KingstVIS Analyzer Developer Guide

Qingdao Kingst Electronics Co., Ltd

Website：www.qdkingst.com

E-mail：service@qdkingst.com

Contents

Ⅰ. Setting up an Analyzer Project

1. General description

The KingstVIS supports custom protocol analyzers, using C++ for development.

This document is divided into two parts. The first part is the steps of setting up a custom analyzer project, and introduces how to generate dynamic link library file of the analyzer on Windows, Linux and Mac OS. The second part is how to write analyzer code, introduce the usage of basic class and its member functions.

2. Setting up project

Before starting the next step, you should make sure that you have KingstVIS_Analyzer_SDK provided by us.

There are *inc*, *lib* folders and the sample *SerialAnalyzer* in the root of *KingstVIS_Analyzer_SDK* folder. Base class header file is stored in the *inc* folder. There are four subfolders of *Win32*, *Win64*, *Linux* and *Mac* in the *lib* folder, The 32-bit and 64-bit *Analizer.lib* and *Analyzer.dll* are stored in the *Win32* and *Win64* folders. *libAnalyzer.so* is stored in *Linux*. *libAnalyzer.dylib* is stored in *Mac*. In the *SerialAnalyzer* folder, the project file and source code of UART are stored, and there are four subfolders: *src, vs2013*, *Linux* and *Mac.* The source code for the Analyzer in the *src* folder. The project file builded by VS2013 is stored in the *vs2013* folder, you can use the project file in Windows to generate the analyzer *.dll* file. In Linux, you can use *makefile* file to generate the *.so* file, which stored in the *Linux* folder. In Mac，you can use *makefile* generate the *.dylib* file, which stored in the *Mac* folder.

2.1 Windows

1. Take the example of generating *.dll* file which use to analyze the SPI protocol, you can modify the project file to get the SPI analyzer from the sample *SerialAnalyzer.* If you want to keep the sample, you can copy it in the same directory, and then modify it on the *SerialAnalyzer-copy*.

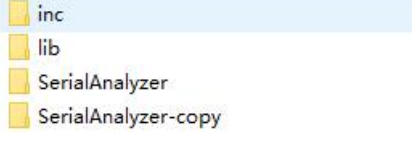


Fig.1

2. Refer to the sample, change the name of the folder to *SpiAnalyzer*. Open this folder and replace the source file in the *src* folder with the source file used to analyze SPI. The creation of the specific source file will be explained in the next chapter. Open *vs2013* Folder, modify the file name to *SiAnalyzer.vcxproj*.

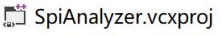


Fig.2

3. Use Visual Studio 2013 (or later) to open the project file *SpiAnalyzer.vcxproj*. If the file type in the open dialog box does not have **.vcxproj* and cannot open the project, because your Visual Studio does not have VC++ components. Please run the Visual Studio Installer again to add VC++ components.

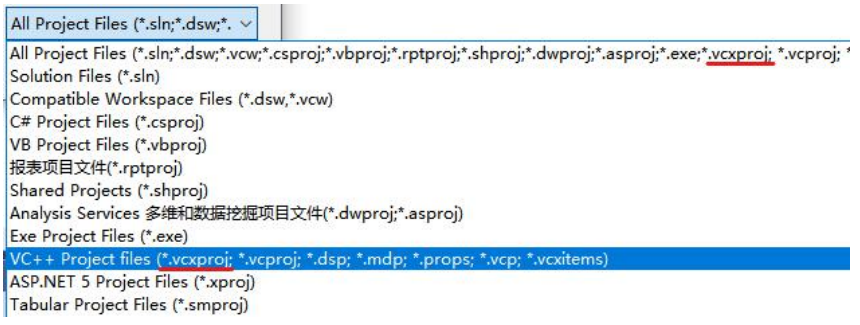


Fig.3

After opening the project file, remove all the original *.cpp* and *.h* files.

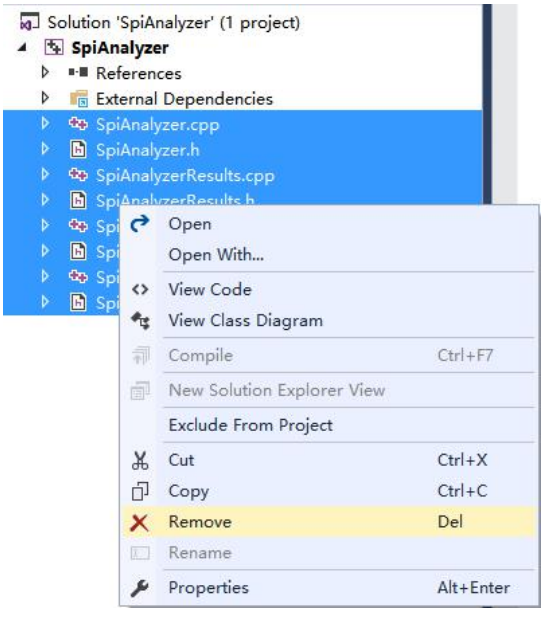


Fig.4

4. Right click on the project name, choose Add -> Existing Item as shown in Figure 5. Open *src* folder, select all *.cpp* and *.h* files you need to add, and add them.

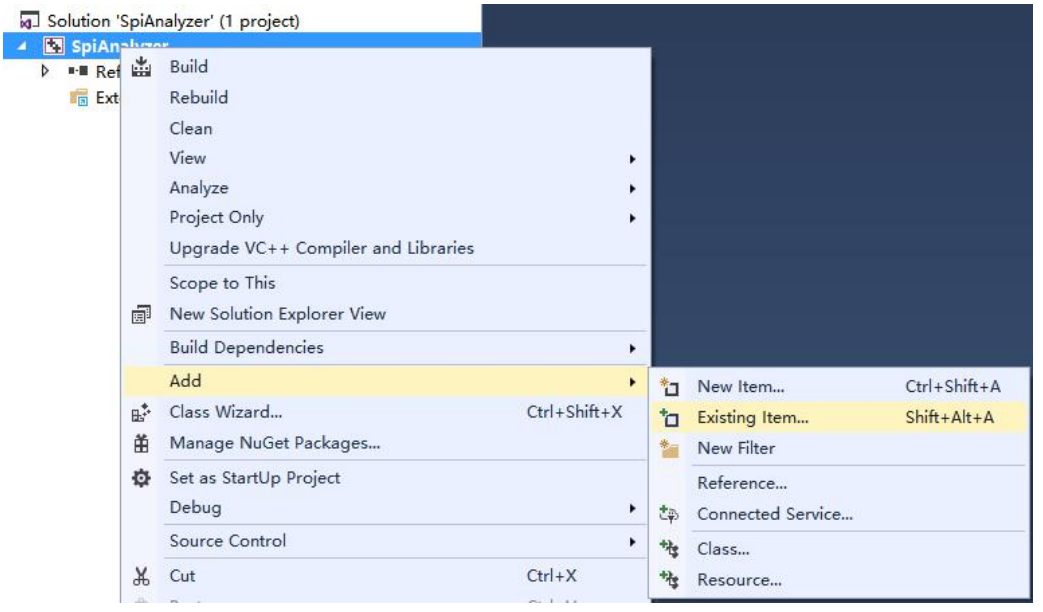


Fig.5

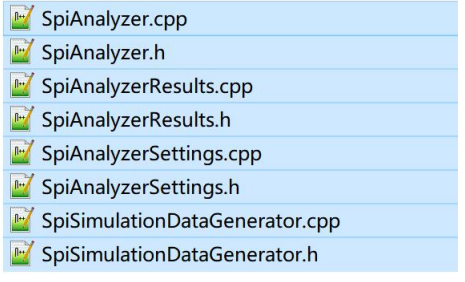


Fig.6

5. After completing the above steps, the project is set up. You can generate the required *.dll* file. Generate 32-bit or 64-bit *.dll* file in Release mode, depending on the OS you are using. The generated *.dll* file name can be set in Visual Studio through the menu Project -> [project name] Properties -> Configuration Properties -> General -> Target Name, as shown in Figure 7. If no change is made, it will be the same as the project name. If you use a later version of Visual Studio, you also need to modify the Platform Toolset.

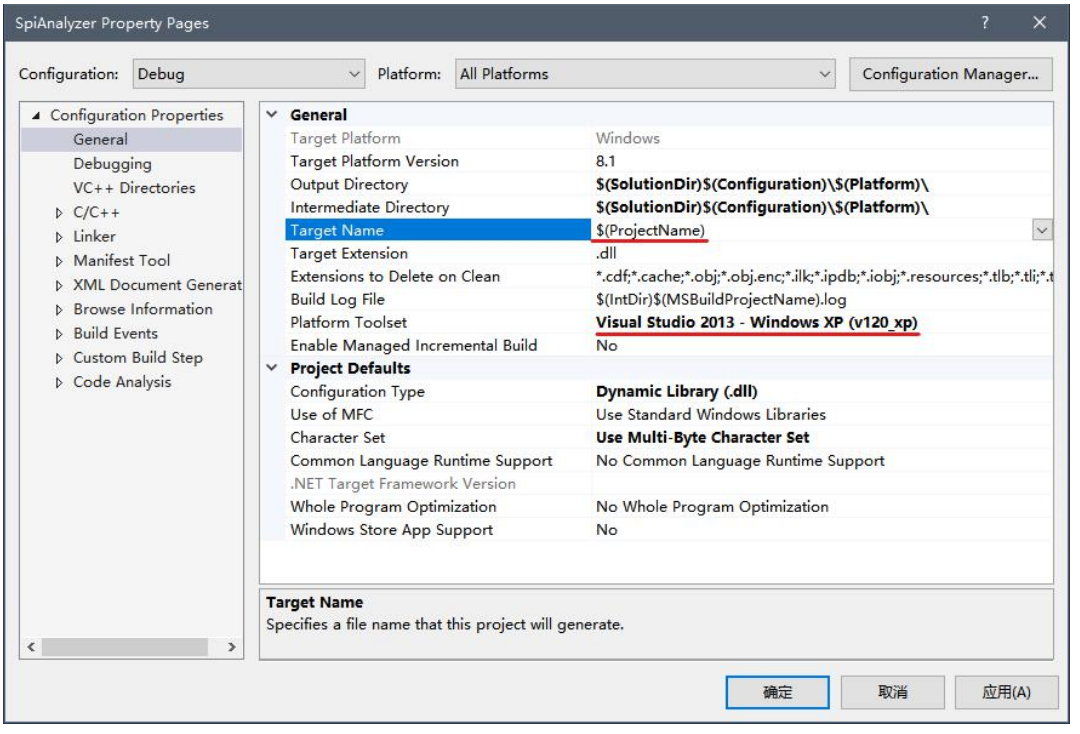


Fig.7

When generating 32-bit file, it should be configured as Release + Win32, as shown in Figure 8. When generating 64-bit file, it should be configured as Release + x64, as shown in Figure 9.

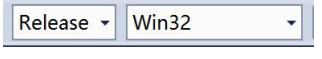


Fig.8

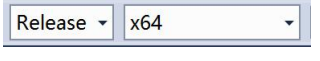


Fig.9

6. The generated 32-bit *.dll* file is located in the "*Release**Win32*\" directory and the 64-bit file is located in the "*Release**x64*\".

7. Copy the *.dll* file to the *Analyzer* directory of the KingstVIS installation directory. Restart the KingstVIS, and your custom analyzer will appear in the analyzer list.

2.2 Linux

1. Take the example of generating *.so* file which use to analyze the SPI protocol. you can modify the project file to get the SPI analyzer from the sample *SerialAnalyzer.* If you want to keep the sample, you can copy it in the same directory, and then modify it on the copied folder.

2. Refer to the sample, change the name of the folder to *SpiAnalyzer*. Open this folder and replace the source file in the *src* folder with the source file used to analyze SPI. The creation of the specific source file will be explained in the next chapter. Open *Linux* Folder, open the *makefile*, and change *libSerial.so*

to the name you want, here to *libSPI.so.*

3. Open the terminal and go to the *Linux* directory, enter the command-"make", and the *libSPI.so* file will be generated in the current directory.

4. Copy the *libSPI.so* file to the *Analyzer* directory of the KingstVIS installation directory. Restart the KingstVIS, and your custom analyzer will appear in the analyzer list.

2.3 Mac

1. Take the example of generating *.so* file which use to analyze the SPI protocol. you can modify the project file to get the SPI analyzer from the sample *SerialAnalyzer.* If you want to keep the sample, you can copy it in the same directory, and then modify it on the copied folder.

2. Refer to the sample, change the name of the folder to *SpiAnalyzer*. Open this folder and replace the source file in the *src* folder with the source file used to analyze SPI. The creation of the specific source file will be explained in the next chapter. Open *Linux* Folder, open the *makefile*, and change *libSerial.dylib* to the name you want, here to *libSPI.dylib.*

3. Open the terminal and go to the *Linux* directory, enter the command-"make", and the *libSPI.dylib* file will be generated in the current directory.

4. Copy the *libSPI.dylib* file to the *Contents/Resources/Analyzer/* directory of the KingstVIS package directory. Restart the KingstVIS, and your custom analyzer will appear in the analyzer list.

5. It may happen that the generated *.dylib* file cannot be loaded because the dependent library cannot be found. You can use the following command to modify the dependent library path.

```
install_name_tool -change "libAnalyzer.dylib" "@executable_path/libAnalyzer.dylib" "libSPI.dylib"
```

II. Writing your Analyzer's Code

There are 4 *.cpp* and 4 *.h* files that you will implement to create your analyzer. You can modify the source files of the samples in the SDK package to simplify the operation.

Conceptually, the analyzer can be broken into 4 main parts – the 4 c++ files. Working on them in a particular order is highly recommended, and this document describes the procedure in this order.

First you'll work on the *AnalyzerSettings*-derived class. You'll define the settings your analyzer needs, and create interfaces that'll allow the KingstVIS to display a GUI for the settings. You'll also implement serialization for these settings so they can be saved and recalled from disk.

Next you implement the *SimulationDataGenerator* class. Here you'll generate simulated data that can be later to test your analyzer, or provide an example of what your analyzer expects.

Third you'll create your *AnalyzerResults*-derived class. This class translates saved results into text for a variety of uses. Here you'll start thinking about the format your results will be saved in. You probably will revisit your file after implementing your analyzer.

Lastly, you'll implement your *Analyzer*-derived class. The main thing you'll do here is translate data streams into results, based on your protocol.

Let's get started!

To write *Serial* protocol (such as UART) analyzer, for example, first create 4 *.cpp* and 4 *.h* files. The four *.cpp* files are *SerialAnalyzer.cpp, SerialAnalyzerResults.cpp, SerialAnalyzerSettings.cpp, SerialSimulationDataGenerator.cpp*, the four *.h* files are *SerialAnalyzer.h, SerialAnalyzerResults.h, SerialAnalyzerSettings.h, SerialSimulationDataGenerator.h*. Let these files be empty first, and then implement them one by one later. When writing your own analyzer, it is also recommended to create the file according to this naming method.

1. Analyzer Settings

Add the created 4 *.cpp* files and 4 *.h* files to the created project, so that a project on the analyzer is completed, and the next step is to complete the 8 empty files one by one.

1.1 {YourName}AnalyzerSettings.h

The first thing to implement is a derived class of the Analyzer Settings class. The code to implement this derived class is written in *SerialAnalyzerSettings.cpp* and *SerialAnalyzerSettings.h*, the name of this derived class could be named *SerialAnalyzerSettings*, This class must inherit from *AnalyzerSettings*, and should include the *AnalyzerSettings.h* header file.

The code of *SerialAnalyzerSettings.h* shown as follows：

```
#ifndef SERIAL_ANALYZER_SETTINGS
#define SERIAL_ANALYZER_SETTINGS

#include <AnalyzerSettings.h>

class SerialAnalyzerSettings : public AnalyzerSettings
{
public:
     SerialAnalyzerSettings();
     virtual ~SerialAnalyzerSettings();

     virtual bool SetSettingsFromInterfaces();
     void UpdateInterfacesFromSettings();
     virtual void LoadSettings(const char *settings);
     virtual const char *SaveSettings();
};

#endif //SERIAL_ANALYZER_SETTINGS
```

In addition, your header will define two sets of variables:

1.1.1 User-modifiable settings

This will always include at least one variable of the type Channel – so the user can specify which input channel to use. This cannot be hard coded, and must be exposed as a setting. (Channel isn't just an index, it also specifies which Logic device the channel is from). Other possible settings depend on your protocol, and might include:

- Bit rate
- Bits per transfer
- Bit ordering (MSB first, LSB first)
- Clock edge (rising, falling) to use
- Enable line polarity

The variable types can be whatever you like - std::string, double, int, enum, etc. Note that these variables will need to be serialized (saved for later, to a file) so when in doubt, stick to simple types (rather than custom classes or structs). The SDK provides a means to serialize and store your variables.

The setting variables added to the *SerialAnalyserSettings* class are as follows：

```
     Channel mInputChannel;
     U32 mBitRate;
     U32 mBitsPerTransfer;
     AnalyzerEnums::ShiftOrder mShiftOrder;
     double mStopBits;
     AnalyzerEnums::Parity mParity;
     bool mInverted;
```

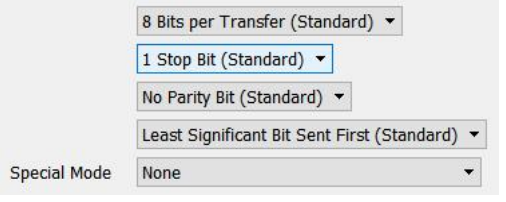
```
    bool mUseAutobaud;
    SerialAnalyzerEnums::Mode mSerialMode;
```

1.1.2 Analyzer Settings Interfaces

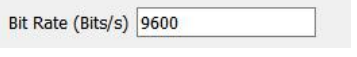
One of the services the Analyzer SDK provides is a means for users to edit your settings, with a GUI, with minimal work on your part. To make this possible, each of your settings variables must have a corresponding interface object. Here are the avalible *AnalyzerSettingsInterface* types:

- AnalyzerSettingInterfaceChannelData: Used exclusivly for input channel selection.

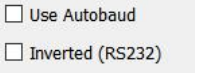
- AnalyzerSettingInterfaceNumberList: Used to provide a list of numerical options for the user to choose from. Note that this can be used to select from several enum types as well, as illustrated below. (Each dropdown below is implemented with its own interface object)



- AnalyzerSettingInterfaceInteger: Allows a user to type an integer into a box.

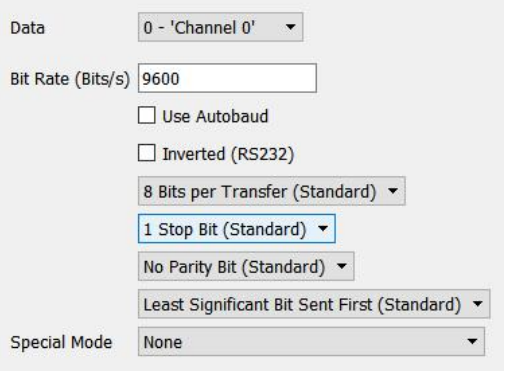


- AnalyzerSettingInterfaceText: Allows a user to enter some text into a textbox.
- AnalyzerSettingInterfaceBool: Provides the user with a checkbox.



AnalyzerSettingsInterface types should be declared as pointers. (We're using the std::auto_ptrtype, which largly acts like a standard (raw) pointer. It's a simple form of what's called a "smart pointer" and it automaticly calls deleteon its contents when it goes out of scope.)

For Serial, the GUI setting interface we want to implement is shown in the following figure.



So you should add the following code to the SerialAnalyserSettings class:

```
    std::auto_ptr< AnalyzerSettingInterfaceChannel > mInputChannelInterface;
    std::auto_ptr< AnalyzerSettingInterfaceInteger > mBitRateInterface;
    std::auto_ptr< AnalyzerSettingInterfaceBool > mInvertedInterface;
    std::auto_ptr< AnalyzerSettingInterfaceBool > mUseAutobaudInterface;
    std::auto_ptr< AnalyzerSettingInterfaceNumberList > mBitsPerTransferInterface;
    std::auto_ptr< AnalyzerSettingInterfaceNumberList > mShiftOrderInterface;
    std::auto_ptr< AnalyzerSettingInterfaceNumberList > mStopBitsInterface;
    std::auto_ptr< AnalyzerSettingInterfaceNumberList > mParityInterface;
    std::auto_ptr< AnalyzerSettingInterfaceNumberList > mSerialModeInterface;
```

1.2 {YourName}AnalyzerSettings.cpp

After completing the contents of the *SerialAnalyserSettings.h* file, you need to complete the *SerialAnalyserSettings.cpp* file, which is to implement the constructors, destructors, etc. in the *SerialAnalyserSettings* class.

1.2.1 The Constructor

First of all, we need to complete the compilation of the constructor. In the constructor, we need to initialize the variable. For the *SerialAnalyzer*, the code for initializing the variable is written as follows:

```
SerialAnalyzerSettings::SerialAnalyzerSettings()
:    mInputChannel(UNDEFINED_CHANNEL),
     mBitRate(9600),
     mBitsPerTransfer(8),
     mStopBits(1.0),
     mParity(AnalyzerEnums::None),
     mShiftOrder(AnalyzerEnums::LsbFirst),
     mInverted(false),
     mUseAutobaud(false),
     mSerialMode(SerialAnalyzerEnums::Normal)
{
}
```

After initializing the variable, the *reset* function is called to complete the creation of the interface.

AnalyzerSettingInterfaceChannelData

```
    mInputChannelInterface.reset(new AnalyzerSettingInterfaceChannel());
```

Next, we call the *SetTitleAndTooltip* function. The title will appear to the left of the input element. Note that often times you won't need a title, but you should use one for *Channels*. The tooltip shows up when hovering over the input element.

```
    void SetTitleAndTooltip(const char* title, const char* tooltip)
```

For example, for the serial channel setting interface, the following settings can be made.

```
    mInputChannelInterface->SetTitleAndTooltip("Data", "Standard Async Serial");
```

The interface name is Data and the prompt message is Standard Async Serial。

Next, we use the SetChannel function to set the channel.

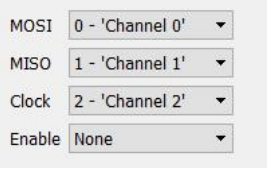
```
void SetChannel(const Channel& channel);
mInputChannelInterface->SetChannel(mInputChannel);
```

And we can use *GetChannel* function to get channel number.

```
Channel GetChannel();
```

Sometimes the channel can be set to None, means that it can not be connected to one channel of the logic analyzer. This situation will only occurs under multiple channels, and an input channel must be set when the protocol is single line.

For example, when parsing SPI protocol, set channel selection, as shown in the following figure.



This means Enable may not be connected to the logic analyzer.

Whether the channel can be set to *None* needs to be set using the *SetSelectionOfNoneIsAllowed* function.

```
void SetSelectionOfNoneIsAllowed(bool is_allowed);
```

If *is_allowed* is true, the channel can be set to None, and if it is false, an input must be set. The default is false, which means that each channel must have one input channel.

You can get whether the channel can be set to *None* using the *getSelectionOfNoneISallowed* function.

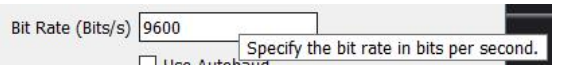
```
void SetSelectionOfNoneIsAllowed(bool is_allowed);
```

AnalyzerSettingInterfaceInteger

```
mBitRateInterface.reset(new AnalyzerSettingInterfaceInteger());
```

Use the *SetTitleAndToolTip* function to set the interface name and prompt information

```
mBitRateInterface->SetTitleAndTooltip(
    "Bit Rate (Bits/s)", "Specify the bit rate in bits per second.");
```



This number to be entered must be an integer. You can set the range of input numbers using the *SetMax* and *SetMin* functions.

```
void SetMax(int max);
void SetMin(int min);
mBitRateInterface->SetMax(100000000);
```

```
mBitRateInterface->SetMin(1);
```

With the above settings, the input data range is limited to an integer between 1 to 10000000.

Setting the input data can be achieved using SetInteger.

```
void SetInteger(int integer);
mBitRateInterface->SetInteger(mBitRate);
```

The input data can be obtained using *GetInteger*.

```
int GetInteger();
```

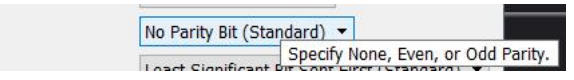
AnalyzerSettingInterfaceNumberList

Explains the use of the Analyzer *SettingInterfaceNumberList* class with how to set parity in the *SerialAnalyzer*.

```
mParityInterface.reset(new AnalyzerSettingInterfaceNumberList());
```

Then set the interface name and prompt information using the *SetTitleAndToolTip* function.

```
mParityInterface->SetTitleAndTooltip("", "Specify None, Even, or Odd Parity.");
```



Add a selection to the interface drop-down list using the AddNumber function.

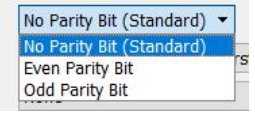
```
void AddNumber(double number, const char *str, const char *tooltip);
```

This function has three input parameters, *number* is associated with the selection and will not be displayed to the user. *str* is the name of the selection and will appear in the list. *tooltip* is a prompt message that will appear when the mouse hovers over the selection item.

```
mParityInterface->AddNumber(AnalyzerEnums::None, "No Parity Bit (Standard)", "");
mParityInterface->AddNumber(AnalyzerEnums::Even, "Even Parity Bit", "");
mParityInterface->AddNumber(AnalyzerEnums::Odd, "Odd Parity Bit", "");
```

When the selection item you want to set is selected in the list, the set number is saved using *SetNumber* function.

```
mParityInterface->SetNumber(mParity);
```



The value set by the *SetNumber* function can be obtained using the *GetNumber* function.

```
double GetNumber();
```

AnalyzerSettingInterfaceBool

Here is an example of setting whether the *SerialAnalyzer* enables the automatic baud rate to show how to use the *AnalyzerSettingInterfaceBool* class, which is also to first create a pointer to the *AnalyzerSettingInterFacebool* and call the *reset* function.

```
mUseAutobaudInterface.reset(new AnalyzerSettingInterfaceBool());
```

Set the interface name and prompt information.

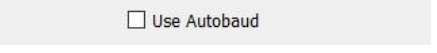
```
mUseAutobaudInterface->SetTitleAndTooltip("", "Automatically find the minimum pulse width
    and calculate the baud rate according to this pulse width.");
```

You can use the SetCheckBoxText function to give the selection box a name, and the name is displayed on the right side of the selection box.

```
void SetCheckBoxText(const char* text);
mUseAutobaudInterface->SetCheckBoxText("Use Autobaud");
```

You can use the SetValue function to enable this radio box setting item.

```
void SetValue(bool value);
mUseAutobaudInterface->SetValue(mUseAutobaud);
```



The *GetValue* function can be used to obtain whether to enable the selection box setting item.

```
bool GetValue();
```

After creating our interfaces (with new), giving them a titles, settings their values, and specifying their allowed options, we need to expose them to the API. We do that with function AddInterface.

```
void AddInterface(AnalyzerSettingInterface* analyzer_setting_interface);
```

The following parameters need to be passed.

```
AddInterface(mInputChannelInterface.get());
AddInterface(mBitRateInterface.get());
AddInterface(mUseAutobaudInterface.get());
AddInterface(mInvertedInterface.get());
AddInterface(mBitsPerTransferInterface.get());
AddInterface(mStopBitsInterface.get());
AddInterface(mParityInterface.get());
AddInterface(mShiftOrderInterface.get());
AddInterface(mSerialModeInterface.get());
```

Specifing the export options

Analyzers can offer more than one export type. For example txt or csv, or even a wav file or bitmap. If these need special settings, they can be specified as analyzer variables/interfaces as we've discussed.

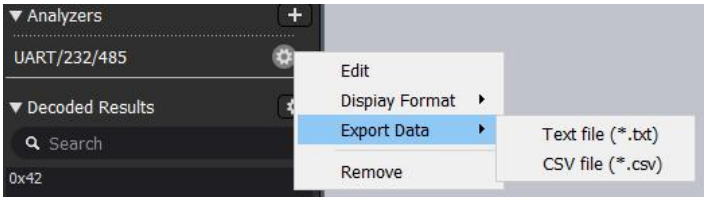
Export options are assigned an ID. Later, when your function for generating export data is called, this ID will be provided. There are two functions you'll need to call to specfiy an export type. Be sure to specify at least one export type (tyically text/csv).

```
void AddExportOption(U32 user_id, const char *menu_text);
void AddExportExtension(U32 user_id, const char *extension_description, const char *extension);
```

user_id means a specify ID, extension_description means the output file's type that we want, extension

means suffix.

```
AddExportOption(0, "Export as text/csv file");
AddExportExtension(0, "Text file", "txt");
AddExportExtension(0, "CSV file", "csv");
```



Specifying which channels are in use

The analyzer must indicate which channel(s) it is using. This is done with the *AddChannel* function. Every time the channel changes (such as when the user changes the channel) the reported channel must be updated. To clear any previous channels that have been set, call *ClearChannels*.

```
void ClearChannels();
void AddChannel(Channel &channel, const char *channel_label, bool is_used);
ClearChannels();
AddChannel(mInputChannel, CHANNEL_NAME, false);
```

Note that in the constructor, we have set is_used to false. This is because by defualt our channel is set to *UNDEFINED_CHANNEL*. After the user has set the channel to something other that *UNDEFINED_CHANNEL*, we would specify true. It would always be true, unless the channel was optional, in which case you will need to examine the channel value, and specify false if the channel is set to *UNDEFINED_CHANNEL*. We'll discuss this later as it comes up.

1.2.2 The Destructor

Generally you won't need to do anything in your *AnalyzerSettings*-derived class's destructor. However, if you are using standard (raw) pointers for your settings interfaces, you'll need to delete them here.

1.2.3 bool {YourName}AnalyzerSettings::SetSettingsFromInterfaces()

As the name implies, in this function we will copy the values saved in our interface objects to our settings variables. This function will be called if the user updates the settings.

We can also examine the values saved in the interface (the user's selections) and choose to reject combinations we don't want to allow. If you want to reject a particular selection, do not assign the values in the interfaces to your settings variables – use temporary variables so you can choose not to assign them at the last moment. To reject a user's selections, return *false*; otherwise return *true*. If you return *false* (reject the user's settings), you also need to call *SetErrorText* to indicate why. This will be presented to the user in a popup dialog.

```
void SetErrorText(const char *error_text);
```

For example, when using more than one channel, you would typically want to make sure that all the

channels are different. You can use the *AnalyzerHelpers::DoChannelsOverlap* function to make that easier if you like.

For your analyzer, it's quite possible that all possible user selections are valid. In that case you can ignore the above.

After assigning the interface values to your settings variables, you also need to update the channel(s) the analyzer will report as being used. Below is an example from *SerialAnalyzerSettings*.

```
bool SerialAnalyzerSettings::SetSettingsFromInterfaces()
{
     if (AnalyzerEnums::Parity(U32(mParityInterface->GetNumber())) != AnalyzerEnums::None)
         if (SerialAnalyzerEnums::Mode(U32( mSerialModeInterface->GetNumber())) !=
             SerialAnalyzerEnums::Normal) {
             SetErrorText("Sorry, but we don't support using parity at the same time as MP mode.");
             return false;
         }
     mInputChannel = mInputChannelInterface->GetChannel();
     mBitRate = mBitRateInterface->GetInteger();
     mBitsPerTransfer = U32(mBitsPerTransferInterface->GetNumber());
     mStopBits = mStopBitsInterface->GetNumber();
     mParity = AnalyzerEnums::Parity(U32(mParityInterface->GetNumber()));
     mShiftOrder = AnalyzerEnums::ShiftOrder(U32(mShiftOrderInterface->GetNumber()));
     mInverted = mInvertedInterface->GetValue();
     mUseAutobaud = mUseAutobaudInterface->GetValue();
     mSerialMode = SerialAnalyzerEnums::Mode(U32(mSerialModeInterface->GetNumber()));

     ClearChannels();
     AddChannel(mInputChannel, CHANNEL_NAME, true);

     return true;
}
```

1.2.4 {YourName}AnalyzerSettings::UpdateInterfacesFromSettings()

UpdateInterfacesFromSettings goes in the opposite direction. In this function, update all your interfaces with the values from your settings variables. Below is an example from *SerialAnalyzerSettings*.

```
void SerialAnalyzerSettings::UpdateInterfacesFromSettings()
{
     mInputChannelInterface->SetChannel(mInputChannel);
     mBitRateInterface->SetInteger(mBitRate);
     mBitsPerTransferInterface->SetNumber(mBitsPerTransfer);
     mStopBitsInterface->SetNumber(mStopBits);
     mParityInterface->SetNumber(mParity);
     mShiftOrderInterface->SetNumber(mShiftOrder);
     mInvertedInterface->SetValue(mInverted);
```

```
    mUseAutobaudInterface->SetValue(mUseAutobaud);
    mSerialModeInterface->SetNumber(mSerialMode);
}
```

1.2.5 void {YourName}AnalyzerSettings::LoadSettings(const char *settings)

In the last to functions of your *AnalyzerSettings*-derived class, you'll implement serialization (persistence) of your settings. It's pretty straightforward.

Your settings are saved in, and loaded from, a single string. You can technically serialize all of your variables into a string anyway you like, including third part libraries like boost, but to keep things simple we provided a mechanism to serialize your variables. We'll discuss that here.

First, you'll need a *SimpleArchive* object. This will perform serialization for us. Use *SetString* to provide the archive with our settings string. This string is passed in as a parameter to *LoadSettings*.

```
struct SimpleArchiveData;
class LOGICAPI SimpleArchive
{
public:
    SimpleArchive();
    ~SimpleArchive();

    void SetString(const char *archive_string);
    const char *GetString();

    bool operator<<(U64 data);
    bool operator<<(U32 data);
    bool operator<<(S64 data);
    bool operator<<(S32 data);
    bool operator<<(double data);
    bool operator<<(bool data);
    bool operator<<(const char *data);
    bool operator<<(Channel &data);

    bool operator>>(U64 &data);
    bool operator>>(U32 &data);
    bool operator>>(S64 &data);
    bool operator>>(S32 &data);
    bool operator>>(double &data);
    bool operator>>(bool &data);
    bool operator>>(char const **data);
    bool operator>>(Channel &data);

protected:
    struct SimpleArchiveData *mData;
```

```
};
```

Next we will use the archive to loaf all of our settings variables, using the overloaded >> operator.

Since our channel values may have changed, we will also need to update the channels we're reporting as using. We need to do this every times settings change.

Lastly, call *UpdateInterfacesFromSettings*. This will update all our interfaces to reflect the newly loaded values.

Below is an example from *SerialAnalzerSettings*.

```
void SerialAnalyzerSettings::LoadSettings(const char *settings)
{
     SimpleArchive text_archive;
     text_archive.SetString(settings);

     const char *name_string;
     text_archive >> &name_string;
     if (strcmp(name_string, "SerialAnalyzer") != 0)
         AnalyzerHelpers::Assert("SerialAnalyzer: Provided with a settings string that doesn't
                                  belong to us;");

     text_archive >> mInputChannel;
     text_archive >> mBitRate;
     text_archive >> mBitsPerTransfer;
     text_archive >> mStopBits;
     text_archive >> *(U32*)&mParity;
     text_archive >> *(U32*)&mShiftOrder;
     text_archive >> mInverted;

     bool use_autobaud;
     if (text_archive >> use_autobaud)
         mUseAutobaud = use_autobaud;

     SerialAnalyzerEnums::Mode mode;
     if (text_archive >> *(U32*)&mode)
         mSerialMode = mode;

     ClearChannels();
     AddChannel(mInputChannel, CHANNEL_NAME, true);

     UpdateInterfacesFromSettings();
}
```

1.2.6 void {YourName}AnalyzerSettings::SaveSettings()

Our last function will save all of our settings variables into a single string. We'll use *SimpleArchive* to serialize them.

The order in which we serialize our settings variables must be exactly the same order as we extract them, in *LoadSettings*.

When returning, use the *SetReturnString* function, as this will provide a pointer to a string that will not go out of scope when the function ends.

Bellow is an example from *SerialAnalyzerSettings*:

```
const char *SerialAnalyzerSettings::SaveSettings()
{
     SimpleArchive text_archive;

     text_archive << "KingstUartAnalyzer";
     text_archive << mInputChannel;
     text_archive << mBitRate;
     text_archive << mBitsPerTransfer;
     text_archive << mStopBits;
     text_archive << mParity;
     text_archive << mShiftOrder;
     text_archive << mInverted;
     text_archive << mUseAutobaud;
     text_archive << mSerialMode;

     return SetReturnString(text_archive.GetString());
}
```

2. SimulationDataGenerator

The next step after creating your *{YourName}AnalyzerSettings* files, is to create your *SimulationDataGenerator* class, and you need to complete *{YourName}SimulationDataGenerator.h* and *{YourName}SimulationDataGenerator.cpp*.

Your *SimulationDataGenerator* class provides simulated data so that you can test your analyzer against controlled, predictable waveforms. Generally you should make the simulated data match the user settings, so you can easily test under a variety of expected conditions. In addition, simulated data gives end users an example of what to expect when using your analyzer, as well as examples of what the waveforms should look like.

That said, fully implementing simulated data is not absolutely required to make an analyzer.

2.1 {YourName}SimulationDataGenerator.h

Besides the constructor and destructor, there are only two required functions, and two required

variables. Other functions and variables can be added, to help implement your simulated data. Here is an example starting point, from *SerialSimulationDataGenerator.h*.

```
#ifndef SERIAL_SIMULATION_DATA_GENERATOR
#define SERIAL_SIMULATION_DATA_GENERATOR

#include <AnalyzerHelpers.h>

class SerialAnalyzerSettings;

class SerialSimulationDataGenerator
{
public:
     SerialSimulationDataGenerator();
     ~SerialSimulationDataGenerator();

     void Initialize(U32 simulation_sample_rate, SerialAnalyzerSettings *settings);
     U32 GenerateSimulationData(U64 newest_sample_requested, U32 sample_rate,
                                SimulationChannelDescriptor **simulation_channels);

protected:
     SerialAnalyzerSettings *mSettings;
     U32 mSimulationSampleRateHz;
     BitState mBitLow;
     BitState mBitHigh;
     U64 mValue;

     U64 mMpModeAddressMask;
     U64 mMpModeDataMask;
     U64 mNumBitsMask;

protected: //Serial specific
     void CreateSerialByte(U64 value);
     ClockGenerator mClockGenerator;
     SimulationChannelDescriptor mSerialSimulationData;
};
#endif //UNIO_SIMULATION_DATA_GENERATOR
```

The key to the SimulationDataGenerator is the class *SimulationChannelDescriptor*. You will need one of these for every channel you will be simulated (serial, for example, only needs to simulate on one channel). When your *GenerateSimulationData* function is called, your job will be to generate additional simulated data, up to the amount requested. When complete, you provide the caller with a pointer to an array of your *SimulationChannelDescriptor* objects.

2.2 {YourName}SimulationDataGenerator.cpp

2.2.1 Constructor / Destructor

You may or may not need anything in your constructor or destructor. For now at least, leave them empty. At the time we're constructed, we really have no idea what the settings are or anything else, so there's not much we can do at this point.

2.2.2 void {YourName}SimulationDataGenerator::Initialize(U32 simulation_sample_rate, {YourName}AnalyzerSettings *settings)

This function provides you with the state of things as they are going to be when we start simulating. We'll need to save this information.

First, save *simulation_sample_rate* and settings to member variables. Notice that we now have a pointer to our *AnalyzerSettings*-derived class. This is good, now we know what all the settings will be for our simulation – which channel(s) it will be on, as well as any other settings we might need – like if the signal is inverted, etc.

Next, we'll want to initialize the state of our *SimulationChannelDescriptor* objects – we need to set what channel it's for, the sample rate, and the initial bit state (high or low).

At this point we'll need to take a step back and discuss some key concepts.

BitState

BitState is a type used often in the SDK. It can be either *BIT_LOW* or *BIT_HIGH*, and represents a channel's logic state.

Sample Rate (samples per second)

Sample Rate refers to how many samples per second the data is. Typically it refers to how fast we're collecting data, but for simulation, it refers to how fast we're generating sample data.

Sample Number

This is the absolute sample number, starting at sample 0. When a data collection starts, the first sample collected is Sample Number 0. The next sample collected is Sample Number 1, etc. This is the same in simulation. The first sample we'll provide is Sample Number 0, and so on.

SimulationChannelDescriptor

We need this object to describe a single channel of data, and what its waveform looks like. We do this in a very simple way:

- We provide the initial state of the channel (BIT_LOW, or BIT_HIGH)
- We move forward some number of samples, and then toggle the channel.

The initial bit state of the channel never changes. The state (high or low) of a particular sample number can be determined by knowing how many times it has toggled up to that point (an even or odd number

of times).

Put another way:

- In the very beginning, we specify the initial state (BIT_LOW or BIT_HIGH). This is the state of Sample Number 0.
- Then, we move forward (advance) some number of samples. 20 samples, for example.
- Then, we toggle the channel (low becomes high, high becomes low).
- Then we move forward (advance) some more. Maybe 100 samples this time.
- Then we toggle again.
- Then we move forward again, and then we toggle again, etc.

Let's explore the functions used to do this:

void Advance(U32 num_samples_to_advance);

As you might guess, this is how we move forward in our simulated waveform. Internally, the object keeps track of what its Sample Number is. The Sample Number starts at 0. After calling *Advance*(10) x3 times, the Sample Number will be 30.

void Transition();

This toggles the channel. *BIT_LOW* becomes *BIT_HIGH*, *BIT_HIGH* becomes *BIT_LOW*. The current Sample Number will become the new *BitState* (BIT_LOW or BIT_HIGH), and all samples after that will also be the new *BitState*, untill we toggle again.

void TransitionIfNeeded(BitState bit_state);

Often we don't want to keep track of the current *BitState*, which toggles every time we call *Transition*. *TransitionIfNeeded* checks the current *BitState*, and only transitions if the current *BitState* doesn't match the one we provide. In other words "Change to this bit_state, if we're not already".

BitState GetCurrentBitState();

This function lets you directly ask what the current *BitState* is.

U64 GetCurrentSampleNumber();

This function lets you ask what the current *SampleNumber* is.

ClockGenerator

ClockGenerator is a class provided in *AnalyzerHelpers.h* which will let you enter time values, rather than numbers-of-samples.

First, create an object using the *ClockGenerator* class, and then call the *init* function to initialize it.

void Init(double target_frequency, U32 sample_rate_hz);

You'll need to call this before using the class. For *sample_rate_hz*, enter the sample rate we'll be generating data at. For *target_frequency*, enter the frequency (in Hz) you will most commonly be using. For example, the bit rate of a SPI clock, etc.

U32 AdvanceByHalfPeriod(double multiple = 1.0);

This function returns how many samples are needed to move forward by one half of the period (for example, the low time for a perfect square wave). You can also enter a multiple. For example, to get the number of samples to move forward for a full period, enter 2.0.

U32 AdvanceByTimeS(double time_s);

This functions provides number of samples needed to advance by the arbitrary time, *time_s*. Note that this is in seconds, so enter 1e-6 for for one microsecond, etc.

Let's take a look at an example in *SerialSimulationDataGenerator.cpp*

```
void SerialSimulationDataGenerator::Initialize(U32 simulation_sample_rate,
                                               SerialAnalyzerSettings *settings)
{
     mSimulationSampleRateHz = simulation_sample_rate;
     mSettings = settings;

     mClockGenerator.Init(mSettings->mBitRate, simulation_sample_rate);
     mSerialSimulationData.SetChannel(mSettings->mInputChannel);
     mSerialSimulationData.SetSampleRate(simulation_sample_rate);

     if (mSettings->mInverted == false) {
         mBitLow = BIT_LOW;
         mBitHigh = BIT_HIGH;
     } else {
         mBitLow = BIT_HIGH;
         mBitHigh = BIT_LOW;
     }

     mSerialSimulationData.SetInitialBitState(mBitHigh);
     mSerialSimulationData.Advance(mClockGenerator.AdvanceByHalfPeriod(10.0));

     mValue = 0;
     mMpModeAddressMask = 0;
     mMpModeDataMask = 0;
     mNumBitsMask = 0;

     U32 num_bits = mSettings->mBitsPerTransfer;
     for (U32 i = 0; i < num_bits; i++) {
         mNumBitsMask <<= 1;
         mNumBitsMask |= 0x1;
     }

     if (mSettings->mSerialMode == SerialAnalyzerEnums::MpModeMsbOneMeansAddress)
         mMpModeAddressMask = 0x1ull << (mSettings->mBitsPerTransfer);
```

```
    if (mSettings->mSerialMode == SerialAnalyzerEnums::MpModeMsbZeroMeansAddress)
        mMpModeDataMask = 0x1ull << (mSettings->mBitsPerTransfer);
}
```

2.2.3 U32 {YourName}SimulationDataGenerator::GenerateSimulationData(U64 largest_sample_requested, U32 sample_rate, SimulationChannelDescriptor **simulation_channels)

This function is repeatedly called to request more simulated data. When it's called, just keep going where you left off. In addition, you can generate more data that requested, to make things easy -- that way you don't have to stop half way in the middle of something and try to pick it back up later exactly where you left off.

When we leave the function, our Sample Number – in our *SimulationChannelDecriptor* object(s) must be equal to or larger than *largest_sample_requested*. Actually, this number needs to first be adjusted (for technical reasons related to future compatibility). Use the helper function *AdjustSimulationTargetSample* to do this, as we'll see in a moment.

The parameter *simulation_channels* is to provide the caller with a pointer to an array of your *SimulationChannelDecriptor* objects. We'll set this pointer at the end of the function. The return value is the number of elements in the array – the number of channels.

The following is an example of *SerialSimulationDataGenerator.cpp*:

```
U32 SerialSimulationDataGenerator::GenerateSimulationData(U64 largest_sample_requested,
        U32 sample_rate, SimulationChannelDescriptor **simulation_channels)
{
    U64 adjusted_largest_sample_requested = AnalyzerHelpers::AdjustSimulationTargetSample(
            largest_sample_requested, sample_rate, mSimulationSampleRateHz);

    while (mSerialSimulationData.GetCurrentSampleNumber() < adjusted_largest_sample_requested) {
        if (mSettings->mSerialMode == SerialAnalyzerEnums::Normal) {
            CreateSerialByte(mValue++);
            mSerialSimulationData.Advance(mClockGenerator.AdvanceByHalfPeriod(10.0));
        } else {
            U64 address = 0x1 | mMpModeAddressMask;
            CreateSerialByte(address);

            for (U32 i=0; i<4; i++) {
                mSerialSimulationData.Advance(mClockGenerator.AdvanceByHalfPeriod(2.0));
                CreateSerialByte((mValue++ & mNumBitsMask) | mMpModeDataMask);
            };

            mSerialSimulationData.Advance(mClockGenerator.AdvanceByHalfPeriod(20.0));
```

```
            address = 0x2 | mMpModeAddressMask;
            CreateSerialByte( address );

            for (U32 i=0; i<4; i++) {
                mSerialSimulationData.Advance( mClockGenerator.AdvanceByHalfPeriod(2.0));
                CreateSerialByte((mValue++ & mNumBitsMask) | mMpModeDataMask);
            };

            mSerialSimulationData.Advance(mClockGenerator.AdvanceByHalfPeriod(20.0));
        }
    }

    *simulation_channels = &mSerialSimulationData;
    return 1;
}

void SerialSimulationDataGenerator::CreateSerialByte(U64 value)
{
    mSerialSimulationData.Transition();
    mSerialSimulationData.Advance(mClockGenerator.AdvanceByHalfPeriod());
    if (mSettings->mInverted == true)
        value = ~value;

    U32 num_bits = mSettings->mBitsPerTransfer;
    if (mSettings->mSerialMode != SerialAnalyzerEnums::Normal)
        num_bits++;

    BitExtractor bit_extractor(value, mSettings->mShiftOrder, num_bits);

    for (U32 i=0; i<num_bits; i++) {
        mSerialSimulationData.TransitionIfNeeded(bit_extractor.GetNextBit());
        mSerialSimulationData.Advance(mClockGenerator.AdvanceByHalfPeriod());
    }

    if (mSettings->mParity == AnalyzerEnums::Even) {
        if (AnalyzerHelpers::IsEven(AnalyzerHelpers::GetOnesCount(value)) == true)
            mSerialSimulationData.TransitionIfNeeded(mBitLow);
        else
            mSerialSimulationData.TransitionIfNeeded(mBitHigh);
        mSerialSimulationData.Advance(mClockGenerator.AdvanceByHalfPeriod());
    } else if (mSettings->mParity == AnalyzerEnums::Odd) {
        if (AnalyzerHelpers::IsOdd(AnalyzerHelpers::GetOnesCount(value)) == true)
            mSerialSimulationData.TransitionIfNeeded(mBitLow);
        else
```

```
            mSerialSimulationData.TransitionIfNeeded(mBitHigh);
        mSerialSimulationData.Advance(mClockGenerator.AdvanceByHalfPeriod());
    }
    mSerialSimulationData.TransitionIfNeeded(mBitHigh);

    mSerialSimulationData.Advance(mClockGenerator.AdvanceByHalfPeriod(mSettings->mStopBits));
}
```

There are a few things we could do to clean this up. First, we could save the *samples_per_bit* as a member variable, and compute it only once, in the *Initialize* function. If we wanted to be more accurate, we could use the *ClockGenerator* class to pre-populate an array of *samples_per_bit* values, so on average the timing would be perfect. We would use this as a lookup each time we *Advance* one bit. Another thing we could do is use the *DataExtractor* class to take care of the bit masking/testing. However, in our simple example what we have works well enough, and it has the advantage of being a bit more transparent.

2.2.4 Simulating Multiple Channels

Simulating multiple channels requres multiple *SimulationChannelDescriptors*, and they must be in an array. The best way to this is to use the helper class, *SimulationChannelDescriptorGroup*.

Here is an example of I2C (2 channels)—these are the the member varaible definitions in *I2cSimulationDataGenerator.h*:

```
    SimulationChannelDescriptorGroup mI2cSimulationChannels;
    SimulationChannelDescriptor *mSda;
    SimulationChannelDescriptor *mScl;
```

Then, in the *Initialize* function:

```
    mSda = mI2cSimulationChannels.Add(settings->mSdaChannel, mSimulationSampleRateHz, BIT_HIGH);
    mScl = mI2cSimulationChannels.Add(settings->mSclChannel, mSimulationSampleRateHz, BIT_HIGH);
```

And to provide the array to the caller of *GenerateSimulationData*:

```
    *simulation_channels = mI2cSimulationChannels.GetArray();
    return mI2cSimulationChannels.GetCount();
```

You can use each *SimulationChannelDescriptor* object pointer separately, calling *Advance*, *Transition*, etc on each one, or you can manipulate them as a group, using the *AdvanceAll* method of the *SimulationChannelDescriptorGroup* object.

```
    void AdvanceAll(U32 num_samples_to_advance)
```

Before returning from *GenerateSimulationData*, be sure that the Sample Number of all of your *SimulationChannelDescriptor* objects exceed *adjusted_largest_sample_requested*.

Examples of generating simulation data：

```
U32 I2cSimulationDataGenerator::GenerateSimulationData(U64 largest_sample_requested,
       U32 sample_rate, SimulationChannelDescriptor **simulation_channels)
```

```
{
    U64 adjusted_largest_sample_requested = AnalyzerHelpers::AdjustSimulationTargetSample(
            largest_sample_requested, sample_rate, mSimulationSampleRateHz);

    while (mScl->GetCurrentSampleNumber() < adjusted_largest_sample_requested) {
        mI2cSimulationChannels.AdvanceAll(mClockGenerator.AdvanceByHalfPeriod(500));

        if (rand() % 20 == 0) {
            CreateStart();
            CreateI2cByte(0x24, I2C_NAK);
            CreateStop();
            mI2cSimulationChannels.AdvanceAll(mClockGenerator.AdvanceByHalfPeriod(80));
        }

        CreateI2cTransaction(0xA0, I2C_WRITE, mValue++ + 12);
        mI2cSimulationChannels.AdvanceAll(mClockGenerator.AdvanceByHalfPeriod(80));
        CreateI2cTransaction(0xA0, I2C_READ, mValue++ - 43 + (rand( ) % 100 ));
        mI2cSimulationChannels.AdvanceAll(mClockGenerator.AdvanceByHalfPeriod(50));
        CreateI2cTransaction(0x24, I2C_READ, mValue++ + (rand() % 100));

        mI2cSimulationChannels.AdvanceAll(mClockGenerator.AdvanceByHalfPeriod(2000));

        CreateI2cTransaction(0x24, I2C_READ, mValue++ + 16 + (rand( ) % 100));

        mI2cSimulationChannels.AdvanceAll(mClockGenerator.AdvanceByHalfPeriod(100));
    }

    *simulation_channels = mI2cSimulationChannels.GetArray();
    return mI2cSimulationChannels.GetCount();
}
```

Note that above we use a number of helper functions and classes. Let's discuss *BitExtractor* briefly.

BitExtractor

```
    BitExtractor(U64 data, AnalyzerEnums::ShiftOrder shift_order, U32 num_bits);
    BitState GetNextBit();
```

Some protocols have varaible numbers of bits per word, and settings for if the most significant bit is first or last. This can be a pain to manage, so we made the *BitExtraxtor* class. This can be done by hand of course if you like, but this class tends to tidy up the code quite a bit in our experiece.

Similar, but reveresed, is the *DataBuilder* class, but as this generally used for collecting data, we'll talk more about it then.

AnalyzerHelpers

Some static helper functions that might be helpful, include：

```
    static bool IsEven(U64 value);
    static bool IsOdd(U64 value);
    static U32 GetOnesCount(U64 value);
    static U32 Diff32(U32 a, U32 b);
```

3. AnalyzerResults

After creating your *SimulationDataGenerator* class, working on your *{YourName}AnalyzerResults* files is the next step.

AnalyzerResults is what we use to transform our results into text for display and as well as exported files, etc.

Tip: You may end up finalizing may of the details about how your results are saved when you work on your main *Analyzer* file – *{YourName}Analyzer.cpp/.h*; You can simply implement the bare minimum of the functions in your *{YourName}AnalyzerResults.cpp* file, and come back to it later.

3.1 {YourName}AnalyzerResults.h

In addition to the constructor and destructor, there are 5 functions we'll need to implement.

Here's the *SerialAnalyzerResults* header file. Yours will like very similar, with the only difference typically being the *enums* and/or *defines* you need.

```
#ifndef SERIAL_ANALYZER_RESULTS
#define SERIAL_ANALYZER_RESULTS

#include <AnalyzerResults.h>

#define FRAMING_ERROR_FLAG (1 << 0)
#define PARITY_ERROR_FLAG (1 << 1)
#define MP_MODE_ADDRESS_FLAG (1 << 2)

class SerialAnalyzer;
class SerialAnalyzerSettings;

class SerialAnalyzerResults : public AnalyzerResults
{
public:
    SerialAnalyzerResults(SerialAnalyzer *analyzer, SerialAnalyzerSettings *settings);
    virtual ~SerialAnalyzerResults();

    virtual void GenerateBubbleText(U64 frame_index, Channel& channel, DisplayBase display_base);
    virtual void GenerateExportFile(const char *file, DisplayBase display_base,
                                    U32 export_type_user_id);
```

```
    virtual void GenerateFrameTabularText(U64 frame_index, DisplayBase display_base);
    virtual void GeneratePacketTabularText(U64 packet_id, DisplayBase display_base);
    virtual void GenerateTransactionTabularText(U64 transaction_id, DisplayBase  display_base);

protected: //functions

protected:  //vars
    SerialAnalyzerSettings *mSettings;
    SerialAnalyzer *mAnalyzer;
};

#endif //SERIAL_ANALYZER_RESULTS
```

3.2 {YourName}AnalyzerResults.cpp

3.2.1 Constructors and analytic functions

In your constructor, save copies of the Analyzer and Settings raw pointers provided. There's generally nothing else to do for the constructor or destructor. Below is an example from *SerialAnalyzerResults.cpp*:

```
SerialAnalyzerResults::SerialAnalyzerResults(SerialAnalyzer *analyzer,
                                             SerialAnalyzerSettings *settings)
    : AnalyzerResults(),
      mSettings(settings),
      mAnalyzer(analyzer)
{
}
SerialAnalyzerResults::~SerialAnalyzerResults()
{
}
```

The basic result an analyzer generates is called a *Frame*. This could be byte of serial data, the header of a CAN packet, the MOSI and MISO values from 8-bit of SPI, etc. Smaller elements, such the Start and Stop events in I2C can be saved as Frames, are probably better saved as be graphical elements (called *Markers*) and otherwise ignored. *Collections* of Frames make up *Packets*, and collections of *Packets* make up *Transactions*.

95% of what you will be concerned about is *Frames*. What exactly a *Frame* represents is your choice, but unless your protocol is fairly complicated (such as USB, CAN, Ethernet) the best bet is to make the *Frame* your main result element.

We'll get into more detail regarding how to save your results when we describe to your *Analyzer*-derived class.

Frame

A *Frame* is an object, with fairly generic member variables which can be used to save results. Here is the definition of a *Frame*:

```
class LOGICAPI Frame
{
public:
     Frame();
     Frame(constFrame &frame);
     ~Frame();
     S64 mStartingSampleInclusive;
     S64 mEndingSampleInclusive;
     U64 mData1;
     U64 mData2;
     U8 mType;
     U8 mFlags;
};
```

A *Frame* represents a piece of information conveyed by your protocol over an expanse of time. The member variables *mStartingSampleInclusive* and *mEndingSampleInclusive* are the sample numbers for the beginning and end of the *Frame*. Note that Frames may not overlap; they cannot even share the same sample. For example, if a single clock edge ends one Frame, and starts a new Frame, then you'll need to add one (+1) to the *mStartingSampleInclusive* of the second frame.

In addition, the *Frame* can carry two 64-bit numbers as data. For example, in SPI, one of these is used for the MISO result, and the other for the MISO result. Often times you'll only use one of these variables.

The *mType* variable is intended to be used to save a custom-defined enum value, representing the type of *Frame*. For example, CAN can have many different types of frames – header, data, CRC, etc. Serial only has one type, and it doesn't use this member variable.

mFlags is intended to be a holder for custom flags which might apply to frame. Note that this is not intended for use with a custom an *enum*, but rather for individual bits that can be or'ed together. For example, in Serial, there is a flag for framing-error, and a flag for parity error.

```
     #define FRAMING_ERROR_FLAG (1 << 0)
     #define PARITY_ERROR_FLAG (1 << 1)
```

Two flags are reserved by the system, and will produce an error or warning indication on the bubble displaying the *Frame*.

3.2.2 void {YourName}AnalyzerResults::GenerateBubbleText(U64 frame_index, Channel &channel, DisplayBase display_base)

GenerateBubbleText exists to retrieve text to put in a bubble to be displayed on the screen. If you like you can leave this function empty, and return to it after implementing the rest of your analyzer.

The *frame_index* is the index to use to get the Frame itself – for example:

```
    Frame frame = GetFrame(frame_index);
```

Rarely, an analyzer needs to display results on more than one channel (SPI is the only example of this in an analyzer we make). If so, the channel which is requesting the bubble is specified in the channel parameter.

display_base specifies the radix (hex, decimal, binary) that any numerical values should be displayed in. There are some helper functions provided so you should never have to deal directly with this issue.

```
    enum DisplayBase {Binary, Decimal, Hexadecimal, ASCII, AsciiHex};
    AnalyzerHelpers::GetNumberString(U64 number, DisplayBase display_base, U32 num_data_bits,
                                     char *result_string, U32 result_string_max_length);
```

In *GetNumberString*, above, note that *num_data_bits* is the number of bits which are actually part of your result. For sample, for I2C, this is always 8. It will depend on your protocol and possibly on user settings. Providing this will let *GetNumberString* produce a well-formatted number with the right amount of zero-padding for the type of value under consideration.

Bubbles can display different length strings, depending on how much room is available. You should generate several results strings. The simplest might simply indicate the type of contents (‘D’ for data, for example), longer ones might indicate the full number (“0xFF01”), and longer ones might be very verbose (“Left Channel Audio Data: 0xFF01”).

To provide strings to the caller, use the *AddStringResult* function. This will make sure that the strings persist after the function has returned. Always call *ClearResultStrings* before adding any string results.

Note that to easily concatenate multiple strings, simply provide *AddStringResult* with more strings.

```
    Void ClearResultStrings();
    Void AddResultString(const char *str1, const char *str2 = NULL, const char *str3 = NULL,
            const char *str4 = NULL, const char *str5 = NULL, const char *str6 = NULL);
```

Here’s the Serial Analyzer’s *GenerateBubbleText* function:

```
void SerialAnalyzerResults::GenerateBubbleText(U64 frame_index, Channel& /*channel*/,
        DisplayBase display_base)  //unrefereced vars commented out to remove warnings.
{
    //we only need to pay attention to 'channel' if we're making bubbles for more than
    //one channel (as set by AddChannelBubblesWillAppearOn)
    ClearResultStrings();
    Frame frame = GetFrame(frame_index);

    bool framing_error = false;
    if ((frame.mFlags & FRAMING_ERROR_FLAG) != 0)
        framing_error = true;

    bool parity_error = false;
```

```
if ((frame.mFlags & PARITY_ERROR_FLAG) != 0)
    parity_error = true;

U32 bits_per_transfer = mSettings->mBitsPerTransfer;
if (mSettings->mSerialMode != SerialAnalyzerEnums::Normal)
    bits_per_transfer--;

char number_str[128];
AnalyzerHelpers::GetNumberString(frame.mData1, display_base, bits_per_transfer,
                                 number_str, 128);

char result_str[128];

//MP mode address case:
bool mp_mode_address_flag = false;
if ((frame.mFlags & MP_MODE_ADDRESS_FLAG) != 0) {
    mp_mode_address_flag = true;

    AddResultString("A");
    AddResultString("Addr");

    if (framing_error == false) {
        snprintf(result_str, sizeof(result_str), "Addr: %s", number_str);
        AddResultString(result_str);

        snprintf(result_str, sizeof(result_str), "Address: %s", number_str);
        AddResultString(result_str);
    } else {
        snprintf(result_str, sizeof(result_str), "Addr: %s (framing error)", number_str);
        AddResultString(result_str);
        snprintf(result_str, sizeof(result_str), "Address: %s (framing error)", number_str);
        AddResultString(result_str);
    }
    return;
}

//normal case:
if ((parity_error == true) || (framing_error == true)) {
    AddResultString("!");

    snprintf(result_str, sizeof(result_str), "%s (error)", number_str);
    AddResultString(result_str);

    if (parity_error == true && framing_error == false)
```

```
            snprintf(result_str, sizeof(result_str), "%s (parity error)", number_str);
        else if (parity_error == false && framing_error == true)
            snprintf(result_str, sizeof(result_str), "%s (framing error)", number_str);
        else
            snprintf(result_str, sizeof(result_str), "%s (framing error & parity error)",
                     number_str);
        AddResultString(result_str);
    } else {
        AddResultString(number_str);
    }
}
```

3.2.3 void {YourName}AnalyzerResults::GenerateExportFile(const char *file, DisplayBase display_base, U32 export_type_user_id)

This function is called when the user tries to export the analyzer results to a file. If you like, you can leave this function empty, and come back to it after finalizing the rest of your analyzer design.

The *file* parameter is string containing the full path of the file you should create and write to with the analyzer results.

```
    std::ofstream file_stream(file, std::ios::out);
```

The *display_base* parameter contains the radix which should be used to display numerical results.

The *export_type_user_id* parameter is the id associated with the export-type the user selected. You specify what these options are (there should be at least one) in the constructor of your *AnalyzerSettings*-derived class. If you only have one export option you can ignore this parameter.

Often times you'll want to print out the time (in seconds) associated with a particular result. To do this, use the *GetTimeString* helper function. You'll need the trigger sample number and the *sample rate* – which can be obtained from your Analyzer object pointer.

```
    U64 trigger_sample = mAnalyzer->GetTriggerSample();
    U32 sample_rate = mAnalyzer->GetSampleRate();
    static void AnalyzerHelpers::GetTimeString(U64 sample, U64 trigger_sample,
                U32 sample_rate_hz, char *result_string, U32 result_string_max_length);
```

Other than that, the implementation is pretty straightforward. Here is an example from *SerialAnalyzerResults.cpp*:

```
void SerialAnalyzerResults::GenerateExportFile(const char *file, DisplayBase display_base,
                                               U32 /*export_type_user_id*/)
{
    //export_type_user_id is only important if we have more than one export type.
    std::stringstream ss;

    U64 trigger_sample = mAnalyzer->GetTriggerSample();
```

```
U32 sample_rate = mAnalyzer->GetSampleRate();
U64 num_frames = GetNumFrames();

void *f = AnalyzerHelpers::StartFile(file);

if (mSettings->mSerialMode == SerialAnalyzerEnums::Normal) {
    //Normal case -- not MP mode.
    ss << "Time [s],Value,Parity Error,Framing Error" << std::endl;

    for (U32 i=0; i < num_frames; i++) {
        Frame frame = GetFrame(i);

        char time_str[128];
        AnalyzerHelpers::GetTimeString(frame.mStartingSampleInclusive,
                                       trigger_sample, sample_rate, time_str, 128);

        char number_str[128];
        AnalyzerHelpers::GetNumberString(frame.mData1, display_base,
                                         mSettings->mBitsPerTransfer, number_str, 128);

        ss << time_str << "," << number_str;

        if ((frame.mFlags & PARITY_ERROR_FLAG) != 0)
            ss << ",Error,";
        else
            ss << ",,";

        if ((frame.mFlags & FRAMING_ERROR_FLAG) != 0)
            ss << "Error";

        ss << std::endl;

        AnalyzerHelpers::AppendToFile((U8*)ss.str().c_str(), ss.str().length(), f);
        ss.str(std::string());

        if (UpdateExportProgressAndCheckForCancel(i, num_frames) == true) {
            AnalyzerHelpers::EndFile(f);
            return;
        }
    }
} else {
    //MP mode.
    ss << "Time [s],Packet ID,Address,Data,Framing Error" << std::endl;
    U64 address = 0;
```

```
        for (U32 i=0; i < num_frames; i++) {
            Frame frame = GetFrame(i);
            if ((frame.mFlags & MP_MODE_ADDRESS_FLAG) != 0) {
                address = frame.mData1;
                continue;
            }

            U64 packet_id = GetPacketContainingFrameSequential(i);
            char time_str[128];
            AnalyzerHelpers::GetTimeString(frame.mStartingSampleInclusive,
                                           trigger_sample, sample_rate, time_str, 128);

            char address_str[128];
            AnalyzerHelpers::GetNumberString(address, display_base,
                                             mSettings->mBitsPerTransfer - 1, address_str, 128);

            char number_str[128];
            AnalyzerHelpers::GetNumberString(frame.mData1, display_base,
                                             mSettings->mBitsPerTransfer - 1, number_str, 128);
            if (packet_id == INVALID_RESULT_INDEX)
                ss << time_str << "," << "" << "," << address_str << "," << number_str << ",";
            else
                ss << time_str << "," << packet_id << "," << address_str << "," << number_str << ",";

            if ((frame.mFlags & FRAMING_ERROR_FLAG) != 0)
                ss << "Error";

            ss << std::endl;

            AnalyzerHelpers::AppendToFile((U8*)ss.str().c_str(), ss.str().length(), f);
            ss.str(std::string());
            if (UpdateExportProgressAndCheckForCancel(i, num_frames) == true) {
                AnalyzerHelpers::EndFile(f);
                return;
            }
        }
    }
    UpdateExportProgressAndCheckForCancel(num_frames, num_frames);
    AnalyzerHelpers::EndFile(f);
}
```

3.2.4 void {YourName}AnalyzerResults::GenerateFrameTabularText(U64 frame_index, DisplayBase display_base)

GenerateFrameTabularText is for producing text for tabular display which is not yet implemented. You can safely leave it empty.

GenerateFrameTabularText is almost the same as *GenerateBubbleText*, except that you should generate only one text result. Ideally the string should be concise, and only be a couple inches long or less under normal (non error) circumstances.

Here is an example from *SerialAnayzerResults.cpp*:

```
    void SerialAnalyzerResults::GenerateFrameTabularText(U64 frame_index,
                                                         DisplayBase display_base)
{
    ClearTabularText();
    Frame frame = GetFrame(frame_index);

    bool framing_error = false;
    if ((frame.mFlags & FRAMING_ERROR_FLAG) != 0)
        framing_error = true;

    bool parity_error = false;
    if ((frame.mFlags & PARITY_ERROR_FLAG) != 0)
        parity_error = true;

    U32 bits_per_transfer = mSettings->mBitsPerTransfer;
    if (mSettings->mSerialMode != SerialAnalyzerEnums::Normal)
        bits_per_transfer--;

    char number_str[128];
    AnalyzerHelpers::GetNumberString(frame.mData1, display_base, bits_per_transfer,
                                     number_str, 128);

    char result_str[128];

    //MP mode address case:
    bool mp_mode_address_flag = false;
    if ((frame.mFlags & MP_MODE_ADDRESS_FLAG) != 0) {
        mp_mode_address_flag = true;

        if (framing_error == false) {
            snprintf(result_str, sizeof(result_str), "Address: %s", number_str);
            AddTabularText(result_str );
        } else {
            snprintf(result_str, sizeof(result_str), "Address: %s (framing error)", number_str);
```

```
            AddTabularText(result_str);
        }
        return;
    }

    //normal case:
    if ((parity_error == true) || (framing_error == true)) {
        if (parity_error == true && framing_error == false)
            snprintf( result_str, sizeof(result_str), "%s (parity error)", number_str );
        else if (parity_error == false && framing_error == true)
            snprintf(result_str, sizeof(result_str), "%s (framing error)", number_str);
        else
            snprintf(result_str, sizeof(result_str), "%s (framing error & parity error)",
                     number_str);
        AddTabularText(result_str);
    } else {
        AddTabularText(number_str);
    }
}
```

3.2.5 void {YourName}AnalyzerResults::GeneratePacketTabularText(U64 packet_id, DisplayBase display_base)

This function is used to produce strings representing packet results for the tabular view. For now, just leave it empty. We'll be updating the SDK and software to take advantage of this capability later.

3.2.6 void {YourName}AnalyzerResults::GenerateTransactionTabularText(U64 transaction_id, DisplayBase display_base)

This function is used to produce strings representing packet results for the tabular view. For now, just leave it empty. We'll be updating the SDK and software to take advantage of this capability later.

4. Analyzer

Your *Analyzer*-derived class is the heart of the analyzer. It's here were we analyze the bits coming in – in real time – and generate analyzer results. Other than a few other housekeeping things, that's it. Let's get started.

4.1 {YourName}Analyzer.h

In addition to the constructor and destructor, here are the functions you'll need to implement:

```
    virtual voidWorkerThread();
    virtual U32 GenerateSimulationData(U64 newest_sample_requested, U32 sample_rate,
                                       SimulationChannelDescriptor **simulation_channels);
    virtual U32 GetMinimumSampleRateHz();
    virtual const char *GetAnalyzerName() const;
```

```
    virtual boolNeedsRerun();
    extern "C"ANALYZER_EXPORT const char *__cdecl GetAnalyzerName();
    extern "C"ANALYZER_EXPORT Analyzer *__cdecl CreateAnalyzer();
    extern "C"ANALYZER_EXPORT void __cdecl DestroyAnalyzer(Analyzer *analyzer);
```

You'll also need these member variables:

```
    std::auto_ptr< {YourName}AnalyzerSettings > mSettings;
    std::auto_ptr< {YourName}AnalyzerResults > mResults;
    {YourName}SimulationDataGenerator mSimulationDataGenerator;
    bool mSimulationInitilized;
```

You'll also need one *AnalyzerChannelData* raw pointer for each input. For *SerialAnalyzer*, for example, we need

```
    AnalyzerChannelData *mSerial;
```

As you develop your analyzer, you'll add additional member variables and helper functions dependning on your analysis needs.

Here is an example of *SerialAnalyzer.h*：

```
#ifndef SERIAL_ANALYZER_H
#define SERIAL_ANALYZER_H

#include <Analyzer.h>
#include "SerialAnalyzerResults.h"
#include "SerialSimulationDataGenerator.h"

class SerialAnalyzerSettings;

class ANALYZER_EXPORT SerialAnalyzer : public Analyzer
{
public:
    SerialAnalyzer();
    virtual ~SerialAnalyzer();
    virtual void SetupResults();
    virtual void WorkerThread();

    virtual U32 GenerateSimulationData(U64 newest_sample_requested, U32 sample_rate,
                                       SimulationChannelDescriptor **simulation_channels);
    virtual U32 GetMinimumSampleRateHz();

    virtual const char *GetAnalyzerName() const;
    virtual bool NeedsRerun();

#pragma warning(push)
#pragma warning(disable : 4251)
```

```
protected: //functions
     void ComputeSampleOffsets();

protected: //vars
     std::auto_ptr< SerialAnalyzerSettings > mSettings;
     std::auto_ptr< SerialAnalyzerResults > mResults;
     AnalyzerChannelData *mSerial;

     SerialSimulationDataGenerator mSimulationDataGenerator;
     bool mSimulationInitilized;

     //Serial analysis vars:
     U32 mSampleRateHz;
     std::vector<U32> mSampleOffsets;
     U32 mParityBitOffset;
     U32 mStartOfStopBitOffset;
     U32 mEndOfStopBitOffset;
     BitState mBitLow;
     BitState mBitHigh;

#pragma warning( pop )
};

extern "C" ANALYZER_EXPORT const char *__cdecl GetAnalyzerName();
extern "C" ANALYZER_EXPORT Analyzer *__cdecl CreateAnalyzer();
extern "C" ANALYZER_EXPORT void __cdecl DestroyAnalyzer(Analyzer *analyzer);

#endif //SERIAL_ANALYZER_H
```

4.2 {YourName}Analyzer.cpp

4.2.1 Constructor

Your constructor will look something like this

```
{YourName}Analyzer::{YourName}Analyzer()
     : Analyzer(),
       mSettings(new{YourName}AnalyzerSettings()),
       mSimulationInitilized(false)
{
     SetAnalyzerSettings(mSettings.get());
}
```

Note that here you're calling the base class conststructor, newing your *AnalyzerSettings*-derived class, and providing the base class with a pointer to your *AnalyzerSettings*-derived object.

4.2.2 Destructor

This only thing your destructor must do is call *KillThread*. This is a base class member function and will make sure your class destructs in the right order.

4.2.3 void {YourName}Analyzer::SetupResults()

In this function, you need to generate the *AnalyzerResults*-derived class object, save the object pointer to the *Analyzer*-derived class, and pass the channel display analyzing result to the *AnalyzerResults*-derived class.

For example in *SerialAnalyzer.cpp*:

```
void SerialAnalyzer::SetupResults()
{
     mResults.reset(new SerialAnalyzerResults(this, mSettings.get()));
     SetAnalyzerResults(mResults.get());
     mResults->AddChannelBubblesWillAppearOn(mSettings->mInputChannel);
}
```

In *SpiAnalyzer.cpp*:

```
void SpiAnalyzer::SetupResults()
{
     mResults.reset(new SpiAnalyzerResults(this, mSettings.get()));
     SetAnalyzerResults(mResults.get());

     if (mSettings->mMosiChannel != UNDEFINED_CHANNEL)
         mResults->AddChannelBubblesWillAppearOn(mSettings->mMosiChannel);
     if(mSettings->mMisoChannel != UNDEFINED_CHANNEL)
         mResults->AddChannelBubblesWillAppearOn(mSettings->mMisoChannel);
}
```

4.2.4 void {YourName}Analyzer::WorkerThread()

This function the key to everything – it's where you'll decode the incoming data. Let's leave it empty for now, and we'll discuss in detail once we complete the other housekeeping functions.

4.2.5 bool {YourName}Analyzer::NeedsRerun()

Generally speaking, just return false in this function. For more detail, read on.

This function is called when your analyzer has finished analyzing the collected data (this condition is detected from outside your analyzer.)

This function gives you the opportunity to run the analyzer all over again, on the same data. To do this, simply return *true*. Otherwise, return *false*. The only thing this is currently used for is for our *Serial* analyzer, for "autobaud". When using autobaud, we don't know ahead of time what the serial bit rate will be. If the rate turns out to be significantly different from the rate we ran the analysis at, we return *true* to re-run the analysis.

If you return *true*, that's all there is to do. Your analyzer will be re-run automatically.

4.2.6 U32 {YourName}Analyzer::GenerateSimulationData(U64 Minimum_sample_index, U32 device_sample_rate, SimulationChannelDescriptor **simulation_channels)

This is the function that gets called to obtain simulated data. We made a dedicated class for handling this earlier – we just need to do some housekeeping here to hook it up.

```
U32 {YourName}Analyzer::GenerateSimulationData(U64 minimum_sample_index, U32 device_sample_rate,
                                               SimulationChannelDescriptor **simulation_channels)
{
     if (mSimulationInitilized == false) {
         mSimulationDataGenerator.Initialize(GetSimulationSampleRate(), mSettings.get());
         mSimulationInitilized = true;
     }
     return mSimulationDataGenerator.GenerateSimulationData(minimum_sample_index,
                                         device_sample_rate, simulation_channels);
}
```

4.2.7 U32 {YourName}SerialAnalyzer::GetMinimumSampleRateHz()

This function is called to see if the user's selected sample rate is sufficient to get good results for this analyzer.

An example in *SerialAnalyzer.cpp*:

```
U32 SerialAnalyzer::GetMinimumSampleRateHz()
{
     return mSettings->mBitRate * 4;
}
```

If we set baud rate as 9600, then the minium sampling frequency was suggested wo be more than 9600*4=38400Hz.

4.2.8 const char *{YourName}Analyzer::GetAnalyzerName() const

Simply return the name you would like to see in software's GUI.

```
     return "UART/232/485";
```

4.2.9 const char *GetAnalyzerName()

Return the same string as in the previous function.

```
     return "UART/232/485";
```

4.2.10 Analyzer *CreateAnalyzer()

Return a pointer to a new instance of your Analyzer-derived class.

4.2.11 void DestroyAnalyzer(Analyzer *analyzer)

Simply call delete on the provided pointer.

4.2.12 Details

Ok, now that everything else is taken care of, let's look at the most important part of the analyer in detail.

First, we'll new our *AnalyzerResults*-derived object.

```
mResults.reset(new{YourName}AnalyzerResults(this, mSettings.get()));
```

Well provide a pointer to our results to the base class:

```
SetAnalyzerResults(mResults.get());
```

Let's indicate which channels we'll be displaying results on (in the form of bubbles). Usually this will only be one channel. (Except in the case of SPI, where we'll want to put bubbles on both the MISO and MISO lines.) Only indicate where we will display bubbles – other markup, like tick marks, arrows, etc, are not bubbles, and should not be reported here.

```
mResults->AddChannelBubblesWillAppearOn(mSettings->mInputChannel);
```

We'll probably want to know (and save in a member variable) the sample rate.

```
mSampleRateHz = GetSampleRate();
```

Now we need to get access to the data itself. We'll need to get pointers to AnalyzerChannelData objects for each channel we'll need data from. For Serial, we'll just need one. For SPI, we might need 4. Etc.

```
mSerial = GetAnalyzerChannelData(mSettings->mInputChannel);
```

We've now ready to start traversing the data, and recording results. We'll look at each of these tasks in turn.

First, a word of advice

A protocol is typically fairly straightforward, when it behaves exactly as it supposed to. The more your analyzer needs to deal with exceptions to the rule, the more sophisticated it'll need to be. The best bet is probably to start as simple as possible, and add more "gotchas" as they are discovered, rather than to try and design an elaborate, bulletproof analyzer from the start, especially when you're new to the API.

AnalyzerChannelData

AnalyzerChannelData is the class that will give us access to the data from a particular input. This will provide data in a serialized form – we will not have "random access" to any bit in the saved data. Rather, we will start at the beginning, and move forward as more data becomes available. In fact we'll never know when we're at the "end" of the data or not – attempts to move forward in the stream will block until more data becomes available. This will allow our analyzer to process data in a real-time manner. (It may backlog, of course, if it can't keep up – although generally the collection will end at some point and we'll be able to finish).

AnalyzerChannelData - State

If we're not sure where are in the stream, or if the input is currently high or low, we can just ask:

```
    U64 GetSampleNumber();
    BitState GetBitState();
```

AnalyzerChannelData - Basic Traversal

We'll need some ability to move forward in the stream. We have three basic ways to do this.

U32 Advance(U32 num_samples);

We can move forward in the stream by a specific number of samples. This function will return how many times the input toggled (changed from a high to a low, or low to a high) to make this move.

U32 AdvanceToAbsPosition(U64 sample_number);

If we want to move forward to a particular absolute position, we can use this function. It also returns the number of times the input changed during the move.

void AdvanceToNextEdge();

We also might want to move forward until the state changes. After calling this function you might want to call *GetSampleNumber* to find out how far you've come.

AnalyzerChannelData- Advanced Traversal (looking ahead without moving)

As you develop your analyzer(s) certain tasks may come up that call for more sophisticated traversal. Here are some ways of doing it.

U64 GetSampleOfNextEdge();

This function does not move your position in the stream. Remember, you can not move backward in the stream, so sometimes seeing what's up ahead without moving can be very important.

bool WouldAdvancingCauseTransition(U32 num_samples);

This function does not move your position in the stream. Here you find out if moving forward a given number of samples would cause the bit state (low or high) to change.

bool WouldAdvancingToAbsPositionCauseTransition(U64 sample_number);

This is the same as the prior function, except you provide the absolute position.

Filling in and saving Frames

Using the above *AnalyzerChannelData* class, we can now move through a channel's data and analyze it. Now let's discus how to store results.

We described *Frames* when talking about the *AnalyzerResults*-derived class. A *Frame* is the basic unit results are saved in. *Frames* have:

- starting and ending time (starting and ending sample number)
- x2 64-bit values to save results in
- an 8-bit type variable - to specify the type of Frame
- an 8-bit flags variable - to specify Yes/No types of results.

When we have analyzed far enough, and now have a complete *Frame* we would like to record, we do it like this:

```
Frame frame;
frame.mStartingSampleInclusive = first_sample_in_frame;
frame.mEndingSampleInclusive = last_sample_in_frame;
frame.mData1 = the_data_we_collected;
//frame.mData2 = some_more_data_we_collected;
//frame.mType = OurTypeEnum;  //unless we only have one type of frame
frame.mFlags = 0;
if (such_and_such_error == true)
    frame.mFlags |= SUCH_AND_SUCH_ERROR_FLAG | DISPLAY_AS_ERROR_FLAG;
if (such_and_such_warning == true)
    frame.mFlags |= SUCH_AND_SUCH_WARNING_FLAG | DISPLAY_AS_WARNING_FLAG;
mResults->AddFrame(frame);
mResults->CommitResults();
ReportProgress(frame.mEndingSampleInclusive);
```

First we make a *Frame* on the stack. Then we fill in all its values. If there's a value you don't need, to save time you can skip setting it. *mFlags* should always be set to zero, however, because certan pre-defined flags will cause the results bubble to indicate a warning or error.

Part of the Frame is expected to be filled in correctly because it's used automaticly by other systems. In particular,

- mStartingSampleInclusive
- mEndingSampleInclusive
- mFlags

should be filled in properly.

Other parts of the Frame are only there so you can create text descriptions or export the data to a desired format.

To save a *Frame*, Use *AddFrame* from your *AnalyzerResults*-derived class. Note that frames must be added in-order, and must not overlap. In other words, you can't add a *Frame* from an earlier time (smaller sample number) after adding a *Frame* form a later time (larger sample number).

Immediately after adding a *Frame*, call *CommitResults*. This makes the *Frame* accessable to the external system.

Also call the *Analyzer* base class *ReportProgress*. Provide it with it the largest sample number you have processed.

Adding Markers

Makers are visual elements you can place on the waveform to highlight various waveform features as they relate to your protocol. For example, in our asynchronous serial analyzer, we place little white dots

at the locations where we sample the input's state. You can also use markers to indicate where the protocol falls out of specification, a rising or falling clock edge, etc. You specify where to put the marker (the sample number), which channel to display it on, and which graphical symbol to use.

```
void AddMarker(U64 sample_number, MarkerType marker_type, Channel &channel);
```

For example, from *SerialAnalyzer.cpp*:

```
mResults->AddMarker(marker_location, AnalyzerResults::Dot, mSettings->mInputChannel);
```

Currently, the available graphical artifacts are

```
enum MarkerType { Dot, ErrorDot, Square, ErrorSquare, UpArrow, DownArrow, X, ErrorX,
                  Start, Stop, One, Zero };
```

Like *Frames*, you must add *Markers* in order.

Markers are strictly for graphical markup, they can not be used to help generate display text, export files, etc. Only *Frames* are accessable to do that.

Packets and Trasactions

Packets and *Trasactions* are only moderatly supported as of now, but they will be becoming more prominate in the software.

Packets are sequential collections of Frames. Grouping *Frames* into *Packets* as you create them is easy:

```
U64 CommitPacketAndStartNewPacket();
void CancelPacketAndStartNewPacket();
```

When you add a *Frame*, it will automatically be added to the current *Packet*. When you've added all the *Frames* you want in a *Packet*, call *CommitPacketAndStartNewPacket*. In some conditions, especially errors, you will want start a new packet without committing the old one. For this, call *CancelPacketAndStartNewPacket*.

Note that *CommitPacketAndStartNewPacket* returns an packet id. You can use this id to assign a particular packet to a transaction.

```
void AddPacketToTransaction(U64 transaction_id, U64 packet_id);
```

Currently, *Packets* are only used when exporting data to text/csv. In the future, analyzer tabular views will support nesting *Frames* into Packets, and identifying *Transactions* (ids) associated with particular *Packets*. Generating the textual content to support this is provided in your *AnalyzerResults*-derived class.

When using Packet IDs when exporting data to text/csv, use the *GetPacketContainingFrameSequential* function, to avoid searching for the packet every time. The *GetPacketContainingFrame* will do a full search and be much less efficient.