KingstVIS 协议解析器开发指南

青岛金思特电子有限公司

官网：www.qdkingst.com
邮箱：service@qdkingst.com
电话：13780615696
QQ：415942827

目 录

一、 建立一个解析器工程

1、 整体说明

KingstVIS 软件支持用户自定义协议解析器，使用 C++语言进行开发。

本文档分为两部分：第一部分为建立协议解析器项目工程的步骤，介绍如何在 Windows、Linux、Mac 系统上生成解析器的动态链接库文件；第二部分为编写解析代码的相关内容，主要介绍基本类及成员函数的用法以及使用中的注意事项。

2、 工程构建

在开始下面的步骤之前，请先确认您已经拥有了我们提供的 KingstVIS_Analyzer_SDK。

在 KingstVIS_Analyzer_SDK 中包含有“inc”、“lib”以及我们提供的样例“SerialAnalyzer”等文件夹。在“inc”中存放的是基类的头文件；在“lib”中还有“Win32”、“Win64”、“Linux”、“Mac”四个文件夹，“Win32”和“Win64”中分别存放 32-bit 和 64-bit 的 Analyzer.lib 和 Analyzer.dll 两个文件，“Linux”中存放 64-bit 的 libAnalyzer.so 文件，“Mac”中存放 64-bit 的 libAnalyzer.dylib 文件；在“SerialAnalyzer”中存放的是异步串行协议（UART）解析器的工程及源代码，其中还有“src”和“vs2013”、“Linux”、“Mac”四个文件夹，“src”中存放解析器的源代码；“vs2013”中存放的是使用 Visual Studio 2013 建立的工程，可以使用该工程在 Windows 系统下生成解析所需要的动态库文件；在 Linux 系统中可以使用“Linux”文件夹下的 makefile 文件生成所需要的动态库文件；在 Mac 系统下可以使用“Mac”文件夹下的 makefile 文件生成所需要的动态库文件。

2.1. Windows

(1)、以生成解析 SPI 协议所需动态库为例进行讲解，可以直接在样例“SerialAnalyzer”的基础上进行修改以得到 SPI 解析器的工程，如果想要保留这个样例，可以将该样例复制一份在同一目录下如图 1 所示，然后在“SerialAnalyzer - 副本”上进行修改；

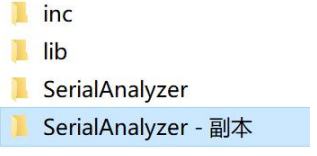


图 1

(2)、首先修改文件夹的名称，可以参考样例进行修改，将文件夹名称修改为“SpiAnalyzer”，然后打开这个文件夹，将“src”文件夹中的源文件替换为用于解析 SPI 的源文件，具体源文件的建立将在下一章节中进行讲解，再打开“vs2013”文件夹，将其中的文件名修改为“SpiAnalyzer.vcxproj”，如图 2 所示；

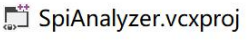


图 2

(3)、使用 Visual Studio 2013（或更高版本）打开“SpiAnalyzer.vcxproj”这个工程文件，如果“打开”对话框中的文件类型没有包含“*.vcxproj”项目而导致无法打开工程，是因为您的 Visual Studio 在安装时没有包含 VC++的组件，请再次运行 Visual Studio 的安装程序添加 VC++组件即可。

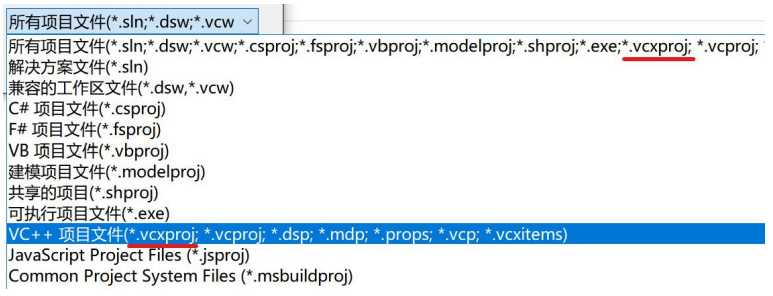


图 3

打开工程文件后，将原有的.cpp 和.h 文件全部移除；

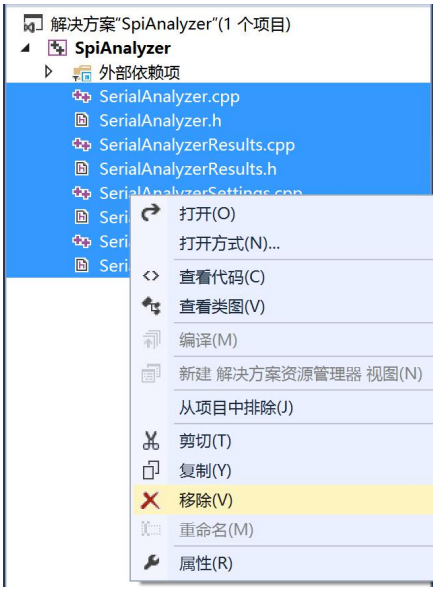


图 4

(4)、在工程上点击鼠标右键，选择“添加”->“现有项”，如图 5 所示，然后再打开“src”文件夹，将需要添加的.cpp 和.h 文件全部选中后点击“添加”，如图 6 所示；

图 5

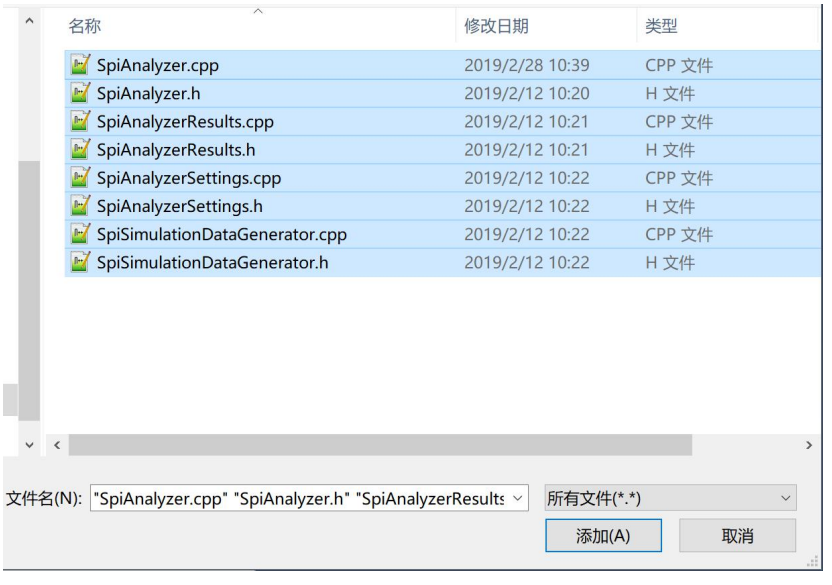


图 6

(5)、在完成上述步骤之后，工程就建好了，接下来就可以生成所需要的.dll 文件了，在 Relesse 模式下生成 32-bit 或者 64-bit 的.dll 文件，根据您所使用的操作系统来决定。生成的.dll 文件名可以在 VS2013 中设置，通过菜单“项目”->“[项目名称]属性”->“配置属性”->“常规”->“目标文件名”进行更改，如图 7。如不做更改则默认与工程名称相同。如果使用更高版本的 Visual Studio 则还需修改其中的“平台工具集”这一项目。

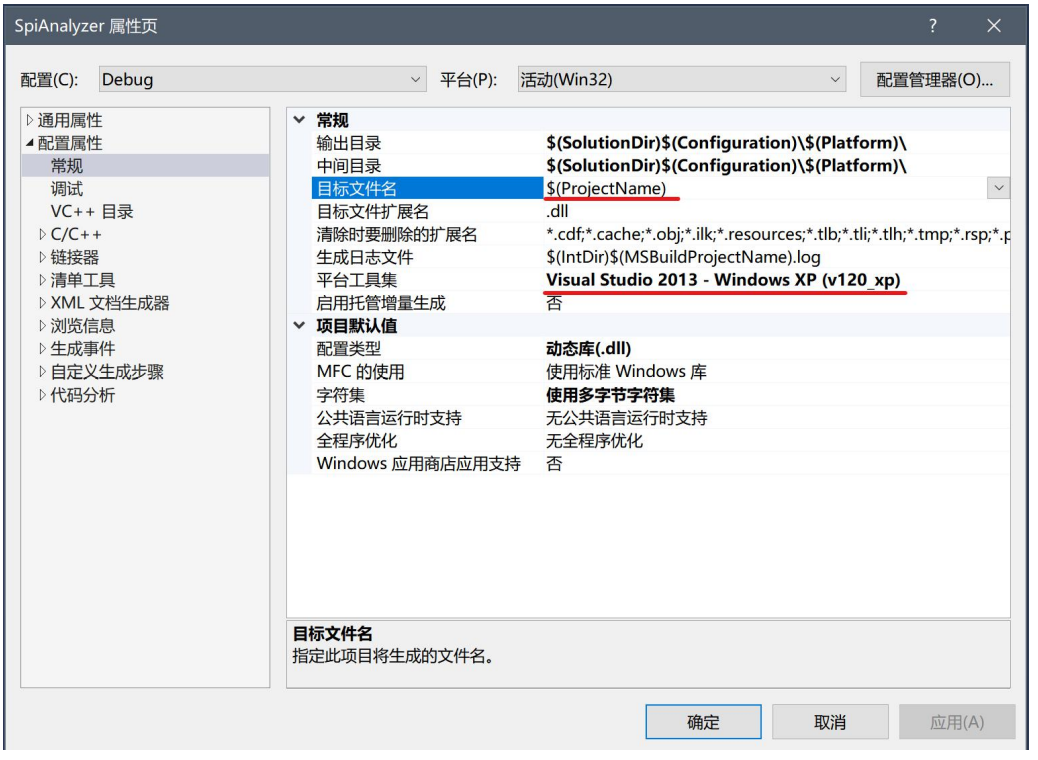


图 7

生成 32-bit 文件时，需配置为 Release + Win32，如图 8 所示；生成 64-bit 文件时，需配置为 Release + x64，如图 9 所示；

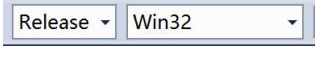


图 8

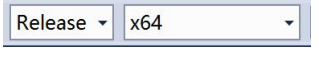


图 9

(6)、生成的 32-bit 的.dll 文件位于“\Release\Win32\”目录下，64-bit 文件则位于“\Release\x64\”目录下。

(7)、将生成的.dll 文件拷贝到 KingstVIS 软件安装目录下的“Analyzer”目录下，重新打开 KingstVIS 软件，您自定义的协议解析器就会出现在解析器列表中了。

2.2. Linux

(1)、以生成解析 SPI 协议所需动态库为例进行讲解，可以直接在样例“SerialAnalyzer”的基础上进行修改以得到 SPI 解析器的工程，如果想要保留这个样例，可以将该样例复制一份在同一目录下，然后在复制出来的文件夹中进行修改；

(2)、首先修改文件夹的名称，可以参考样例进行修改，将文件夹名修改为“SpiAnalyzer”，然后打开这个文件夹，将“src”文件夹中的源文件替换为用于解析 SPI 的源文件，具体源文件的创建将在下一章节中进行讲解，再打开“Linux”文件夹，打开其中的 makefile 文件，将

文件中的 libSerial.so 修改为您所需要的名称，例如我们修改为“libSPI.so”；

(3)、打开终端，进入到“Linux”目录下，在终端上输入 make 命令，就会在当前目录下生成“libSPI.so”动态库文件；

(4)、将生成的.so 文件拷贝到 KingstVIS 软件安装目录下的“Analyzer”目录下，重新打开 KingstVIS 软件，您自定义的协议解析器就会出现在解析器列表中了。

2.3. Mac

(1)、以生成解析 SPI 协议所需动态库为例进行讲解，可以直接在样例“SerialAnalyzer”的基础上进行修改以得到 SPI 解析器的工程，如果想要保留这个样例，可以将该样例复制一份在同一目录下，然后在复制出来的文件夹中进行修改；

(2)、首先修改文件夹的名称，可以参考样例进行修改，将文件夹名修改为“SpiAnalyzer”，然后打开这个文件夹，将“src”文件夹中的源文件替换为用于解析 SPI 的源文件，具体源文件的创建将在下一章节中进行讲解，再打开“Mac”文件夹，打开其中的 makefile 文件，将文件中的 libSerial.dylib 修改为您所需要的名称，例如我们修改为“libSPI.dylib”；

(3)、打开终端，进入到“Mac”目录下，在终端上输入 make 命令，就会在当前目录下生成“libSPI.dylib”动态库文件；

(4)、将生成的.dylib 文件拷贝到 KingstVIS 软件安装目录下的“Contents/Resources/Analyzer/”目录下，重新打开 KingstVIS 软件，您自定义的协议解析器就会出现在解析器列表中了。

(5)、Mac 下可能会出现因找不到依赖库 libAnalyzer.dylib 而导致生成的.dylib 文件无法加载的情况，可以使用以下命令修改依赖库路径。

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

二、 编写自己的解析器代码

要创建自己的协议解析器，需要自己实现 4 个.cpp 文件和对应的 4 个.h 文件，可以在 SDK 包中样例的基础上进行修改，以简化操作。

一般来说，协议解析可以被划分为 4 个主要部分，对应 4 组 c++文件。建议您按照一定的顺序进行代码的编写工作，下面将会按照这个顺序进行介绍。

首先要实现的是 AnalyzerSettings 的派生类，你需要定义解析所需的设置，而且要创建 KingstVIS 软件用以显示的接口，以及保存和恢复设置所需的数据流。

下一步要实现的是 SimulationDataGenerator 类，这里可以生成协议所需的模拟数据，方便测试和调试。

第三步要创建的是 AnalyzerResults 的派生类，它会将保存的解析结果转换为文本，可用于多种用途，在这里你需要考虑这些转换的文本要以什么样的格式呈现出来。

最后要实现的是 Analyzer 的派生类，主要是根据具体的协议规范，将采样得到的数据流转换为解析结果。

下面我们开始具体介绍！

以编写 Serial 协议（异步串行协议 UART）解析器为例。首先建立 4 个.cpp 和 4 个.h 文件，4 个.cpp 文件为 SerialAnalyzer.cpp、SerialAnalyzerResults.cpp、SerialAnalyzerSettings.cpp、SerialSimulationDataGenerator.cpp；4 个.h 文件为 SerialAnalyzer.h、SerialAnalyzerResults.h、SerialAnalyzerSettings.h、SerialSimulationDataGenerator.h。先让这些文件空着，等后续再一一实现。在编写自己的解析器时，也推荐按照这种命名模式去创建文件。

1、 AnalyzerSettings 派生类实现

将创建好的 4 个.cpp 文件和 4 个.h 文件添加到已创建好的工程中，这样一个关于解析器的工程就建立完毕，接下来就需要依次去完成这 8 个文件。

1.1. {YourName}AnalyzerSettings.h

首先要实现的是 AnalyzerSettings 类的派生类，实现这个派生类，代码要在 SerialAnalyzerSettings.cpp 和 SerialAnalyzerSettings.h 中编写，这个派生类的名字可以是 SerialAnalyzerSettings，因为该类继承自 AnalyzerSettings，所以在 SerialAnalyzerSettings.h 这个文件中需要包含 AnalyzerSettings 的头文件。

SerialAnalyzerSettings.h 这个文件中的具体代码如下：

```
#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
```

此外在这个类中还将定义两组变量：一组是在类中定义的设置变量；另一组是为定义的设置变量提供可以修改的接口。

1.1.1. 设置变量

上面这段代码所展现出来的，就是在编写解析器时继承自 AnalyzerSettings 的派生类所要包含的内容。这个派生类中不只是这些函数，也可以向这个类里面添加其它成员，这个类中应该至少包含类变量 Channel，用户可以据此指定要使用的通道，通道不能指定为常量，而是可设置的，这个 Channel 变量不仅仅是索引，还表示与逻辑分析仪那个通道相连接。其它可用设置变量取决于你所实现的协议，其中可能包括：

- 波特率
- 每次传输的位数
- 位传输顺序（MSB 在前或 LSB 在前）
- 时钟边沿（上升沿、下降沿）
- 使能信号的高低电平选择

变量类型没有限制，例如 std::string、double、int、enum 等，这些变量需要进行串行化处理（稍后用于保存到文件中）。SDK 提供了进行串行化和保存变量的方法。

向 SerialAnalyzerSettings 类中添加的设置变量如下所示：

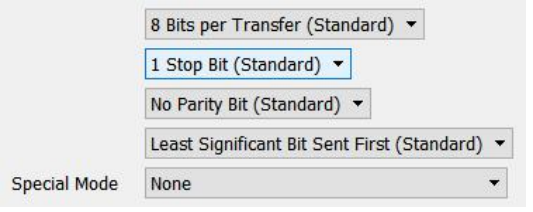
```
    Channel mInputChannel;
    U32 mBitRate;
    U32 mBitsPerTransfer;
    AnalyzerEnums::ShiftOrder mShiftOrder;
    double mStopBits;
    AnalyzerEnums::Parity mParity;
    bool mInverted;
    bool mUseAutobaud;
    SerialAnalyzerEnums::Mode mSerialMode;
```

1.1.2. 修改设置变量的接口

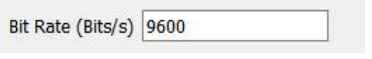
协议解析 SDK 的一个作用就是在 GUI 界面中提供可以修改类中设置的方法。要达到这个目的，每个设置变量都要具有一个相应的接口，这就需要使用 AnalyzerSettingsInterface 类，下面是 AnalyzerSettingsInterface 可以使用的类型：

AnalyzerSettingInterfaceChannelData，用于输入通道的选择。

AnalyzerSettingInterfaceNumberList，提供用户可以选择的数字选项列表。注意，这也可以用于从几个枚举类型中进行选择，如下所示。（下面的每个下拉列表都是用自己的接口对象实现的）

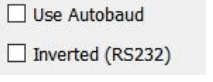


AnalyzerSettingInterfaceInteger，供用户输入整数。



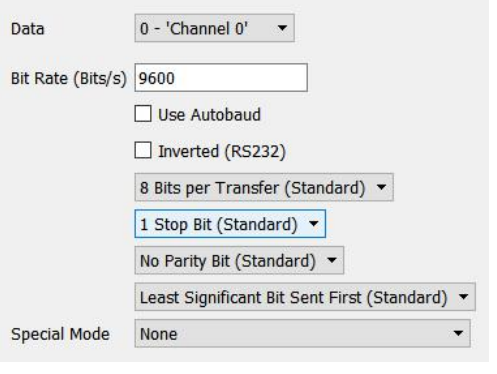
AnalyzerSettingInterfaceText：供用户输入一些描述性的文本。

AnalyzerSettingInterfaceBool：单选框。



AnalyzerSettingInterfaceChannelData、AnalyzerSettingInterfaceNumberList、AnalyzerSettingInterfaceInteger、AnalyzerSettingInterfaceText、AnalyzerSettingInterfaceBool 这 5 个类全部继承自 AnalyzerSettingsInterface，在使用这 5 个类创建对象指针时使用了“智能指针” std::auto_ptr，当指针离开作用域时会自动调用 delete 删除这个指针，这样就不用担心内存泄漏的问题。

对于 Serial 这个解析器，我们要想实现的 GUI 设置界面如下图所示。



所以应该向 SerialAnalyzerSettings 这个类中，添加如下一段代码：

```
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;
```

在向 SerialAnalyzerSettings 类中添加完上面这段代码后，SerialAnalyzerSettings.h 中的代

码就编写完成了。

1.2. {YourName}AnalyzerSettings.cpp

在完成 SerialAnalyzerSettings.h 文件的内容后，就需要去完成 SerialAnalyzerSettings.cpp 这个文件，需要对 SerialAnalyzerSettings 这个类中的构造函数、析构函数等进行实现。

1.2.1. 构造函数

首先要完成构造函数的编写，在构造函数中要对变量进行初始化，对于 Serial 这个解析器，初始化变量的代码编写如下：

```
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)
{
}
```

在初始化变量之后，就要去创建接口，由于使用的 std::auto_ptr，所以创建时调用的是其成员函数 reset。

①. AnalyzerSettingInterfaceChannelData

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

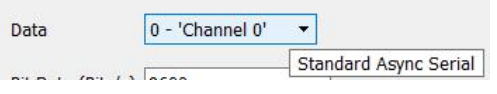
接下来，通过调用成员函数 SetTitleAndTooltip，可以给这个通道设置接口起一个名字和当鼠标停留在输入元素上时出现的提示，名字会显示在输入元素的左侧。注意，有的时候设置接口可以没有名称，但是通道一般都应该有名称。这个函数有两个输入参数，title 是名称，tooltip 是提示信息。

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

例如，对于 Serial 的通道设置接口，可以做下面的设置。

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

接口名字是 Data，提示信息是 Standard Async Serial，显示结果如下图：



接下来设置通道，可以调用 SetChannel 函数。

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

可以通过调用 GetChannel 函数获得设置的通道。

```
    Channel GetChannel();
```

有时候通道可以设置为 None，即可以不与逻辑分析仪通道相连接，这种情况只会出现在多通道下，当协议为单线时必须得设置一个输入通道。

例如，解析 SPI 协议时，设置通道选择，如下图：

这就表示“Enable”可以不与逻辑分析仪通道相连接。

通道是否可以设置为 None，需要调用 SetSelectionOfNoneIsAllowed 函数进行设置。

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

is_allowed 为 true 时表示该通道可以设置为 None，为 false 时必须要设置一个输入。默认为 false，即每个通道必须设置一个输入。

可以通过调用 GetSelectionOfNoneIsAllowed 函数获得该通道是否被允许设置为 None。

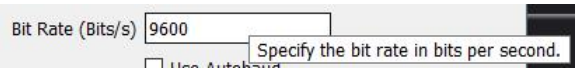
```
void SetSelectionOfNoneIsAllowed(bool is_allowed);
```

②. AnalyzerSettingInterfaceInteger

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

同样是调用 SetTitleAndTooltip 函数设置接口名称和提示信息，该函数与设置 Channel 中的说明相同。

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



输入的数必须是整数，可以通过调用 SetMax 和 SetMin 函数来设置数据的有效范围。

```
void SetMax(int max);
void SetMin(int min);
mBitRateInterface->SetMax(100000000);
mBitRateInterface->SetMin(1);
```

通过上述设置，输入数据的范围被限制在 1 到 100000000 之间。

设置输入数据可以通过调用 SetInteger 实现。

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

已输入的数据可以通过调用 GetInteger 函数获得。

```
int GetInteger();
```

③. AnalyzerSettingInterfaceNumberList

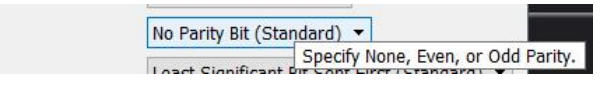
此例仅通过对 Serial 解析器中的奇偶校验设置来说明 AnalyzerSettingInterfaceNumberList 类的使用。

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

接着设置接口名称和提示信息，同样需调用 SetTitleAndTooltip，

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

此处的接口没有设置名称，只是设置了提示信息。



通过调用 AddNumber 来向该接口下拉列表中添加选择项。

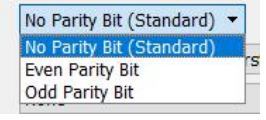
```
void AddNumber(double number, const char *str, const char *tooltip);
```

该函数有三个输入参数：number 与选项相关联，不会对用户显示；str 是选项的名称，会出现在列表中；tooltip 是提示信息，当鼠标悬停在该选项上时会出现提示信息。

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

当在列表中选中想要设置的选项后，是通过调用 SetNumber 函数将设置的数据保存下来。

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



通过 SetNumber 设置的值，可以通过 GetNumber 函数获取。

```
double GetNumber();
```

④. AnalyzerSettingInterfaceBool

此处以设置 Serial 解析器是否启用自动波特率为例，说明 AnalyzerSettingInterfaceBool 类的使用，同样是先创建一个 AnalyzerSettingInterfaceBool 的指针，需调用 reset 函数。

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

设置接口名称和提示信息。

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

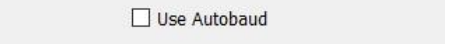
可以调用 SetCheckBoxText 函数为单选框设置一个名称，这个名称的显示与调用 SetTitleAndTooltip 函数设置的名称显示方式不一样，通过 SetCheckBoxText 函数设置的名称显示在单选框的右侧。

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

该单选框设置项的启用与否通过调用 SetValue 函数来设置。

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

value 为 ture 时启用单选框设置项，为 false 则不启用。



在 GUI 设置界面上，如要启用“Use Autobaud”这个功能则需用鼠标点击选中。

可通过 GetValue 函数获取是否启用了该单选框设置项。

```
bool GetValue();
```

在创建完接口、指定标题和提示信息、指定可用的选择项、设置默认值后，我们需要将它们提供给 API，可以使用下面的函数。

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

对于 Serial 解析器需要提供给的接口如下所示：

```
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());
```

⑤. 指定解析文件导出选项

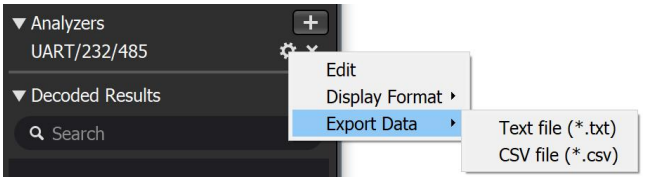
解析提供的输出类型可能不止一种，比如 txt、 csv、甚至是 wav 文件或位图等。

输出选项被指定了一个 ID，在调用生成输出数据的函数时，需要提供该 ID。在指定输出类型时需要调用两个函数，一定要指定至少一种输出类型（一般为 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 为指定的 ID，extension_description 为输出文件类型的描述信息，extension 为输出文件的扩展名。

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



⑥. 指定所使用的通道

协议解析必须要指定所用的通道，通道由 AddChannel 函数指定。每次通道改变时（如用户修改通道），涉及到的通道必须要更新。要清除之前已经设置过的通道，需调用 ClearChannels 函数。

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

请注意在构造函数中，需要将 is_used 指定为 false。这是因为所有通道都默认设置为了 UNDEFINED_CHANNEL。在用户将通道修改为 UNDEFINED_CHANNEL 外的其它值时，应该指定为 true。

1.2.2. 析构函数

在 AnalyzerSettings 类的析构函数中，一般什么也不用添加。

1.2.3. bool {YourName}AnalyzerSettings::SetSettingsFromInterfaces()

从函数名称就可以看出，该函数会将保存在接口目标中的数据复制到设置变量中去，若用户要更新数据，则需要调用该函数。

同时还可以在该函数中检查用户设置的值是否合理，如果不合理则应该拒绝用户的设置，这种情况下，应该返回 false，否则返回 true。如果返回 false，则需要调用 SetErrorText 函数来指示原因，这将在弹出的对话框中呈现给用户。

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

例如，当解析器需要设置多个通道时，通常要确保所有通道都不相同，这时你可以调用 AnalyzerHelpers::DoChannelsOverlap 函数来判断要设置的通道是否和已经设置过的通道相同，如果相同就说明设置不合理，应该给出提示。

当然，您在写自己的解析器时可以忽略上面的检查，只要您能确保设置的正确就行了。

在为设置变量分配接口数据后，还需要更新协议解析器正在使用的通道，以下是 SerialAnalyzerSettings.cpp 中的例子。

```
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. void {YourName}AnalyzerSettings::UpdateInterfacesFromSettings()

该函数会将设置变量的数据更新到接口中去，以下是 SerialAnalyzerSettings.cpp 中的例子。

```
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)

当用户软件关闭后，再次打开时，软件会将之前使用的解析器自动添加上，故在软件关闭时需要保存之前解析器中的设置。设置内容是保存在字符串中的，所以需要将所需的变量放到字符串中，我们提供了一种将变量进行串行化的机制，下面将会对这一机制进行介绍。

首先，需要使用一个 SimpleArchive 对象，它能执行串行化操作。通过 SetString 函数将字符串赋值给 SimpleArchive，该字符串为 LoadSettings 函数的输入参数。

```
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;
};
```

下一步我们可以使用 SimpleArchive 中的 >> 运算符加载所有的设置变量。

由于通道值可能会改变，还需要更新正在使用的通道，每次设置改变时都应如此处理。

最后，调用 UpdateInterfacesFromSettings 函数，这样会以新加载的数值更新所有的接口。

以下是 SerialAnalzerSettings.cpp 中的例子。

```
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()

该函数会将所有的设置变量保存在一个字符串中，我们需要使用 SimpleArchive 来对数据进行串行化处理。

保存数据的顺序必须要同 LoadSettings 提取时完全一致。

返回时使用 SetReturnString 函数，这样在函数结束时，该函数提供的指针所指向的位置还会存在。

以下是 SimpleSerialAnalzerSettings.cpp 中的例子。

```
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());
```

```
}
```

至此就完成了{YourName}AnalzerSettings.cpp 和{YourName}AnalzerSettings.h 这两个文件，实现了 AnalzerSettings 的派生类，这个派生类的主要功能是给 GUI 设置界面提供设置接口，完成在解析协议时所使用到的设置功能。

2、 SimulationDataGenerator 类实现

在创建完{YourName}AnalyzerSettings 文件后，下一步要实现 SimulationDataGenerator 类。需要完成{YourName}SimulationDataGenerator.h 和{YourName}SimulationDataGenerator.cpp 这两个文件。

SimulationDataGenerator 类可以提供用于测试的数据，一般说来，模拟数据应该符合用户设置，这样可以测试多种已知条件下的情况。另外，通过模拟数据可以在软件上产生波形，并可以对模拟数据进行解析，解析结果可以显示在软件界面上。

2.1. {YourName}SimulationDataGenerator.h

除了构造函数和析构函数外，所需要的还有两个函数和两个变量，当然你可以添加其它的函数和变量，以帮助实现你的模拟数据。以下是 SerialSimulationDataGenerator.h 中的例子。

```
#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
```

SimulationDataGenerator 的关键之处在于 SimulationChannelDescriptor 类，每个模拟的通道都需要一个实例（本示例只需模拟一个通道）。在调用 GenerateSimulationData 函数时，将会生成一定数量的模拟数据，当产生的模拟数据达到需要的数量后，你要给调用者提供所实现的 SimulationChannelDescriptor 数组的指针。

2.2. {YourName}SimulationDataGenerator.cpp

2.2.1. 构造函数/析构函数

可以让构造函数和析构函数为空，而且在构建的时候也不知道要模拟的协议有什么样的设置，所以这个时候没有要处理的事情。

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

该函数提供在开始模拟数据时的状态，需要把从该函数传入的参数保存下来。

首先，将 simulation_sample_rate 和 settings 保存到成员变量中。在传入的参数中有个指向 AnalyzerSettings 的派生类的指针，通过该指针就可以知道具体的设置内容，例如使用了那个通道、设置了哪些参数等，在模拟数据的时候需要用到。

然后，初始化 SimulationChannelDescriptor 实例的状态，如设置所用通道、采样频率以及模拟数据初始位状态（高或低）等。

为了让方便您编写后续代码，先来介绍几个概念。

BitState

BitState 为 SDK 中常用的一个类型，其可以为 BIT_LOW 或 BIT_HIGH，代表通道的逻辑状态。

采样频率（每秒的采样点数）

采样频率表示每秒采样的点数，其通常表示采集数据的速度，不过对于模拟来说，它代表生成采样数据的速度。

采样数量

从 0 开始，采样的绝对数量。当数据采集开始后，采样到的第一个数据为样本 0，模拟时也是一样。

SimulationChannelDescriptor

需要使用这个类模拟出一个通道的数据，根据模拟的数据可以在软件界面上绘制出协议的波形。现在我们来看一下具体如何实现模拟数据。

指定通道初始状态（BIT_LOW 或 BIT_HIGH）；

往前移动一些采样数据，然后将其翻转；

再往前移动一些采样数据，然后再将其翻转，一直持续这种操作，直到模拟数据量达到需要的数据量。

通道的初始位状态不会改变。可以通过起始到该点所经过的翻转次数（偶数或奇数次）来确定特定采样点的状态（高或低）。

另一种更加具体点的描述，如下：

最开始时，指定初始状态（BIT_LOW 或 BIT_HIGH），也就是采样数量为 0 时的状态

下一步，我们向前移动一些采样数据

下一步，翻转通道（低变高、高变低）

下一步，再向前移动

下一步，再翻转

继续移动和翻转

下面来看一下实现这些操作的函数：

```
void Advance(U32 num_samples_to_advance);
```

利用该函数向前移动模拟波形，即增加采样数量，例如开始时为采样数量 0，在调用 Advance(10)函数 3 次后，采样数量就变为 30。

```
void Transition();
```

翻转通道，即把通道由低电平（BIT_LOW）变成高电平（BIT_HIGH）或者由高电平（BIT_HIGH）变成低电平（BIT_LOW）。当前的采样数量的电平状态（BitState）会变成的新的 BitState（BIT_LOW 或 BIT_HIGH），在我们再次翻转之前，其后的采样数量的 BitState 也会是新的 BitState。

```
void TransitionIfNeeded(BitState bit_state);
```

调用 TransitionIfNeeded 时可以把你传入的 bit_state 与当前的 BitState 比较，如果两者的状态一样，则不用改变通道当前的 BitState，反之，则需要将当前的 BitState 翻转。

```
BitState GetCurrentBitState();
```

查询当前的 BitState，即当前的电平状态。

```
U64 GetCurrentSampleNumber();
```

获取当前的 SampleNumber，即采样数量。

ClockGenerator

ClockGenerator 是 AnalyzerHelpers.h 中提供的一个类，它可以让你输入时间值，然后得到采样点数。

首先，使用 ClockGenerator 类创建一个对象，然后调用 Init 函数进行初始化。

```
void Init(double target_frequency, U32 sample_rate_hz);
```

target_frequency 这个参数是你想要模拟数据的频率（单位 Hz），比如 Serail 协议的波特率、SPI 时钟的波特率等；sample_rate_hz 这个参数是生成数据的采样率，就是通过软件界面上设置的采样率。

```
U32 AdvanceByHalfPeriod(double multiple = 1.0);
```

本函数返回移动半个周期所需的采样点数，以半个周期为基本单位。例如，要移动一个周期时需要输入 2.0。

```
U32 AdvanceByTimeS(double time_s);
```

本函数返回移动 time_s 时间所需的采样点数，时间单位为秒，输入 1e-6 表示 1 微妙。

以下是 SerialSimulationDataGenerator.cpp 中的例子。

```
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)

在生成所需的模拟数据时，需要反复调用本函数。另外，为了简化处理，你可以生成比所需稍多些的模拟数据。

调用完该函数后，SimulationChannelDecriptor 中的采样数量和 largest_sample_requested 相比必须要相等或更大。

参数 simulation_channels 给调用者提供了指向 SimulationChannelDecriptor 目标的指针。我们可以在函数的最后设置该指针，该函数的返回值为数组中的元素数量，也就是通道的数量。本函数的主要任务是生成模拟数据，我们可以在循环中进行处理，在生成足够的数

据后退出循环。

以下是 SerialSimulationDataGenerator.cpp 中的例子。

```
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));
}
```

要模拟生成比较精确的模拟数据，一种方法是使用 ClockGenerator 类预先生成一个向前移动 1bit 需要的采样点数量的数组，这样在调用 Advance 函数向前移动 1bit 时，可以从这个数组中查找要移动的采样点数量；另一种方法就是利用 DataExtractor 类处理位的掩码，上面的例子就是这么处理的。

2.2.4. 模拟多通道数据

模拟多通道需要多个 SimulationChannelDescriptor，而且必须要位于一个数组中，最好的办法是利用辅助类 SimulationChannelDescriptorGroup。

以下是 I2C（2 个通道）的一个例子，定义在 I2cSimulationDataGenerator.h 中：

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

在 Initialize 函数中：

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

在 GenerateSimulationData 函数中调用 SimulationChannelDescriptor 指针数组的方法为：

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

可以单独使用每个 SimulationChannelDescriptor 指针，调用 Advance、 Transition 等函数实现模拟数据，或者将它们作为一个组进行操作，使用 SimulationChannelDescriptorGroup 的 AdvanceAll。

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

在从 GenerateSimulationData 返回前，应该确保 SimulationChannelDescriptor 的采样数量大于 adjusted_largest_sample_requested。

生成模拟数据实例如下：

```
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();
}
```

在前面的例子中我们使用了多个帮助函数和类，下面我们来简单介绍一下 BitExtractor。

BitExtractor

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

对于一些协议来说，字中的位数是可变的，而且可以设置最高位是第一个还是最后一个。这样操作起来比较麻烦，因此我们构建了 BitExtractor 类。当然，你也可以自己实现，不过使用这个类可以使代码更加简洁。

DataBuilder 和 BitExtractor 类似，不过功能相反。

AnalyzerHelpers

有些静态辅助函数也可能会很有帮助，其中包括：

```
static bool IsEven(U64 value);
```

偶校验判断。

```
static bool IsOdd(U64 value);
```

奇校验判断。

```
static U32 GetOnesCount(U64 value);
```

获取数据位数。

```
static U32 Diff32(U32 a, U32 b);
```

获取两个数的差，a-b。

3、 AnalyzerResults 派生类实现

在创建了 SimulationDataGenerator 类后，就需要实现 AnalyzerResults 的派生类了，即{YourName}AnalyzerResults 类，这需要完成{YourName}AnalyzerResults.cpp 和{YourName}AnalyzerResults.h 文件。AnalyzerResults 派生类用于将解析结果转换为显示和输出文件的信息等。

提示：因为只有在完成了{YourName}Analyzer.cpp/.h 这个两个文件之后，才能确定你保存的解析结果是什么样的，这样才能更好的使用 AnalyzerResults 派生类生成所需的显示和输出信息，所以可以先简单完成 AnalyzerResults 派生类，等完成{YourName}Analyzer.cpp/.h 这两个文件后再次回到这个类，继续完成所要实现的功能。

3.1. {YourName}AnalyzerResults.h

在这个派生类中，除了构造和析构函数外，还需要另外 5 个函数。

以下是 SerialAnalyzerResults 的头文件，您自己实现的头文件与之类似，区别一般为所需的 enum 和 define 定义。

```
#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. 构造和析构函数

在构造函数中，保存 Analyzer 和 Settings 的原始指针，对构造函数或析构函数通常不需要做其它的事。以下是 SerialAnalyzerResults.cpp 中的例子。

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

协议解析产生结果的最小单位为 Frame，它可以是串行数据字节、CAN 数据包头或者 SPI 的 MOSI 以及 MISO 的 8 位数据等。I2C 的起始位和停止位等更小元素也可以被保存为 Frame，不过保存为图形项（名为 Marker）则会更好。Frame 的组合为 Packet，Packet 的组合为 Transaction。

Frame 是需要重点关注的，其实际表示的内容由您自己决定，不过若协议并非特别复杂（如 USB、 CAN 等），可以将 Frame 作为主要的结果元素。

在后面谈到 Analyzer 时我们将会深入介绍下如何保存结果。

Frame

Frame 可用于保存结果，以下是 Frame 的定义：

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

Frame 表示一段时间内协议所传达的信息，成员变量 mStartingSampleInclusive 和 mEndingSampleInclusive 代表 Frame 的起始和结束的采样点数。**需要注意的是：**Frame 间不能重叠，不能共用同一个采样点。例如：若某 Frame 的结束点位于一个时钟边沿处，而新 Frame 要从此处开始，那么这个新 Frame 的 mStartingSampleInclusive 需要加 1。

另外，Frame 中可以有两个 64 位的数据，即 mDta1 和 mData2，例如：对于 SPI，其中一个数据用于 MISO，另一个则用于 MISO。一般情况下只需要一个数据。

mType 变量用于保存自定义的枚举值，表示 Frame 的类型。例如：CAN 具有许多种 Frame，如头、数据、CRC 等，异步串行数据则只有一种类型，因此不会使用这个成员变量。

mFlag 是为了提供适用于 Frame 的自定义标志。但请注意，这个标志不能是普通枚举类型的，而应该是可用于组合在一起的单个 bit。

例如，在异步串行协议中，有一个帧错误标志和一个奇偶校验错误标志。

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

这两个 flag 将会被保存在系统中，并在显示解析结果时产生错误或者警告提示。

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

GenerateBubbleText 函数用于生成显示在屏幕上的信息，您也可以先将本函数留空，待完成{YourName}Analyzer.cpp/.h 这两个文件后再实现本函数。

frame_index 参数提供 Frame 自身的索引值，例如：

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

显示解析结果的通道一般不会多余 1 个（SPI 是一个特例），这样需要显示结果信息的通道就在 channel 参数中指定。

display_base 参数指定了数据值显示的格式（二进制 Bin / 十进制 Dec / 十六进制 Hex / ASCII / ASCII & Hex），可以使用辅助函数来处理这种情况。

```
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);
```

对于上面的 GetNumberString，该函数可以将 number 转成指定格式的字符串。number 为要转化的数；display_base 为要转换成的格式；num_data_bits 为数的实际位数，例如：I2C 中总是 8；result_string 为转换成的字符串；result_string_max_length 设置字符串的最大长度。

根据实际可用的显示空间，显示信息可以是不同长度的字符串。您应该生成多种结果字符串，最简单的可能只需表示内容的类型（比如“D”代表数据），长些的可能会表示整个数据（比如“0xFF01”），也可能会非常长（比如“Left Channel Audio Data: 0xFF01”）。设置多个字符串是为了当软件显示解析结果时，对界面上的波形进行缩放后，会呈现出不同的解析信息，当显示区域短时显示短的字符串，长的时候显示长的字符串。

要将字符串提供给调用者，可以使用 AddStringResult 函数，要确保函数返回时字符串仍然存在。在增加新的字符串结果时，要调用 ClearResultStrings。

要合并多个字符串，可以给 AddStringResult 输入多个字符串。

```
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);
```

以下是异步串行协议 GenerateBubbleText 函数的例子：

```
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)

在用户要将解析结果输出到文件中去时，要调用本函数。同样也可以先将本函数留空，等完成{ YourName }Analyzer.cpp/.h 这两个文件再来完成此函数。

file 参数是解析结果保存文件的完整路径。

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

display_base 参数是解析结果的数据格式。

export_type_user_id 参数是与用户选择的导出文件关联的 id。您可以在 AnalyzerSettings 派生类的构造函数中指定这些选项（应该至少有一个）。如果您只有一个导出选项，您可以忽略此参数。

有时您还需要输出与特定结果相关联的时间（单位为秒）， 此时需要使用 GetTimeString 辅助函数，该函数需要采样频率和采样点数。

```
    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);
```

该函数可以将与结果相关联的时间转换成字符串，sample 为当前的采样点数；trigger_sample 为触发位置的采样点数；sample_rate_hz 为采样频率；result_string 为转换成的字符串指针；result_string_max_length 设置字符串的最大长度。

以下是 SerialAnalyzerResults.cpp 中的例子：

```
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 用于为软件界面右下方的解析结果列表生成所需的字符串，同样可以先把该函数留空，等稍后再完成。

frame_index 参数为 Frame 自身的索引值，例如：

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

display_base 参数指定了数据值显示的格式。

GenerateFrameTabularText 函数与 GenerateBubbleText 函数几乎相同，只不过 GenerateFrameTabularText 函数应该只生成一个字符串结果。理想情况下，字符串应该简洁，在正常（非错误）情况下应该尽量短。

以下是 SerialAnalyzerResults.cpp 中的例子：

```
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)

该函数尚未实现，留空即可。

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

该函数尚未实现，留空即可。

4、 Analyzer 派生类实现

Analyzer 的派生类是协议解析器的核心，我们利用这个类实时分析数据位并生成解析结果。

4.1. {YourName}Analyzer.h

除了构造函数和析构函数，还需要实现以下函数：

```
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);
```

还需要这些成员变量：

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

每个输入还需要一个 AnalyzerChannelData 指针，例如对于 SerialAnalyzer，我们需要：

```
AnalyzerChannelData *mSerial;
```

在开发自己的协议解析器时，根据实际需要，可能还要增加额外的成员变量和辅助函数。

以下是 SerialAnalyzer.h 中的例子：

```
#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. 构造函数

构造函数示例如下：

```
{YourName}Analyzer::{YourName}Analyzer()
```

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

请注意，这里你会调用基类的构造函数，新构建了 AnalyzerSettings 的派生类，并且将基类作为指针提供给 AnalyzerSettings 的派生类。

4.2.2. 析构函数

这个函数中只需要调用 KillThread 函数即可。

4.2.3. void {YourName}Analyzer::SetupResults()

在该函数中需要生成 AnalyzerResults 派生类的实例对象，同时要将该对象的指针保存到 Analyzer 派生类中，并且要将需要显示解析结果的通道传递给 AnalyzerResults 的派生对象。

例如在 SerialAnalyzer.cpp 中：

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

在 SpiAnalyzer.cpp 中：

```
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()

本函数是关键所在，在此处进行数据解析（后续有具体介绍）。

4.2.5. bool {YourName}Analyzer::NeedsRerun()

一般来说，该函数返回 false 即可。

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

本函数用于获取模拟数据，在此之前我们已经完成了一个专门用来生成模拟数据的类，这里我们则需要对其进行调用。

示例如下：

```
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()

通过此函数设置在解析此协议时，要达到的最小采样率。

以下是 SerialAnalyzer.cpp 中的例子：

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

如果 Serial 波特率设置为 9600，则采集此信号时需要设置的最小采样率为 9600*4 = 38400Hz，设置比这个值更大的采样率将会更好。

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

返回在 KingstVIS 软件界面中显示的该协议解析器的名称。

例如在 SerialAnalyzer.cpp 中：

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

4.2.9. const char *GetAnalyzerName()

返回与前面函数相同的字符串。

例如在 SerialAnalyzer.cpp 中：

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

4.2.10. Analyzer *CreateAnalyzer()

返回 Analyzer 派生类实例的指针。

4.2.11. void DestroyAnalyzer(Analyzer *analyzer)

删除 Analyzer 指针。

4.2.12. 解析过程中的细节

下面介绍 void {YourName}Analyzer::WorkerThread()这个函数中关键部分的一些细节。

如果先前未在 SetupResults 函数中创建新建 AnalyzerResults 的派生类，则需要首先新建 AnalyzerResults 的派生类。

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

需要将其指针提供给基类。

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

指定要显示结果的通道，通常只需一个通道（除了 SPI 的例子，MISO 和 MOSI 都需要显

示）。这里只需指定显示信息的通道，而其它（如显示 marker）不应在此指定。

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

可能还需要获得采样频率（保存在成员变量中）。

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

要访问采样数据，还需每个通道数据 AnalyzerChannelData 的指针，异步串行协议只需一个，而 SPI 则需要 4 个。

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

现在我们已经准备好进行数据解析了，并且要记录结果，让我们来依次进行这些任务。

①.首先是一条建议

在协议正常执行时一般都是非常简单的，考虑的异常情况越多，处理就越复杂。开始时越简单越好，不要试图建立一个非常精细、完美无缺的解析，特别是你对 API 还不够熟悉时。

②. AnalyzerChannelData

通过 AnalyzerChannelData 类，我们可以访问输入的数据。我们只能按照先后顺序访问数据，而不能进行随机访问。该类提供的函数，可以使的解析更加方便，在解析的时候要充分使用此类中的函数。

③. AnalyzerChannelData - 状态

若不确定当前数据的位置，以及输入是高是低，我们可以调用：

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

④. AnalyzerChannelData - 基本遍历

要遍历数据，可以使用三种方式。

U32 Advance(U32 num_samples);

该函数可以向前移动特定数量的采样点数，本函数将返回移动期间电平翻转的次数（由高变低或由低变高）。

U32 AdvanceToAbsPosition(U64 sample_number);

若要向前移动到特定的绝对位置，我们可以使用该函数，它也会返回移动期间电平的变化次数。

void AdvanceToNextEdge();

该函数向前移动到状态改变处，即下个信号边沿所在处，调用该函数之后，你可以调用 GetSampleNumber 函数来获取该处的采样点数。

⑤. AnalyzerChannelData - 高级遍历（向前无移动）

在开发协议解析器时，特定任务可能会进行更加复杂的遍历，下面是一些处理方法。

U64 GetSampleOfNextEdge();

该函数不会移动数据中的位置，函数返回下个边沿的采样点数。

bool WouldAdvancingCauseTransition(U32 num_samples);

该函数不会移动数据中的位置，函数返回移动一定数量的采样数据时是否会引起位状态的变化（由高变低或由低变高）。

bool WouldAdvancingToAbsPositionCauseTransition(U64 sample_number);

该函数不会移动数据中的位置，函数返回移动到特定位置时是否会引起位状态的变化（由高

变低或由低变高）。

⑥.填充和保存 Frame

利用 AnalyzerChannelData 类，我们可以移动通道数据并对其进行分析，下面我们来介绍下如何保存解析数据。

前面介绍 AnalyzerResults 派生类时谈到了 Frame，它是保存解析结果的基本单位，其中包括：

起始和结束时间（开始和结束的采样点数）

保存结果的 2 个 64 位数据

1 个 8 位类型变量，指定 Frame 的类型

1 个 8 位标志变量，指定结果的 Yes/No 类型

以下是 Frame 的使用实例：

```
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);
```

首先我们创建了一个 Frame，然后对其赋值。若有的数值不需要，则可以跳过。

mFlags 应该先置为 0，然后等后续满足设置的判断条件时，再设置为相应的值。

Frame 的有些部分需要正确赋值，特别是：

mStartingSampleInclusive

mEndingSampleInclusive

mFlags

要保存 Frame，使用 AnalyzerResults 继承类中的 AddFrame，请注意 Frame 需要一个个按照顺序依次添加，而且不能重叠。

在添加 Frame 后，立即调用 CommitResults，这样外部系统可以立即访问 Frame。

还要调用 ReportProgress，输入参数为已处理的最大采样数。

⑦.添加 Marker

Marker 是波形上的表示协议某些特性的标记，例如：在异步串行协议中，我们在每个数据 bit 上添加了一个小白点，表示该 bit 的值取自当前位置上波形的电平值。也可以用 Marker 表示协议出错以及时钟信号的上升和下降沿等。

需要指定放置 Marker 的位置、所处的通道以及使用哪种图形符号。

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

例如在 SerialAnalyzer.cpp 中：

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

目前可用的 Marker 类型包括：

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

与添加 Frame 相同，你需要依次添加 Marker。

Marker 只用作图形记号，不能用于产生显示信息以及输出文件等。

⑧. Packet 和 Trasactions

Packet 为 Frame 的组合，将 Frame 加入 Packet 中相当简单：

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

在添加 Frame 时，它会被自动添加到当前 Packet 中，在将所有所需 Frame 加到 Packet 中后，可以调用 CommitPacketAndStartNewPacket，有些情况下，特别是出错时，你可能需要启动一个新的 packet，此时需要调用 CancelPacketAndStartNewPacket。

请注意 CommitPacketAndStartNewPacket 会返回一个 Packet ID。

目前，Packet 仅在输出数据到 text/csv 时使用。在输出文件使用 Packet ID 时，应该使用 GetPacketContainingFrameSequential 函数，避免每次都重新检索所需的 Packet，GetPacketContainingFrame 会从头搜索，因此效率更低。