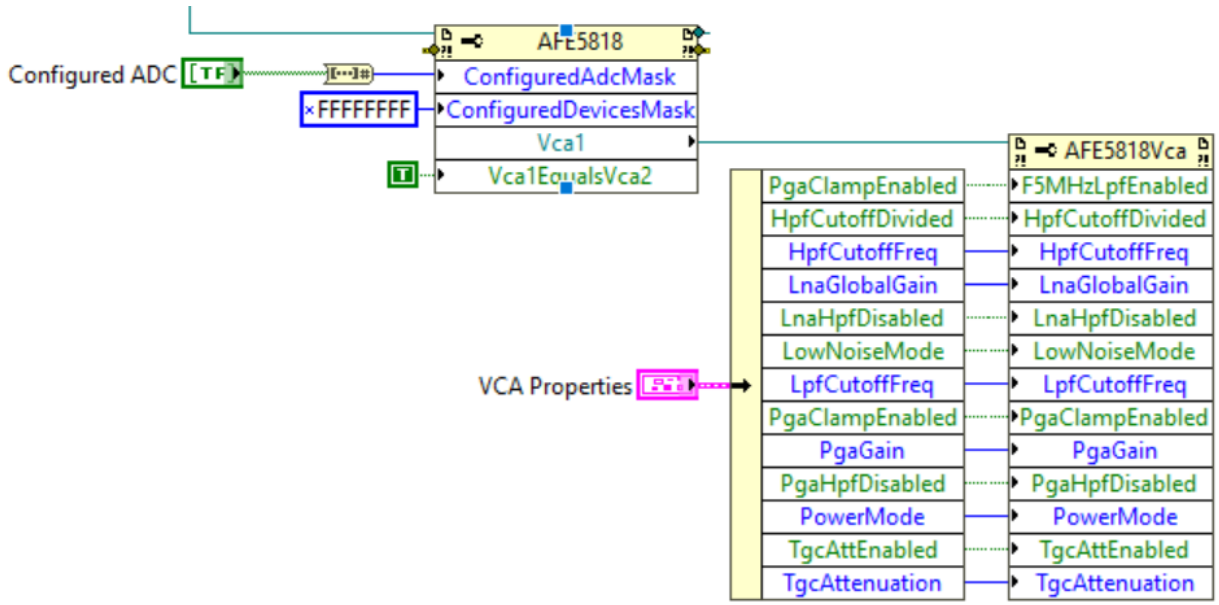


*Рисунок 13 Изменение свойств классов AFE6818 и AFE5818Vca в LabVIEW*

Каждое свойство класса имеет некоторый диапазон допустимых значений. При присвоении свойству значения производится его проверка и свойство изменяется, только если новое значение находится в этом диапазоне. Поэтому при создании графического интерфейса пользователя следует прочитать свойство сразу после присвоения и обновить соответствующий элемент управления прочитанным значением. Так пользователь сможет увидеть, что введённое им значение неверно и оно не было сохранено и не было передано в устройство. На рисунках ниже показано, как можно считывать новые значения свойств.

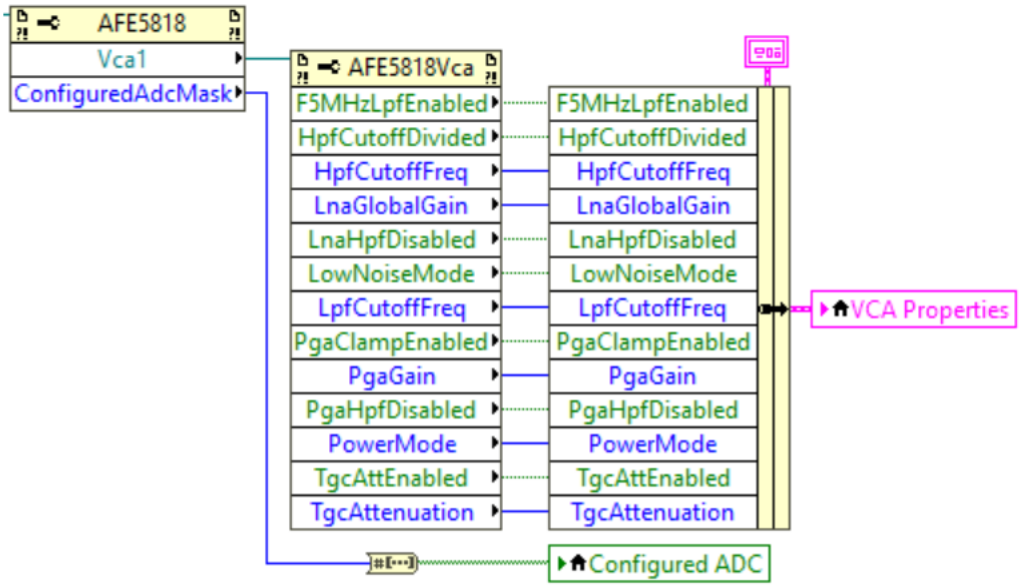


*Рисунок 14 Чтение свойств классов AFE5818 и AFE5818Vca в LabVIEW*

Посредством свойств классов **AFE5818** и **AFE5818Vca** можно изменить только основные параметры работы АЦП. Все остальные параметры можно редактировать в файле AFE5818.xlsm, который находится в папке doc SDK. После завершения редактирования нужно нажать кнопку «Create ini file» на странице «Result», и сгенерированный AFE5818.ini скопировать в папку PhotoSoundLibs\Device. При первом включении устройства новый файл загрузится в устройство и АЦП будет работать с новыми параметрами.

В папке examples\labview находится пример afe5818.vi, который реализует описанную выше настройку АЦП. А в справочном разделе этого руководства можно найти описание всех свойств и методов классов **AFE5818** и **AFE5818Vca**.

### Настройка АЦП AFE5818 в Visual Studio C#

Настройка АЦП **AFE5818** осуществляется при помощи классов **AFE5818** и **AFE5818Vca**. Экземпляры этих классов создаются не пользователем, а диспетчером устройств после успешного подключения к устройствам. Ссылка на созданные экземпляры хранятся в одноименном свойстве **AFE5818** диспетчера устройств (класс **DeviceManager**). До подключения устройств в этом свойстве находится пустая ссылка (null).

Для изменения какого-либо параметра нужно просто присвоить новое значение соответствующему свойству экземпляра класса, например **deviceManager.AFE5818.LowNoiseMode = true**. Это значение будет автоматически передано в устройство по системной шине, например USB, а также сохранено в памяти для последующей записи настроек в конфигурационный файл. Для упрощения работы со свойствами классов можно использовать элемент управления **PropertyGrid**. Если назначить его свойству **SelectedObject** ссылку на класс **AFE5818**: **propertyGrid1.SelectedObject = deviceManager.AFE5818**, то можно редактировать все свойства этого класса и класса

**AFE5818Vca** для свойств **Vca1** и **Vca2**. После присвоения пользователем нового значения какому-либо свойству, элемент управления **PropertyGrid** записывает свойство, а затем считывает его обратно и выводит считанное значение в окне. Таким образом, проверка на диапазон допустимых значений выполняется автоматически. Кроме того, для перечисляемых свойств **PropertyGrid** формирует списки, из которых пользователь может выбрать нужное значение.

Посредством свойств классов **AFE5818** и **AFE5818Vca** можно изменить только основные параметры работы АЦП. Все остальные параметры можно редактировать в файле AFE5818.xlsm, который находится в папке doc SDK. После завершения редактирования нужно нажать кнопку «Create ini file» на странице «Result», и сгенерированный AFE5818.ini скопировать в папку PhotoSoundLibs\Device. При первом включении устройства новый файл загрузится в устройство и АЦП будет работать с новыми параметрами.

В таблице ниже приведён код из примера проекта AFE5818_AFE5832 из папки examples\visual\SdkExamples, который реализует описанную выше настройку АЦП. А в справочном разделе этого руководства можно найти описание всех свойств и методов классов **AFE5818** и **AFE5818Vca**.

*Table 8 Пример программы Visual C# для управления сбором данных*

```
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;
            }
        }
    }
}
```

## Настройка АЦП AFE5832 в Matlab

Настройка АЦП **AFE5832** осуществляется при помощи классов **AFE5832** и **AFE5832Die**. Экземпляры этих классов создаются не пользователем, а диспетчером устройств после успешного подключения к устройствам. Ссылка на созданные экземпляры хранятся в одноименном свойстве **AFE5832** диспетчера устройств (класс **DeviceManager**). До подключения устройств в этом свойстве находится пустая ссылка (null).

Для изменения какого-либо параметра нужно просто присвоить новое значение соответствующему свойству экземпляра класса, например **dev.AFE5832. EnableAttenuatorHpf= true**. Это значение будет автоматически передано в устройство по системной шине, например USB, а также сохранено в памяти для последующей записи настроек в конфигурационный файл. Некоторые параметры представлены перечисляемыми (**enum**) свойствами, например, свойство **AttenuatorHpfCorner** класса **AFE5832**. Таким свойствам можно присвоить только определенные значения, которые можно посмотреть при помощи команды **enumeration** MATLAB. Пример использования этой команды ниже:

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

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

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

Для того, чтобы в Matlab присвоить какое-либо значение свойству **enum** необходимо сначала получить список значений в переменной, а затем использовать эту переменную с индексом нужного значения. Пример получения таких списков для **enum** свойств классов **AFE5832** и **AFE5832die** представлен ниже:

```
AttenuatorHpfCorner = System.Enum.GetValues(dev.AFE5832.Attenua-
torHpfCorner.GetType);
LpfCutoffFreq = System.Enum.GetValues(dev.AFE5832.Odd.LpfCutoffFreq.Get-
Type);
HpfCutoffFreq = System.Enum.GetValues(dev.AFE5832.Odd.HpfCutoffFreq.Get-
Type);
```

Ниже представлен пример настройки АЦП **AFE5832**. Для того, чтобы не передавать данные в устройство каждый раз при изменении одного параметра, сначала нужно присвоить значение **false** свойству **AutoUpdate**, а после изменения всех параметров нужно вызвать метод **Configure** класса **AFE5818**.

```
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;
```

Посредством свойств классов **AFE5832** и **AFE5832Die** можно изменить только основные параметры работы АЦП. Все остальные параметры можно редактировать в файле AFE5832.xlsm, который находится в папке doc SDK. После завершения редактирования нужно нажать кнопку «Create ini file» на странице «Result», и сгенерированный AFE5832.ini

скопировать в папку PhotoSoundLibs\Device. При первом включении устройства новый файл загрузится в устройство и АЦП будет работать с новыми параметрами.

В таблице ниже приведён код скрипта afe5832.m из папки examples\matlab, который реализует описанную выше настройку АЦП. А в справочном разделе этого руководства можно найти описание всех свойств и методов классов **AFE5832** и **AFE5832Die**.

*Table 9 Пример скрипта MATLAB для настройки АЦП AFE5832*

```
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.Attenua-
torHpfCorner.GetType);
    LpfCutoffFreq = System.Enum.GetValues(dev.AFE5832.Odd.Lpf-
CutoffFreq.GetType);
    HpfCutoffFreq = System.Enum.GetValues(dev.AFE5832.Odd.Hpf-
CutoffFreq.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');
```

## Настройка АЦП AFE5832 в LabVIEW

Настройка АЦП **AFE5832** осуществляется при помощи классов **AFE5832** и **AFE5832Die**. Экземпляры этих классов создаются не пользователем, а диспетчером устройств после успешного подключения к устройствам. Ссылка на созданные экземпляры хранятся в одноименном свойстве **AFE5832** диспетчера устройств (класс **DeviceManager**). До подключения устройств в этом свойстве находится пустая ссылка (null).

Для изменения какого-либо параметра нужно просто присвоить новое значение соответствующему свойству экземпляра класса, как показано на рисунках ниже. Это

значение будет автоматически передано в устройство по системной шине, например USB, а также сохранено в памяти для последующей записи настроек в конфигурационный файл. В LabVIEW можно не делать свой обработчик изменения значения для каждого элемента управления, а обновлять несколько свойств в общем обработчике. Поскольку пользователь может изменить значение только одного элемента управления за раз, то и в обработчике будет только одно новое значение. Внутренняя проверка в классе выявит это новое значение и настройки будут переданы в устройство по системной шине один раз.

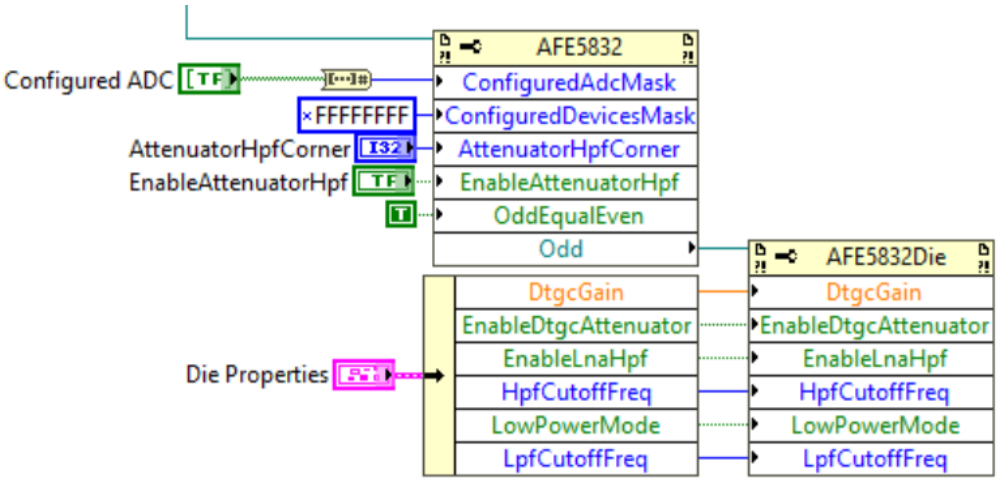


*Рисунок 15 Изменение свойств классов AFE5832 и AFE5832Die в LabVIEW*

Каждое свойство класса имеет некоторый диапазон допустимых значений. При присвоении свойству значения производится его проверка и свойство изменяется, только если новое значение находится в этом диапазоне. Поэтому при создании графического интерфейса пользователя следует прочитать свойство сразу после присвоения и обновить соответствующий элемент управления прочитанным значением. Так пользователь сможет увидеть, что введённое им значение неверно и оно не было сохранено и не было передано в устройство. На рисунках ниже показано, как можно считывать новые значения свойств.

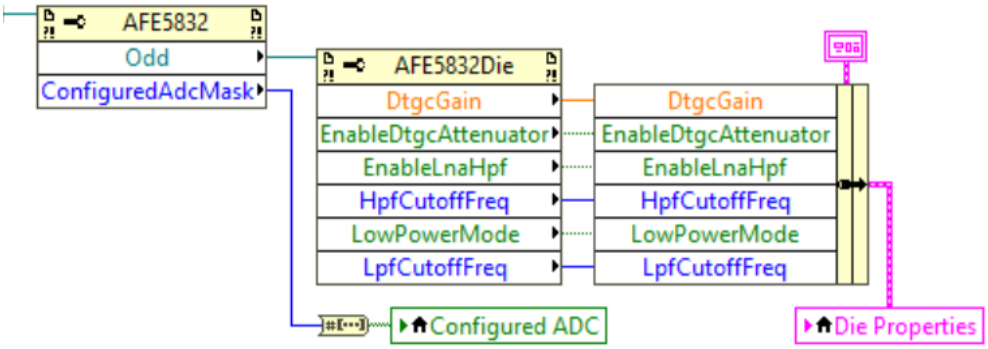


*Рисунок 16 Чтение свойств классов AFE5832 и AFE5832Die в LabVIEW*

Посредством свойств классов **AFE5832** и **AFE5832Die** можно изменить только основные параметры работы АЦП. Все остальные параметры можно редактировать в файле

AFE5832.xlsm, который находится в папке doc SDK. После завершения редактирования нужно нажать кнопку «Create ini file» на странице «Result», и сгенерированный AFE5832.ini скопировать в папку PhotoSoundLibs\Device. При первом включении устройства новый файл загрузится в устройство и АЦП будет работать с новыми параметрами.

В папке examples\labview находится пример afe5832.vi, который реализует описанную выше настройку АЦП. А в справочном разделе этого руководства можно найти описание всех свойств и методов классов **AFE5832** и **AFE5832Die**.

### Настройка АЦП AFE5832 в Visual Studio C#

Настройка АЦП **AFE5832** осуществляется при помощи классов **AFE5832** и **AFE5832Die**. Экземпляры этих классов создаются не пользователем, а диспетчером устройств после успешного подключения к устройствам. Ссылка на созданные экземпляры хранятся в одноименном свойстве **AFE5832** диспетчера устройств (класс **DeviceManager**). До подключения устройств в этом свойстве находится пустая ссылка (null).

Для изменения какого-либо параметра нужно просто присвоить новое значение соответствующему свойству экземпляра класса, например **dev.AFE5832. EnableAttenuatorHpf= true**. Это значение будет автоматически передано в устройство по системной шине, например USB, а также сохранено в памяти для последующей записи настроек в конфигурационный файл. Для упрощения работы со свойствами классов можно использовать элемент управления **PropertyGrid**. Если назначить его свойству **SelectedObject** ссылку на класс **AFE5832**: **propertyGrid1.SelectedObject = deviceManager.AFE5832**, то можно редактировать все свойства этого класса и класса **AFE5832Die** для свойств **Odd** и **Even**. После присвоения пользователем нового значения какому-либо свойству, элемент управления **PropertyGrid** записывает свойство, а затем считывает его обратно и выводит считанное значение в окне. Таким образом, проверка на диапазон допустимых значений выполняется автоматически. Кроме того, для перечисляемых свойств **PropertyGrid** формирует списки, из которых пользователь может выбрать нужное значение.

Посредством свойств классов **AFE5832** и **AFE5832Die** можно изменить только основные параметры работы АЦП. Все остальные параметры можно редактировать в файле AFE5832.xlsm, который находится в папке doc SDK. После завершения редактирования нужно нажать кнопку «Create ini file» на странице «Result», и сгенерированный AFE5832.ini скопировать в папку PhotoSoundLibs\Device. При первом включении устройства новый файл загрузится в устройство и АЦП будет работать с новыми параметрами.

В таблице () приведён код из примера проекта AFE5818_AFE5832 из папки examples\visual\SdkExamples, который реализует описанную выше настройку АЦП. А в справочном разделе этого руководства можно найти описание всех свойств и методов классов **AFE5832** и **AFE5832Die**.

## Real-time обработка данных в MATLAB

Обработка данных в реальном масштабе времени в среде MATLAB рассмотрена ниже на примере построения синограммы. В результате обработки пользователь может видеть на экране видеоизображение, отображающее информацию о сигнале в сенсорах подключенного УЗИ датчика.

Получение данных АЦП для обработки осуществляется при помощи класса **DataLogger**. Экземпляры этого класса (регистраторы данных) создаются пользователем при помощи метода **CreateLogger** диспетчера устройств (класс **DeviceManager**):

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

Вызывать метод **CreateLogger** следует только после подключения к устройствам, в противном случае метод возвращает пустую ссылку. Метод возвращает ссылку на созданный регистратор данных, а его аргументами являются название создаваемого регистратора и длина очереди при записи в память. Название используется для сохранения настроек регистратора в конфигурационном файле. Длина очереди измеряется в кадрах данных АЦП и может быть 1 и выше. Для очереди с потерями, как правило, достаточно 1, а для очередей без потерь это значение следует подбирать экспериментально так, чтобы не было пропусков данных при обработке.

Далее следует задать свойства созданного регистратора, которые определяет продолжительность ввода данных и устройства, с которых данные считываются:

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

Регистратор начинает запись данных АЦП в очередь в памяти сразу после вызова его метода **StartLoggingToMemory(LossyQueue)**, где **LossyQueue = true** для очередей с потерями:

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

Перед тем как извлекать данные из очереди нужно подготовить для них буфер в памяти. Это можно сделать при помощи команды MATLAB **NET.createArray**. Объём выделяемой памяти можно задать максимальным, а затем определять действительный объём данных:

```
FrameBuffer = NET.createArray('System.Int16',dev.MaxSamplesToCap-
ture*dev.MaxChannelsToCapture);
```

Непосредственно извлечением данных из памяти занимается метод **GetFrame**:

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

Входными аргументами метода являются выделенный буфер данных в памяти и признак транспонирования кадра. Если он равен **true**, то первый индекс (строка) задаёт номер отсчёта (время), а второй (столбец) – номер канала. В противном случае – строка определяет канал, а столбец время. При вызове метод ожидает данные и, если время ожидания не вышло, то возвращает **valid = true** и заполняет остальные выходные аргументы параметрами кадра данных. Выходные параметры кадра данных более подробно описаны в справочном разделе.

Построение синограммы сводится к перестановке данных в соответствии с картой каналов. Карта каналов представляет собой упорядоченный массив с номерами каналов АЦП, индекс массива соответствует номеру сенсора датчика УЗИ:

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

В конце обработки результат масштабируется по диапазону значений и выводится в виде изображения с заданной цветовой палитрой на экран:

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

В таблице ниже представлен код скрипта realtime.m из папки examples\matlab, который реализует описанную выше обработку данных. В справочном разделе этого руководства можно найти описание всех свойств и методов класса **DataLogger**.

*Table 10 Пример скрипта MATLAB для real-time обработки данных АЦП*

```
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.MaxSamplesToCap-
ture*dev.MaxChannelsToCapture);
        logger.StartLoggingToMemory(true);

        while isvalid(fig)
            [valid,channels,samples,frame_num,trig_time,trig_src,sam-
ple_rate] = logger.GetFrame(FrameBuffer,false);
            if valid
                tmpData = single(FrameBuffer);
                frame = reshape(tmpData(1:(channels*samples)),samples,chan-
nels);
                mapped = frame(:,chmap);
                xlim([0 channels])
                ylim([0 samples])
                colorbar
                xlabel('Channels')
                ylabel('Samples')
                if isempty(image)
                    image = imagesc('XData',1:channels,'YData',1:sam-
ples,'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 обработка данных в LabVIEW

Обработка данных в реальном масштабе времени в среде LabVIEW рассмотрена ниже на примере построения синограммы. В результате обработки пользователь может видеть на экране видеоизображение, отображающее информацию о сигнале в сенсорах подключенного УЗИ датчика.

Получение данных АЦП для обработки осуществляется при помощи класса **DataLogger**. Экземпляры этого класса (регистраторы данных) создаются пользователем при помощи метода **CreateLogger** диспетчера устройств **DeviceManager** как показано на рисунке ниже.

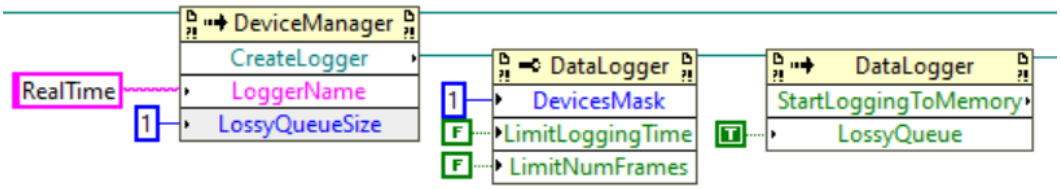


*Рисунок 17 Создание и настройка регистратора данных в Labview*

Вызывать метод **CreateLogger** следует только после подключения к устройствам, в противном случае метод возвращает пустую ссылку. Метод возвращает ссылку на созданный регистратор данных, а его аргументами являются название создаваемого регистратора и длина очереди при записи в память. Название используется для сохранения настроек регистратора в конфигурационном файле. Длина очереди измеряется в кадрах данных АЦП и может быть 1 и выше. Для очереди с потерями, как правило, достаточно 1, а для очередей без потерь это значение следует подбирать экспериментально так, чтобы не было пропусков данных при обработке.

Далее следует задать свойства созданного регистратора, которые определяет продолжительность ввода данных и устройства, с которых данные считываются. После этого можно начинать запись данных АЦП в очередь в память при помощи метода **StartLoggingToMemory(LossyQueue)**, где **LossyQueue = true** для очередей с потерями (Рисунок 17).

Перед тем как извлекать данные из очереди нужно подготовить для них буфер в памяти. Объём выделяемой памяти можно задать максимальным, а затем определять действительный объём данных (Рисунок 18).

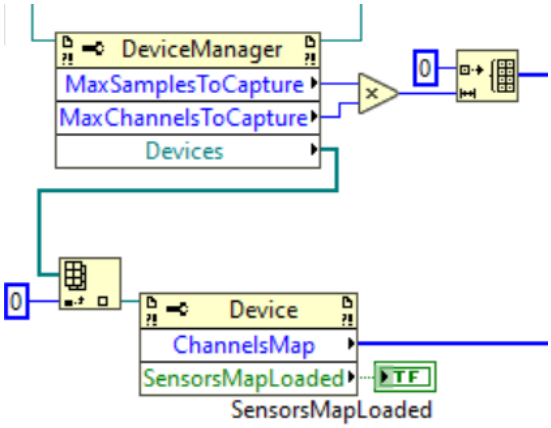


*Рисунок 18 Подготовка к обработке данных в LabView*

Непосредственно извлечением данных из памяти занимается метод **GetFrame**. Входными аргументами метода являются выделенный буфер данных в памяти и признак транспонирования кадра. Если он равен **true**, то первый индекс (строка) задаёт номер отсчёта (время), а второй (столбец) – номер канала. В противном случае – строка определяет канал, а столбец время. При вызове метод ожидает данные и, если время

ожидания не вышло, то возвращает **valid = true** и заполняет остальные выходные аргументы параметрами кадра данных. Выходные параметры кадра данных более подробно описаны в справочном разделе. Данные, скопированные в буфер данных, следует ограничить по длину в соответствии с выходными аргументами **FrameChannels** и **FrameSamples** и преобразовать в двумерный массив для последующей обработки (Рисунок 19).

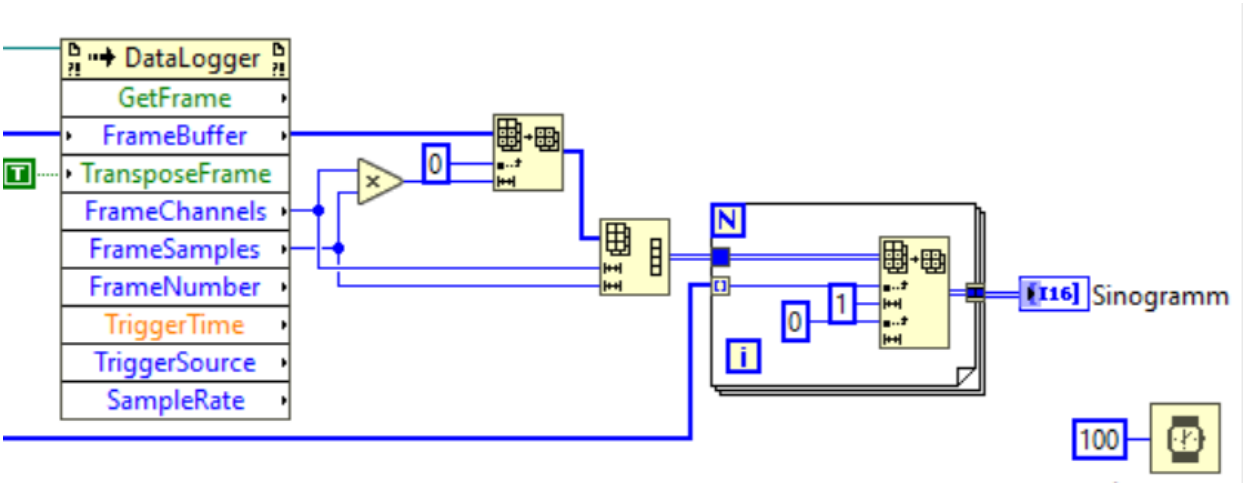


Рисунок 19 Извлечение данных из памяти в LabVIEW

Построение синограммы сводится к перестановке данных в соответствии с картой каналов. Карта каналов представляет собой упорядоченный массив с номерами каналов АЦП, индекс массива соответствует номеру сенсора датчика УЗИ (Рисунок 18). Результат перестановки масштабируется по диапазону значений и выводится в виде изображения с заданной цветовой палитрой на экран как показано на рисунке выше.

В папке examples\labview находится пример realtime.vi, который реализует описанную выше обработку данных. В справочном разделе этого руководства можно найти описание всех свойств и методов класса **DataLogger**.

## Real-time обработка данных в Visual C#

Обработка данных в реальном масштабе времени в среде Visual C# рассмотрена ниже на примере построения синограммы. В результате обработки пользователь может видеть на экране видеоизображение, отображающее информацию о сигнале в сенсорах подключенного УЗИ датчика.

Получение данных АЦП для обработки осуществляется при помощи класса **DataLogger**. Экземпляры этого класса (регистраторы данных) создаются пользователем при помощи метода **CreateLogger** диспетчера устройств (класс **DeviceManager**):

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

Вызывать метод **CreateLogger** следует только после подключения к устройствам, в противном случае метод возвращает пустую ссылку. Метод возвращает ссылку на созданный регистратор данных, а его аргументами являются название создаваемого регистратора и длина очереди при записи в память. Название используется для сохранения настроек регистратора в конфигурационном файле. Длина очереди измеряется в кадрах данных АЦП и может быть 1 и выше. Для очереди с потерями, как правило, достаточно 1, а для очередей без потерь это значение следует подбирать экспериментально так, чтобы не было пропусков данных при обработке.

Далее следует задать свойства созданного регистратора, которые определяет продолжительность ввода данных и устройства, с которых данные считываются:

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

Регистратор начинает запись данных АЦП в очередь в памяти сразу после вызова его метода **StartLoggingToMemory(LossyQueue)**, где **LossyQueue = true** для очередей с потерями:

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

Перед тем как извлекать данные из очереди нужно выделить для них буфер в памяти. Объём выделяемой памяти можно задать максимальным, а затем определять действительный объём данных:

```
frameBuffer = new short[deviceManager.MaxSamplesToCapture * deviceManager.MaxChannel-
sToCapture];
```

Непосредственно извлечением данных из памяти занимается метод **GetFrame**:

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

Входными аргументами метода являются выделенный буфер данных в памяти **frame-Buffer** и признак транспонирования кадра **transposeFrame**. Если он равен **true**, то первый индекс (строка) задаёт номер отсчёта (время), а второй (столбец) – номер канала. В противном случае – строка определяет канал, а столбец время. При вызове метод ожидает данные и, если время ожидания не вышло, то он возвращает **true** и заполняет выходные аргументы параметрами кадра данных. Выходные параметры кадра данных более подробно описаны в справочном разделе.

Построение синограммы сводится к перестановке данных в соответствии с картой каналов. Карта каналов представляет собой упорядоченный массив с номерами каналов АЦП, индекс массива соответствует номеру сенсора датчика УЗИ:

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

Результат перестановки масштабируется по диапазону значений и выводится в виде изображения с заданной цветовой палитрой на экран. Поскольку для обработки данных требуется быстродействия, то код обработки заключается в блок **unsafe**:

```
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;
    }
}
```

Во избежание подвисаний пользовательского интерфейса, обработка данных осуществляется в отдельном потоке класса **BackgroundWorker**. Методы и свойства класса **DataLogger** являются ThreadSafe, но могут вызывать события, например, событие **OnError** диспетчера устройств. Если обработчик события использует графические элементы интерфейса, то обращаться к ним следует через метод **Invoke** этих элементов. Ниже приведён пример вывода сообщения об ошибке при помощи элемента типа **Label**:

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

В папке examples\visual\SdkExamples находится проект RealTime, который реализует описанную выше обработку данных. В справочном разделе этого руководства можно найти описание всех свойств и методов класса **DataLogger**.

# Справочник по библиотеке классов PhotoSoundClasses.dll

## Класс Capture

*Таблица 1 Свойства и методы класса Capture*

| Название | Тип | Описание |
|---|---|---|
| **Configure** | Метод | Перезапускает сбор данных АЦП с параметрами из свойств класса |
| **SamplesToCapture** | Integer, 32 bit | Число отсчётов для сбора данных на канал АЦП |
| **FramesPerPacket** | Integer, 32 bit | Число кадров данных АЦП в одном пакете, передаваемом по системной шине данных |
| **DecimationFactor** | Integer, 32 bit | Коэффициент прореживания отсчётов АЦП |

| | | |
|---|---|---|
| **WaitTrigger** | Boolean | Признак ожидания события триггера перед запуском сбора данных |
| **EnabledAdcMask** | Unsigned, 32 bit | Маска АЦП, разрешенных для сбора данных |
| **AutoUpdate** | Boolean | Признак автоматической передачи настроек в устройство при изменении свойств класса |

### Configure

Метод **Configure** класса **Capture** запрещает сбор данных АЦП, затем передаёт настройки, рассчитанные из свойств класса **Capture** в устройство по системной шине, а затем снова разрешает сбор данных АЦП. Также метод присваивает значение **true** свойству **AutoUpdate** класса **Capture**.

### SamplesToCapture

Свойство **SamplesToCapture** класса **Capture** задаёт число отсчётов данных на канал АЦП, которые записываются в буфер кадра данных в устройстве после запуска сбора, а после передаются в ПК по системной шине. Максимальное число отсчётов зависит от размера буфера кадра устройства. Чтобы узнать максимальное число отсчётов для конкретного устройства нужно прочитать значение свойства **MaxSamples** соответствующего экземпляра класса **Device**.

### FramesPerPacket

Свойство **FramesPerPacket** класса **Capture** задаёт число кадров данных АЦП в одном пакете, передаваемом по системной шине данных. Поскольку система прерываний ПК имеет ограниченную частоту работы, то для уменьшения частоты прерываний кадры данных АЦП объединяются в пакет. Это позволяет увеличить скорость передачи данных до величины пропускной способности системной шины. С другой стороны, частота получения кадров программой пользователя снижается во **FramesPerPacket** раз. Это может привести к слишком низкой частоте обновления данных на графике, если запуск сбора данных производится по событию триггера с небольшой частотой повторения. Поэтому в таких случаях значение **FramesPerPacket** следует задать равным 1. В остальных случаях **FramesPerPacket** можно оставить равным 10.

### DecimationFactor

Свойство **DecimationFactor** класса **Capture** управляет прореживанием отсчётов данных АЦП. Если оно равно 1, то отсчёты записываются в буфер кадра устройства без пропусков. Если **DecimationFactor** = 2, то записывается отсчёт 1, затем пропуск записи, затем записывается отсчёт 3 и т.д. Если **DecimationFactor** = 3, то записывается только каждый третий отсчёт. Таким образом, частота дискретизации выходных данных, передаваемых в ПК, равна частоте дискретизации АЦП, поделённое на **DecimationFactor**. Чтобы узнать

частоту дискретизации АЦП для конкретного устройства нужно прочитать значение свойства **MaxSampleRate** соответствующего экземпляра класса **Device**. При таком прореживании данных может возникнуть эффект наложения спектра, если полоса пропускания входного сигнала АЦП больше половины частоты дискретизации выходных данных. Для устранения этого эффекта можно настроить полосу пропускания фильтра нижних частот АЦП (см. описание классов АЦП ниже).

### WaitTrigger

Свойство **WaitTrigger** класса **Capture** разрешает или запрещает ожидание события триггера перед запуском сбора данных. Если ожидание запрещено, то новый запуск сбора данных следует сразу за окончанием передачи предыдущего кадра данных АЦП из буфера устройства в ПК. Если разрешено, то по окончанию передачи данных сначала ожидается событие триггера, а затем следует запуск. Если частота повторения событий триггера слишком высокая, то событие триггера может быть до окончания передачи данных в ПК. В этом случае запуска сбора данных не будет, но будет инкрементирован счётчик событий от триггера, а также зафиксирован пропуск события. Число пропущенных событий может быть считано из свойства **LostEvents** класса **Device**.

### EnabledAdcMask

Свойство **EnabledAdcMask** класса **Capture** задаёт разрешённые для сбора данных чипы АЦП. Свойство является общим для всех устройств, поэтому число чипов АЦП определяется устройством с наибольшим количеством установленных чипов. Это число можно считать из свойства **MaxAdcPerDevice** диспетчера устройств (класс **DeviceManager**). Каждый бит **EnabledAdcMask** соответствует одному чипу АЦП, бит 0 – чипу № 1, бит 1 – чипу № 2 и т.д. Если значение бита равно 1, то АЦП разрешено для сбора данных, в противном случае – запрещено. Уменьшенное количество чипов АЦП может потребоваться для сокращения объёма передаваемых в ПК данных. Это позволяет сократить время передачи и работать с большей частотой повторений событий триггера, а также сократить размер файлов данных при записи на диск.

### AutoUpdate

Свойство **AutoUpdate** класса **Capture** разрешает или запрещает автоматическую передачу настроек в устройство при изменении свойств класса **Capture**. Если значение свойства равно **true**, то при записи нового значения в какое-либо из свойства класса **Capture**, обновлённые настройки автоматически передаются в устройство. Если значение свойство равно **false**, то можно присвоить новые значения нескольких свойствам класса **Capture**, а потом вызвать метод **Configure**, который передаст настройки в устройство и восстановит значение true для свойства **AutoUpdate**.

## Класс Trigger

*Таблица 2 Свойства и методы класса Trigger*

| Название | Тип | Описание |
|---|---|---|
| **Configure** | Метод | Передаёт настройки из свойств класса в устройство по системной шине |
| **GetInputFrequencies** | Метод | Считывает из устройства по системной шине значения частот сигналов на входах триггера |
| **TriggerOutputs** | Массив TriggerOutput | Массив экземпляров класса TriggerOutput |
| **InputNames** | Массив string | Массив с названиями входов триггера |
| **SlaveDelays** | Массив Integer, 32 bit | Массив задержек запуска сбора данных по отношению к сигналу с ведомого HDMI разъёма |
| **GeneratorFrequency** | Double | Частота внутреннего генератора, Гц |
| **ConnectToGenerator** | Boolean | Признак разрешения использования внутреннего генератора для запуска сбора данных |
| **InputsDelay** | Integer, 32 bit | Задержка запуска сбора данных по отношению к сигналу с выбранного входа триггера или генератора в тактах частоты дискретизации АЦП |
| **InputsGuard** | Integer, 32 bit | Интервал защиты от шума на входах триггера в тактах частоты дискретизации АЦП |
| **EnabledInputsMask** | Unsigned, 32 bit | Маска входов триггера, разрешенных для запуска сбора данных |
| **InvertedInputsMask** | Unsigned, 32 bit | Маска входов триггера с отрицательной полярностью входного сигнала |
| **AutoUpdate** | Boolean | Признак автоматической передачи настроек в устройство при изменении свойств класса |

### Configure

Метод **Configure** класса **Trigger** передаёт настройки, рассчитанные из свойств класса **Trigger** в устройство по системной шине, а также присваивает значение **true** свойству **AutoUpdate**. Хотя класс **Trigger** и содержит массив экземпляров класса **TriggerOutput**, настройки из свойств класса **TriggerOutput** в устройство не передаются. Для этого у класса **TriggerOutput** есть свой метод **Configure** и каждый выход триггера настраивается по отдельности.

### GetInputFrequencies

Метод **UpdateInputFrequencies** класса **Trigger** считывает из устройства по системной шине значения частот сигналов на входах триггера и возвращает массив чисел со значениями частот сигналов в герцах.

### TriggerOutputs

Свойство **TriggerOutputs** класса **Trigger** представляет собой массив экземпляров класса **TriggerOutput**. Длина массива равна количеству выходов триггера и каждый элемент массива соответствует своему выходу триггера.

### InputNames

Свойство **InputNames** класса **Trigger** представляет собой массив строк с пользовательскими названиями входов триггера. Строки могут содержать любой описательный текст для удобства обозначения входов.

### SlaveDelays

Свойство **SlaveDelays** класса **Trigger** представляет собой массив целых чисел с задержками запуска сбора данных для каждого ведомого устройства. Длина массива равна 14 – максимальному количеству ведомых устройств. Элемент 0 массива соответствует первому ведомому устройству, а элемент 13 – последнему в цепи подключений через HDMI кабели. Задержка калибруется по графику входного сигнала от аналогового сигнала от одного источника, поданного на входы АЦП при помощи кабелей одинаковой длины. Величины задержки должна быть в пределах от 0 до значения свойства **InputsDelay** ведущего устройства.

### GeneratorFrequency

Свойство **GeneratorFrequency** класса **Trigger** задаёт частоту внутреннего генератора в герцах. Внутренний генератора может служить источником сигнала триггера как для запуска сбора данных, так и для любого из выходов триггера.

### ConnectToGenerator

Свойство **ConnectToGenerator** класса **Trigger** разрешает или запрещает использование внутреннего генератора для запуска сбора данных. Для запуска сбора данных от генератора следует также разрешить ожидание события триггера (свойство **WaitTrigger** класса **Capture**).

### InputsDelay

Свойство **InputsDelay** класса **Trigger** задаёт задержку запуска сбора данных по отношению к сигналу триггера в тактах частоты дискретизации АЦП. Сигнал запуска может поступать как с одного или нескольких входов триггера, так и с внутреннего генератора.

### InputsGuard

Свойство **InputsGuard** класса **Trigger** задаёт интервал защиты от шума на входах триггера в тактах частоты дискретизации АЦП. Событие триггера фиксируется по переднему фронту сигнала триггера для положительной полярности сигнала или по заднему – для отрицательной. Импульсный шум на входе триггера может привести к тому, что будет зафиксировано несколько фронтов. Хотя запуск сбора данных произойдёт по первому фронту, остальные фронты могут привести к неверным значениям счётчика событий триггера и ложному количеству пропусков событий триггера.

### EnabledInputsMask

Свойство **EnabledInputsMask** класса **Trigger** задаёт разрешённые для запуска сбора данных входы триггера. Каждый бит **EnabledInputsMask** соответствует одному входу триггера, бит 0 – входу № 1, бит 1 – входу № 2 и т.д. Если значение бита равно 1, то сигнал со входа триггера запускает сбор данных, если 0 – то нет. Если несколько входов разрешены для запуска, то сбор данных будет запущен по сигналу с любого из этих входов. Номер входа, по сигналу с которого был запущен сбор данных, запоминается устройством и передаётся в ПК в кадре данных АЦП. Для запуска сбора данных от входа триггера следует также разрешить ожидание события триггера (свойство **WaitTrigger** класса **Capture**).

### InvertedInputsMask

Свойство **InvertedInputsMask** класса **Trigger** задаёт входы триггера с отрицательной полярностью входного сигнала. Каждый бит **InvertedInputsMask** соответствует одному входу триггера, бит 0 – входу № 1, бит 1 – входу № 2 и т.д. Если значение бита равно 1, то вход триггера имеет отрицательную полярность входного сигнала, если 0 – то положительную.

### AutoUpdate

Свойство **AutoUpdate** класса **Trigger** разрешает или запрещает автоматическую передачу настроек в устройство при изменении свойств класса **Trigger**. Если значение свойства равно **true**, то при записи нового значения в какое-либо из свойства класса **Trigger**, обновлённые настройки автоматически передаются в устройство. Если значение свойство равно **false**, то можно присвоить новые значения нескольких свойствам класса **Trigger**, а потом вызвать метод **Configure**, который передаст настройки в устройство и восстановит значение true для свойства **AutoUpdate**.

## Класс TriggerOutput

*Таблица 3 Свойства и методы класса TriggerOutput*

| Название | Тип | Описание |
|---|---|---|

| | | |
|---|---|---|
| **Configure** | Метод | Передаёт настройки из свойств класса в устройство по системной шине |
| **PulseWidth** | Double | Длительность импульса на выходе триггера в микросекундах |
| **Delay** | Double | Задержка сигнала на выходе триггера в микросекундах |
| **SourcesMask** | Unsigned, 32 bit | Маска входов триггера, подключенных к выходу триггера |
| **ConnectToGenerator** | Boolean | Признак подключения внутреннего генератора к выходу триггера |
| **Enable** | Boolean | Признак разрешения сигнала на входе триггера |
| **InvertInputsDelay** | Boolean | Признак отрицательной полярности сигнала на выходе триггера |
| **AutoUpdate** | Boolean | Признак автоматической передачи настроек в устройство при изменении свойств класса |

### Configure

Метод **Configure** класса **TriggerOutput** передаёт настройки, рассчитанные из свойств класса **TriggerOutput** в устройство по системной шине, а также присваивает значение **true** свойству **AutoUpdate**.

### PulseWidth

Свойство **PulseWidth** класса **TriggerOutput** задаёт длительность импульса на выходе триггера в микросекундах. Длительность импульса не должна превышать период повторения импульсов на выходе триггера.

### Delay

Свойство **Delay** класса **TriggerOutput** задаёт задержку сигнала на выходе триггера по отношению к сигналу на подключенном входе триггера или к сигналу от внутреннего генератора в микросекундах.

### SourcesMask

Свойство **SourcesMask** класса **TriggerOutput** определяет какие входы триггера подключены к выходу триггера. Каждый бит **SourcesMask** соответствует одному входу триггера, бит 0 – входу № 1, бит 1 – входу № 2 и т.д. Если значение бита равно 1, то вход триггера подключен к выходу триггера, в противном случае – не подключен. Если несколько входов подключены к выходу, то сигнал на выходе является функцией логического ИЛИ сигналов со входов триггера.

### ConnectToGenerator

Свойство **ConnectToGenerator** класса **TriggerOutput** подключает внутренний генератор к выходу триггера или отключает его. Внутренний генератор может быть подключен совместно с одним или несколькими входами триггера. В этом случае сигнал на выходе триггера является функцией логического ИЛИ сигналов со входов триггера.

### Enable

Свойство **Enable** класса **TriggerOutput** включает или выключает импульсный сигнал на выходе триггера.

### Invert

Свойство **Invert** класса **TriggerOutput** определяет полярность сигнала на выходе триггера. Если оно равно **true**, то полярность сигнала отрицательная, если **false** – положительная.

### AutoUpdate

Свойство **AutoUpdate** класса **TriggerOutput** разрешает или запрещает автоматическую передачу настроек в устройство при изменении свойств класса **Trigger**. Если значение свойства равно **true**, то при записи нового значения в какое-либо из свойства класса **Trigger**, обновлённые настройки автоматически передаются в устройство. Если значение свойство равно **false**, то можно присвоить новые значения нескольких свойствам класса **Trigger**, а потом вызвать метод **Configure**, который передаст настройки в устройство и восстановит значение true для свойства **AutoUpdate**.

## Класс DataLogger

*Таблица 4 Свойства и методы класса DataLogger*

| Название | Тип | Описание |
|---|---|---|
| **Configure** | Метод | Вызывает событие OnPropertyChanged диспетчера устройств DeviceManager |
| **StartLoggingToFile** | Метод | Запускает запись данных в файл |
| **StartLog-gingToMemory** | Метод | Запускает запись данных в память |
| **StopLogging** | Метод | Останавливает запись данных |
| **GetFrame** | Метод | Извлекает кадр данных АЦП из очереди в памяти |
| **OnStartLogging** | Событие | Событие вызывается при запуске записи данных |
| **OnStopLogging** | Событие | Событие вызывается при остановке записи данных |
| **LimitNumFrames** | Boolean | Признак ограничения числа записываемых кадров |
| **LimitLoggingTime** | Boolean | Признак ограничения времени записи |
| **LimitFileSize** | Boolean | Признак ограничения размера файла данных |

| | | |
|---|---|---|
| **DataFolder** | String | Полный путь к папке для записи файлов данных |
| **DevicesMask** | Unsigned, 32 bit | Маска устройств, данные которых записываются в память или в файл |
| **MaxLoggedFramesInputsDelay** | Integer, 32 bit | Максимальное число записываемых кадров данных |
| **MaxFileSize** | Integer, 32 bit | Максимальный размер файла данных в мегабайтах |
| **LoggingTimeout** | Integer, 32 bit | Максимальное время записи в файл или в память в секундах |
| **Logging** | Boolean | Признак активной записи данных |
| **Progress** | Integer, 32 bit | Прогресс записи данных в процентах |
| **NumLoggedFrames** | Integer, 32 bit | Текущее число записанных кадров данных |
| **LoggingTime** | Double | Текущее время записи данных в память или в файл |
| **FileSize** | Double | Текущее размер файла данных в мегабайтах |
| **AutoUpdate** | Boolean | Признак автоматической передачи настроек в устройство при изменении свойств класса |

### Configure

Метод **Configure** для класса **DataLogger** не выполняет каких-либо действий и зарезервирован на будущее.

### StartLoggingToFile

Метод **StartLoggingToFile**(**FileName**) класса **DataLogger** запускает запись данных АЦП в файл. Аргумент **FileName** типа **String** передаёт имя файла без расширения и без пути к файлу. Метод возвращает **true**, если запись началась без ошибок.

### StartLoggingToMemory

Метод **StartLoggingToMemory(LossyQueue)** класса **DataLogger** запускает запись данных АЦП в очередь в памяти. Аргумент **LossyQueue** типа Boolean указывает, что нужно создавать очередь с потерями, в противном случае будет создана очередь без потерь. Длина очереди указывается при создании экземпляра класса **DataLogger** в методе **CreateLogger** диспетчера устройств **DeviceManager**. Метод возвращает **true**, если запись началась без ошибок.

### StopLogging

Метод **StopLogging** класса **DataLogger** останавливает запись данных АЦП в файл или в память. Поскольку для завершения записи в файл требуется некоторое время, следует

дождаться окончания записи посредством проверки признака **Logging** класса **DataLogger** или ожидать события **OnStopLogging** класса **DataLogger**.

### GetFrame

Метод **GetFrame(FrameBuffer, TransposeFrame, FrameChannels, FrameSamples, FrameNumber, TriggerTime, TriggerSource, SampleRate)** класса **DataLogger** ожидает один кадр данных АЦП, извлекает его из очереди в памяти и копирует в представленный буфер **FrameBuffer**. Аргумент **TransposeFrame** должен быть **true**, если двумерный массив данных должен иметь отсчёты по строкам (первый индекс) и каналы по столбцам и должен быть равен **false**, если двумерный массив данных должен иметь каналы по строкам и отсчёты по столбцам. Остальные аргументы – ссылки для приёма параметров кадра данных: **FrameChannels** – число каналов АЦП в кадре, **FrameSamples** – число отсчётов АЦП в кадре, **FrameNumber** – порядковый номер кадра, **TriggerTime** – отсчёт времени события триггера для данного кадра в миллисекундах, **SampleRate** – частота дискретизации данных кадра в герцах. Метод возвращает **true**, если данные были извлечены из очереди успешно и **false**, если данные время ожидания кадра вышло.

### OnStartLogging

Событие **OnStartLogging** класса **DataLogger** вызывается сразу после успешного запуска записи данных в файл или в память. Обработчик события должен иметь стандартные аргументы типа **object** и **EventArgs**.

### OnStopLogging

Событие **OnStopLogging** класса **DataLogger** вызывается по автоматическому или принудительному завершению записи данных в файл или в память. Обработчик события должен иметь стандартные аргументы типа **object** и **EventArgs**.

### LimitNumFrames

Свойство **LimitNumFrames** класса **DataLogger** разрешает или запрещает автоматическую остановку записи данных в файл или в память, если число записанных кадров данных АЦП равно максимальному значению, задаваемое свойством **MaxLoggedFrames** класса **DataLogger**.

### LimitLoggingTime

Свойство **LimitLoggingTime** класса **DataLogger** разрешает или запрещает автоматическую остановку записи данных в файл или в память, если время записи превысило максимальное значение, задаваемое свойством **LoggingTimeout** класса **DataLogger**.

### LimitFileSize

Свойство **LimitFileSize** класса **DataLogger** разрешает или запрещает автоматическую остановку записи данных в файл или в память, если размер файла превысил максимальное значение, задаваемое свойством **MaxFileSize** класса **DataLogger**.

### DataFolder

Свойство **DataFolder** класса **DataLogger** определяет полный путь к папке для сохранения записываемых файлов данных.

### DevicesMask

Свойство **DevicesMask** класса **DataLogger** определяет устройства, данные с которых записываются в файл или в память. Каждый бит **DevicesMask** соответствует одному устройству, бит 0 – устройству с **Id** = 0, бит 1 – устройству с **Id** = 1 и т.д. Если значение бита равно 1, то данные с устройства записываются, в противном случае – нет.

### MaxLoggedFrames

Свойство **MaxLoggedFrames** класса **DataLogger** задаёт максимальное число записываемых кадров данных АЦП. Для остановки записи по превышению максимального числа записанных кадров нужно также установить в **true** свойство **LimitNumFrames** класса **DataLogger**.

### MaxFileSize

Свойство **MaxFileSize** класса **DataLogger** задаёт максимальный размер файла данных в мегабайтах. Для остановки записи по превышению максимального размера файла нужно также установить в **true** свойство **LimitFileSize** класса **DataLogger**.

### LoggingTimeout

Свойство **LoggingTimeout** класса **DataLogger** задаёт максимальное время записи данных в файл или в память в секундах. Для остановки записи по превышению максимального времени записи нужно также установить в **true** свойство **LimitLoggingTime** класса **DataLogger**.

### Logging

Свойство **Logging** класса **DataLogger** показывает состояние записи данных и является свойством только для чтения. Если значение свойства равно **true**, то запись производится, если **false** – то нет.

### Progress

Свойство **Progress** класса **DataLogger** показывает текущий прогресс записи данных в процентах и является свойством только для чтения. Прогресс записи рассчитывается либо для окончания записи по превышению числа кадров, либо по превышению размера файла.

При этом значение прогресса выводится для того условия, которое будет выполнено раньше. Остановка записи хотя бы по одному из условий должна быть разрешена свойствами **LimitFileSize** или **LimitNumFrames** класса **DataLogger**.

### NumLoggedFrames

Свойство **NumLoggedFrames** класса **DataLogger** показывает текущее число записанных в файл или память кадров данных АЦП и является свойством только для чтения.

### LoggingTime

Свойство **LoggingTime** класса **DataLogger** показывает текущее время в секундах с начала записи данных в файл или в память и является свойством только для чтения.

### FileSize

Свойство **FileSize** класса **DataLogger** показывает текущий размер файла данных в мегабайтах и является свойством только для чтения.

### AutoUpdate

Свойство **AutoUpdate** класса **DataLogger** разрешает или запрещает оповещение посредством события **OnPropertyChanged** диспетчера устройств при изменении свойств класса **DataLogger**. Перед началом группового изменения свойств класса **DataLogger** этому свойству можно присвоить значение **false**, а по окончанию **true**. В этом случае событие **OnPropertyChanged** будет вызвано только один раз.

## Класс AFE5818

*Таблица 5 Свойства и методы класса AFE5818*

| Название | Тип | Описание |
|---|---|---|
| **Configure** | Метод | Передаёт настройки из свойств класса в устройство по системной шине |
| **ConfiguredDevic-esMask** | Unsigned, 32 bit | Маска устройств для настройки |
| **ConfiguredAdcMask** | Unsigned, 32 bit | Маска чипов АЦП для настройки |
| **Vca1EqualsVca2** | Boolean | Если true, то настройки для VCA №2 берутся из свойства Vca1, если false, то из Vca2 |
| **Vca1, Vca2** | Класс AFE5818Vca | Ссылки на класс AFE5818Vca с настройками для VCA №1 и для VCA №2 |
| **AutoUpdate** | Boolean | Признак автоматической передачи настроек в устройство при изменении свойств класса |

### Configure

Метод **Configure** класса **AFE5818** передаёт настройки, рассчитанные из свойств класса **AFE5818** в устройство по системной шине, а также присваивает значение **true** свойству **AutoUpdate**.

### ConfiguredDevicesMask

Свойство **ConfiguredDevicesMask** класса **AFE5818** задаёт маску устройств для настройки. Каждый бит маски соответствует одному устройству: бит 0 – устройству с Id = 0, бит 1 – устройству с Id = 1 и т.д., где Id – идентификатор устройства на системной шине. При изменении маски настройки не передаются автоматически в устройство, но новая маска учитывается при изменении других свойств. Так если какому-либо свойству будет присвоено такое же значение, то если маска не была изменена, то настройки не будут переданы в устройство, а если была изменена – то будут. Количество подключенных устройств можно прочитать из свойства **DevicesCount** диспетчера устройств.

### ConfiguredAdcMask

Свойство **ConfiguredAdcMask** класса **AFE5818** задаёт маску чипов АЦП для настройки. Каждый бит маски соответствует одному чипу: бит 0 – чипу №1, бит 1 – чипу №2 и т.д. При изменении маски настройки не передаются автоматически в устройство, но новая маска учитывается при изменении других свойств. Так если какому-либо свойству будет присвоено такое же значение, то если маска не была изменена, то настройки не будут переданы в устройство, а если была изменена – то будут.

### Vca1EqualsVca2

Свойство **Vca1EqualsVca2** класса **AFE5818** определяет, являются ли настройки VCA №1 и VCA №2 одинаковыми. Если значение свойства **true**, то настройки для VCA №2 берутся из свойства **Vca1** класса **AFE5818**, если **false**, то из **Vca2**.

### AutoUpdate

Свойство **AutoUpdate** класса **AFE5818** разрешает или запрещает автоматическую передачу настроек в устройство при изменении свойств класса **AFE5818**. Если значение свойства равно **true**, то при записи нового значения в какое-либо из свойств класса **AFE5818**, обновлённые настройки автоматически передаются в устройство. Если значение свойство равно **false**, то можно присвоить новые значения нескольких свойствам класса **AFE5818**, а потом вызвать метод **Configure**, который передаст настройки в устройство и восстановит значение **true** для свойства **AutoUpdate**.

### Vca1, Vca2

Свойства **Vca1** и **Vca2** класса **AFE5818** являются ссылками на класс **AFE5818Vca**. Описание этого класса представлено ниже. При изменении свойств класса **AFE5818Vca**

будет автоматически вызван метод **Configure** класса **AFE5818**, если свойство **AutoUpdate** класса **AFE5818** равно **true**.

*Таблица 6 Свойства и методы класса AFE5818Vca*

| Название | Тип | Описание |
|---|---|---|
| **HpfCutoffDivided** | Boolean | Если true, то частота среза фильтра высоких частот блока LNA уменьшается в три раза |
| **LowNoiseMode** | Boolean | Признак включения режима пониженного шума VCA |
| **PgaHpfDisabled** | Boolean | Признак отключения фильтра высоких частот блока PGA |
| **LnaHpfDisabled** | Boolean | Признак отключения фильтра высоких частот блока LNA |
| **PgaClampEnabled** | Boolean | Признак подключения ограничителя напряжения PGA |
| **F5MHzLpfEnabled** | Boolean | Признак подключения фильтра низких частот |
| **TgcAttEnabled** | Boolean | Признак подключения аттенюатора в блоке TGC |
| **PowerMode** | Enum | Режим энергопотребления блока VCA |
| **HpfCutoffFreq** | Enum | Частота среза фильтра высоких частот |
| **LpfCutoffFreq** | Enum | Частота среза фильтра низких частот |
| Error! Reference source not found. | Enum | Коэффициент передачи аттенюатора в блоке TGC |
| **LnaGlobalGain** | Enum | Коэффициент усиления блока LNA |
| **PgaGain** | Enum | Коэффициент усиления блока PGA |

### HpfCutoffDivided

Свойство **HpfCutoffDivided** класса **AFE5818Vca** разрешает (**true**) или запрещает (**false**) уменьшение частоты среза фильтра высоких частот блока LNA в три раза.

### LowNoiseMode

Свойство **LowNoiseMode** класса **AFE5818Vca** разрешает (**true**) или запрещает (**false**) режим пониженного шума **VCA** для высокоимпедансных датчиков.

### PgaHpfDisabled

Свойство **PgaHpfDisabled** класса **AFE5818Vca** подключает (**false**) или отключает (**true**) фильтр высоких частот блока PGA.

### LnaHpfDisabled

Свойство **LnaHpfDisabled** класса **AFE5818Vca** подключает (**false**) или отключает (**true**) фильтр высоких частот блока LNA.

### PgaClampEnabled

Свойство **PgaClampEnabled** класса **AFE5818Vca** подключает (**true**) или отключает (**false**) ограничитель напряжения в блоке PGA.

### F5MHzLpfEnabled

Свойство **F5MHzLpfEnabled** класса **AFE5818Vca** подключает (**true**) или отключает (**false**) фильтр низких частот первого порядка с полосой пропускания 5 МГц.

### TgcAttEnabled

Свойство **TgcAttEnabled** класса **AFE5818Vca** подключает (**true**) или отключает (**false**) аттенюатор в блоке TGC.

### PowerMode

Свойство **PowerMode** класса **AFE5818Vca** определяет режим энергопотребления блока VCA, является перечисляемым свойством и ему можно присвоить три значения: **LowNoise**, **LowPower**, **MediumPower**.

### HpfCutoffFreq

Свойство **HpfCutoffFreq** класса **AFE5818Vca** определяет частоту среза фильтра высоких частот, является перечисляемым свойством и ему можно присвоить значения: **_50_kHz**, **_100_kHz**, **_150_kHz**, **_200_kHz**.

### LpfCutoffFreq

Свойство **LpfCutoffFreq** класса **AFE5818Vca** определяет частоту среза фильтра низких частот, является перечисляемым свойством и ему можно присвоить значения: **_10_MHz**, **_15_MHz**, **_20_MHz**, **_30_MHz**, **_35_MHz**, **_50_MHz**.

### TgcAttenuation

Свойство **TgcAttenuation** класса **AFE5818Vca** задаёт коэффициент передачи аттенюатора в блоке TGC, является перечисляемым свойством и ему можно присвоить значения: **_0_dB**, **_6_dB**, **_12_dB**, **_18_dB**, **_24_dB**, **_30_dB**, **_36_dB**.

### LnaGlobalGain

Свойство **LnaGlobalGain** класса **AFE5818Vca** определяет коэффициент усиления блока LNA, является перечисляемым свойством и ему можно присвоить значения: **_12_dB**, **_18_dB**, **_24_dB**.

### PgaGain

Свойство **PgaGain** класса **AFE5818Vca** определяет коэффициент усиления блока PGA, является перечисляемым свойством и ему можно присвоить значения: **_24_dB**, **_30_dB**.

## Класс AFE5832

*Таблица 7 Свойства и методы класса AFE5832*

| Название | Тип | Описание |
|---|---|---|

| | | |
|---|---|---|
| **Configure** | Метод | Передаёт настройки из свойств класса в устройство по системной шине |
| **ConfiguredDevic-esMask** | Unsigned, 32 bit | Маска устройств для настройки |
| **ConfiguredAdcMask** | Unsigned, 32 bit | Маска чипов АЦП для настройки |
| **EnableAttenuatorHpf** | Boolean | Признак подключения фильтра высоких частот аттенюатора |
| **AttenuatorHpfCorner** | Enum | Крутизна фильтра высоких частот аттенюатора |
| **OddEqualEven** | Boolean | Если true, то настройки для Even die берутся из свойства Odd, если false, то из Even |
| **Odd, Even** | Класс AFE5832Die | Ссылки на класс AFE5832Die с настройками для Odd die и для Even die |
| **AutoUpdate** | Boolean | Признак автоматической передачи настроек в устройство при изменении свойств класса |

### Configure

Метод **Configure** класса **AFE5832** передаёт настройки, рассчитанные из свойств класса **AFE5832** в устройство по системной шине, а также присваивает значение **true** свойству **AutoUpdate**.

### ConfiguredDevicesMask

Свойство **ConfiguredDevicesMask** класса **AFE5832** задаёт маску устройств для настройки. Каждый бит маски соответствует одному устройству: бит 0 – устройству с Id = 0, бит 1 – устройству с Id = 1 и т.д., где Id – идентификатор устройства на системной шине. При изменении маски настройки не передаются автоматически в устройство, но новая маска учитывается при изменении других свойств. Так если какому-либо свойству будет присвоено такое же значение, то если маска не была изменена, то настройки не будут переданы в устройство, а если была изменена – то будут. Количество подключенных устройств можно прочитать из свойства **DevicesCount** диспетчера устройств.

### ConfiguredAdcMask

Свойство **ConfiguredAdcMask** класса **AFE5832** задаёт маску чипов АЦП для настройки. Каждый бит маски соответствует одному чипу: бит 0 – чипу №1, бит 1 – чипу №2 и т.д. При изменении маски настройки не передаются автоматически в устройство, но новая маска учитывается при изменении других свойств. Так если какому-либо свойству будет присвоено такое же значение, то если маска не была изменена, то настройки не будут переданы в устройство, а если была изменена – то будут.

### EnableAttenuatorHpf

Свойство **EnableAttenuatorHpf** класса **AFE5832** подключает (**true**) или отключает (**false**) фильтр высоких частот блока аттенюатора.

### AttenuatorHpfCorner

Свойство **AttenuatorHpfCorner** класса **AFE5832** определяет крутизну фильтра высоких частот аттенюатора, является перечисляемым свойством и ему можно присвоить значения: **C2**, **C3**, **C4**, **C5**, **C6**, **C7**, **C8**, **C9**, **C10**.

### OddEqualEven

Свойство **OddEqualEven** класса **AFE5832** определяет, являются ли настройки Even die и Odd die одинаковыми. Если значение свойства **true**, то настройки для Even die берутся из свойства **Odd** класса **AFE5832**, если **false**, то из **Even**.

### AutoUpdate

Свойство **AutoUpdate** класса **AFE5832** разрешает или запрещает автоматическую передачу настроек в устройство при изменении свойств класса **AFE5832**. Если значение свойства равно **true**, то при записи нового значения в какое-либо из свойств класса **AFE5832**, обновлённые настройки автоматически передаются в устройство. Если значение свойство равно **false**, то можно присвоить новые значения нескольких свойствам класса **AFE5832**, а потом вызвать метод **Configure**, который передаст настройки в устройство и восстановит значение **true** для свойства **AutoUpdate**.

### Odd, Even

Свойства **Odd** и **Even** класса **AFE5832** являются ссылками на класс **AFE5832Die**. Описание этого класса представлено ниже. При изменении свойств класса **AFE5832Die** будет автоматически вызван метод **Configure** класса **AFE5832**, если свойство **AutoUpdate** класса **AFE5832** равно **true**.

*Таблица 8 Свойства и методы класса AFE5832Die*

| Название | Тип | Описание |
|---|---|---|
| **LpfCutoffFreq** | Enum | Частота среза фильтра низких частот блока LNA |
| **HpfCutoffFreq** | Enum | Частота среза фильтра высоких частот блока LNA |
| **DtgcGain** | Double | Коэффициент усиления цифрового TGC, дБ |
| **EnableLnaHpf** | Boolean | Признак подключения фильтра высоких частот блока LNA |
| **LowPowerMode** | Boolean | Признак включения режима с пониженным энергопотреблением блока VCA |
| **EnableDtgcAttenuator** | Boolean | Признак подключения аттенюатора в блоке цифрового TGC |

### LpfCutoffFreq

Свойство **LpfCutoffFreq** класса **AFE5832Die** определяет частоту среза фильтра низких частот блока LNA, является перечисляемым свойством и ему можно присвоить значения: **_10_MHz**, **_15_MHz**, **_20_MHz**, **_30_MHz**. Если включен режим с пониженным энергопотреблением (свойство **LowPowerMode** класса **AFE5832Die** равно **true**), то значения частот нужно делить на два.

### HpfCutoffFreq

Свойство **HpfCutoffFreq** класса **AFE5832Die** определяет частоту среза фильтра высоких частот блока LNA, является перечисляемым свойством и ему можно присвоить значения: **_75_kHz**, **_150_kHz**.

### DtgcGain

Свойство **DtgcGain** класса **AFE5832Die** определяет коэффициент усиления цифрового TGC в децибелах.

### EnableLnaHpf

Свойство **EnableLnaHpf** класса **AFE5832Die** подключает (**true**) или отключает (**false**) фильтр высоких частот в блоке LNA.

### LowPowerMode

Свойство **LowPowerMode** класса **AFE5832Die** включает (**true**) или выключает (**false**) режим с пониженным энергопотреблением блока VCA.

### EnableDtgcAttenuator

Свойство **EnableDtgcAttenuator** класса **AFE5832Die** подключает (**true**) или отключает (**false**) аттенюатор в блоке цифрового TGC.

## Класс AFE5832LP

*Таблица 9 Свойства и методы класса AFE5832LP*

| Название | Тип | Описание |
|---|---|---|
| **Configure** | Метод | Передаёт настройки из свойств класса в устройство по системной шине |
| **ConfiguredDevic-esMask** | Unsigned, 32 bit | Маска устройств для настройки |
| **ConfiguredAdcMask** | Unsigned, 32 bit | Маска чипов АЦП для настройки |
| **HpfCornerFreq** | Enum | Крутизна фильтра верхних частот |
| **LpfCutoffFreqs** | Enum | Частота среза фильтра низких частот |

| | | |
|---|---|---|
| **PgaGainOdd-EqualEven** | Enum | Коэффициент усиления PGA |
| **LnaGainOdd, Even** | Enum | Коэффициент усиления LNA |
| **LowPowerMode** | Boolean | Признак энергосберегающего режима |
| **LowLatencyEnable** | Boolean | Признак режима с низкой задержкой сигнала и отключенной цифровой обработкой |
| **Attenuator** | Double | Коэффициент ослабления цифрового аттенюатора от 0 до 36 дБ |
| **AutoUpdate** | Boolean | Признак автоматической передачи настроек в устройство при изменении свойств класса |

### Configure

Метод **Configure** класса **AFE5832LP** передаёт настройки, рассчитанные из свойств класса **AFE5832LP** в устройство по системной шине, а также присваивает значение **true** свойству **AutoUpdate**.

### ConfiguredDevicesMask

Свойство **ConfiguredDevicesMask** класса **AFE5832LP** задаёт маску устройств для настройки. Каждый бит маски соответствует одному устройству: бит 0 – устройству с Id = 0, бит 1 – устройству с Id = 1 и т.д., где Id – идентификатор устройства на системной шине. При изменении маски настройки не передаются автоматически в устройство, но новая маска учитывается при изменении других свойств. Так если какому-либо свойству будет присвоено такое же значение, то если маска не была изменена, то настройки не будут переданы в устройство, а если была изменена – то будут. Количество подключенных устройств можно прочитать из свойства **DevicesCount** диспетчера устройств.

### ConfiguredAdcMask

Свойство **ConfiguredAdcMask** класса **AFE5832** задаёт маску чипов АЦП для настройки. Каждый бит маски соответствует одному чипу: бит 0 – чипу №1, бит 1 – чипу №2 и т.д. При изменении маски настройки не передаются автоматически в устройство, но новая маска учитывается при изменении других свойств. Так если какому-либо свойству будет присвоено такое же значение, то если маска не была изменена, то настройки не будут переданы в устройство, а если была изменена – то будут.

### HpfCornerFreq

Свойство **HpfCornerFreq** класса **AFE5832LP** определяет крутизну фильтра высоких частот, является перечисляемым свойством и ему можно присвоить значения: **_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

Свойство **LpfCutoffFreq** класса **AFE5832LP** определяет частоту среза фильтра низких частот блока, является перечисляемым свойством и ему можно присвоить значения: **_10_MHz**, **_15_MHz**, **_20_MHz**, **_25_MHz**. Если включен режим с пониженным энергопотреблением (свойство **LowPowerMode** класса **AFE5832LP** равно **true**), то значение **_25_MHz** будет соответствовать частоте 20 МГЦ, а остальные значения будут соответствовать указанным в них частотам.

### PgaGain

Свойство **PgaGain** класса **AFE5832LP** определяет усиление блока PGA, является перечисляемым свойством и ему можно присвоить значения: **_21_dB, _24_dB, _27_dB**.

### LnaGain

Свойство **LnaGain** класса **AFE5832LP** определяет усиление блока LNA, является перечисляемым свойством и ему можно присвоить значения: **_15_dB, _18_dB, _21_dB**.

### LowPowerMode

Свойство **LowPowerMode** класса **AFE5832LP** включает (**true**) или выключает (**false**) режим с пониженным энергопотреблением.

### LowLatencyEnable

Свойство **LowLatencyEnable** класса **AFE5832LP** включает (**true**) или выключает (**false**) режим с низкой задержкой сигнала и отключенной цифровой обработкой.

### Attenuator

Свойство **Attenuator** класса **AFE5832LP** задаёт коэффициент ослабления цифрового аттенюатора в децибелах в диапазое от 0 до 36 с шагом 0.125 дБ.

### AutoUpdate

Свойство **AutoUpdate** класса **AFE5832LP** разрешает или запрещает автоматическую передачу настроек в устройство при изменении свойств класса **AFE5832LP**. Если значение свойства равно **true**, то при записи нового значения в какое-либо из свойств класса **AFE5832LP**, обновлённые настройки автоматически передаются в устройство. Если значение свойство равно **false**, то можно присвоить новые значения нескольких свойствам класса **AFE5832LP**, а потом вызвать метод **Configure**, который передаст настройки в устройство и восстановит значение **true** для свойства **AutoUpdate**.

## Формат файла данных

Данные АЦП сохраняются в бинарном RAW файле. Файл состоит из заголовка файла (Таблица 10) и N кадров данных АЦП. Количество кадров данных записано в заголовке файлов. Каждый кадр также содержит заголовок и данные (Таблица 11). Каждый кадр соответствует одному устройству, кадры записываются в файл строго последовательно от устройства с меньшим порядковым номером к устройству с большим порядковым номером. Порядковый номер устройства определяется его положением в цепи устройств, соединённых HDMI кабелем. Ведущее устройство имеет порядковый номер 0, а последний порядковый номер имеет устройство, подключенное последним. Чтобы определить, какому устройству принадлежит конкретный кадр следует проанализировать поле **Boards Mask** в заголовке файла.

Данные в кадре расположены последовательно – сначала 1-й отсчёт канала №1, затем 1-й отсчёт канала №2 и так до последнего канала, затем 2-й отсчёт канала №1, затем 2-й отсчёт канала №2 и так до последнего канала. Далее последовательность повторяется до последнего отсчёта последнего канала. В заголовке файла указано общее число каналов, записанных в файле, и одинаковое для всех кадров данных число отсчётов данных на канал АЦП. А в заголовке кадра указано число каналов для соответствующего устройства. Число каналов определяется числом разрешенных АЦП и числом каналов на одно АЦП. Для того, чтобы определить какой канал относится к какому АЦП, следует проанализировать маску разрешенных АЦП **ADC Mask** для текущего кадра. Каналы всегда расположены в порядке возрастания по номерам АЦП.

*Таблица 10 Формат заголовка файла данных*

| Поле | Тип | Описание |
|---|---|---|
| **Format version** | Double | Версия формата файла |
| **Number of frames** | Integer, 32 bit | Количество кадров данных в файле |
| **Header length** | Integer, 32 bit | Размер заголовка файла в байтах |
| **Frame length** | Integer, 32 bit | Размер кадра данных в байтах |
| **Sample rate** | Integer, 32 bit | Частота дискретизации данных в герцах |
| **Number of channels** | Integer, 32 bit | Количество каналов АЦП в файле |
| **Number of samples** | Integer, 32 bit | Количество отсчётов данных на канал АЦП в каждом кадре |
| **Number of boards** | Integer,32 bit | Количество устройств, данные с которых записаны в файле |
| **Boards mask** | Unsigned, 32 bit | Маска устройств в порядке от ведущего устройства (бит №0) до последнего ведомого (бит № 31). Если данные с устройства с порядковым номером N, начиная от ведущего устройства с номером 0, записаны в файл, то бит N равен 1, в противном случае 0 |

Таблица 11 Формат кадра данных

| Поле | Тип | Описание |
|---|---|---|
| **Number of channels** | Integer, 32 bit | Количество каналов АЦП в кадре |
| **Number of samples** | Integer, 32 bit | Количество отсчётов на канал |
| **Sample rate** | Integer, 32 bit | Частота дискретизации данных в герцах |
| **Trigger source** | Integer, 32 bit | Маска входа триггера данных кадра. Бит 0 маски соответствует входу №1, бит 1 маски – входу №2 и т.д. |
| **Trigger time** | Double | Отсчёт времени сигнала триггера данных кадра в миллисекундах |
| **Frame number** | Unsigned,32 bit | Числовая метка кадра |
| **ADC Mask** | Unsigned,32 bit | Маска разрешенных чипов АЦП. Каждый бит маски соответствует одному чипу АЦП, бит 0 – чипу № 1, бит 1 – чипу № 2 и т.д. Если значение бита равно 1, то АЦП разрешено для сбора данных, в противном случае – запрещено |
| **ADC data** | Массив Integer, 16 bit | Массив данных АЦП, данные расположены последовательно по каналам: сначала все каналы для отсчёта №1, затем все каналы для отсчёта №2 и т.д. |