Contents

Setting up an Analyzer Project

This document is to split into two main parts. The first deals with setting up your analyzer project – getting the build environment set up on Windows, Mac, and/or Linux.

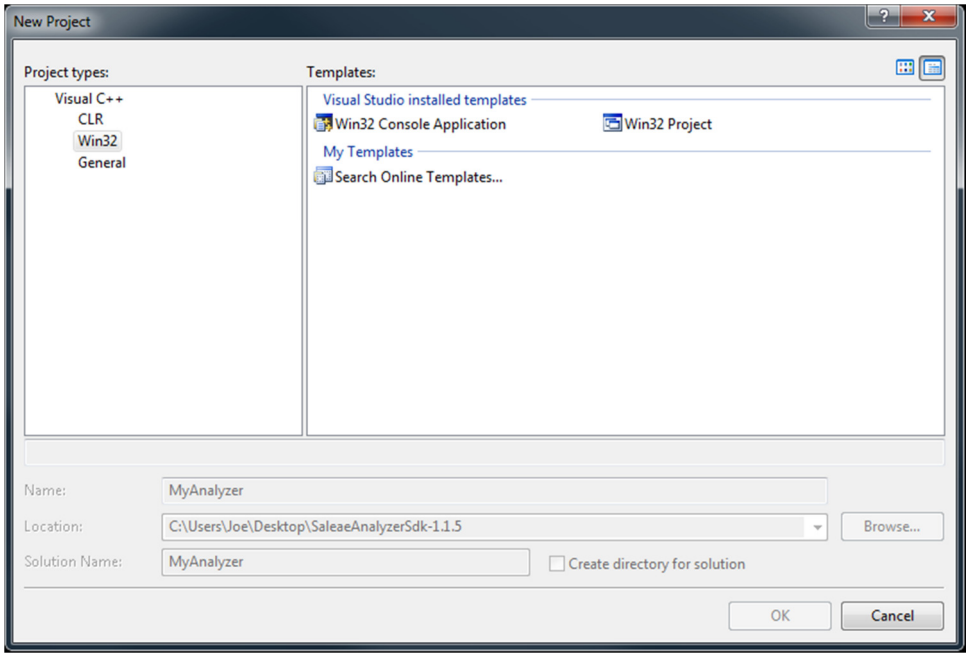
Then second part deals with writing the code, and the details of the SDK.

Visual Studio

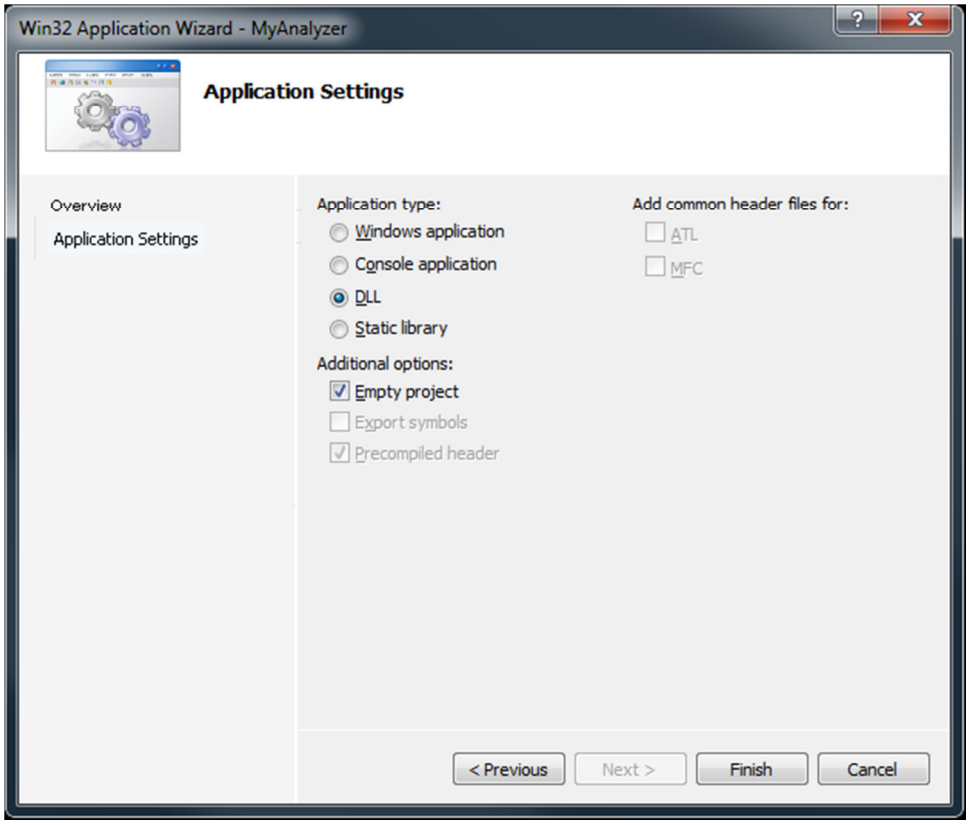
This section deals with setting up a Visual Studio project from scratch. The SDK already includes a Visual Studio project that should run build and run out of the box – but if you want the details this section is for you.

In addition, the following section discusses how to automatically change all the source files to match the name of your analyzer – which can be a nice way to get started on a new analyzer.

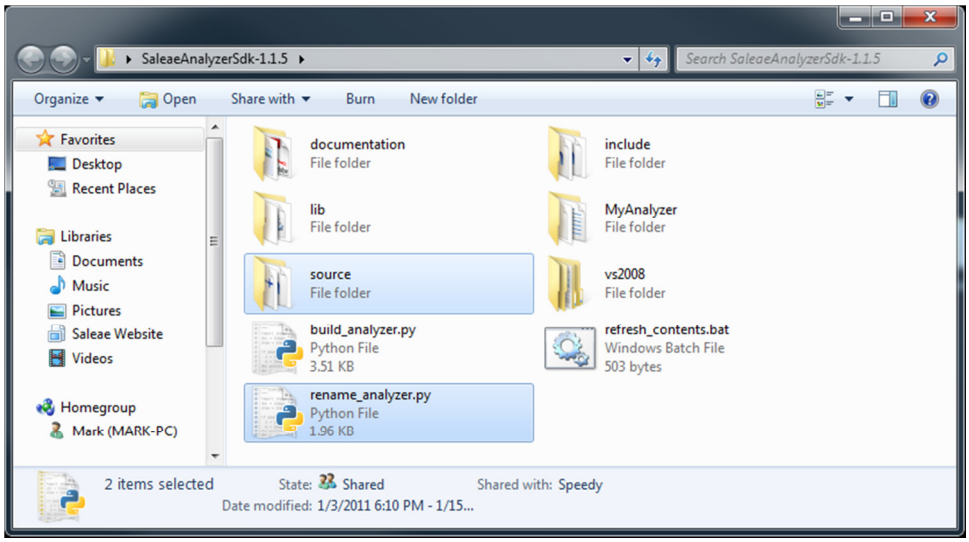
1. If you haven't already, download the latest *SaleaeAnalyzerSdk-1.1.x.zip* file and extract to this to your desktop, or other convenient location.
2. Launch VS2008 – we're using the C++ express version. http://www.microsoft.com/express/Downloads/ (as of the time of this writing, there is VS2008 tab still on the page).
    a. File->New-Project
    b. Visual C++; Win32
    c. Win32 Project (under templates)
    d. Name: **{YourName}Analyzer** (Example: SuperSerialAnalyzer)
    e. Location: *Desktop\SaleaeAnalyzerSdk-1.1.x*
    f. Make sure Create directory for solution is not checked.
    g. Press OK.

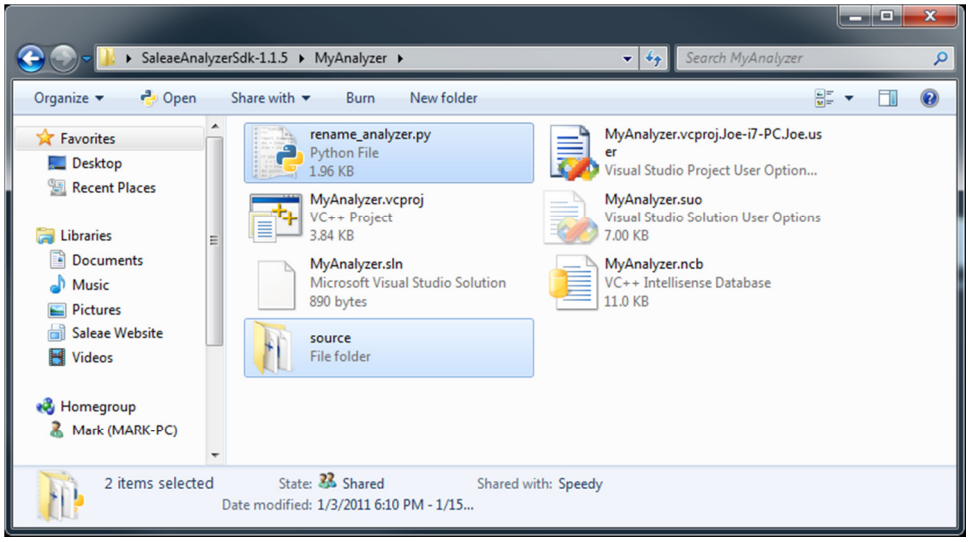


3. On the left-hand plane, click Application Settings
    a. Under Application type, select DLL
    b. Under Additional options, check Empty Project
    c. Press Finish

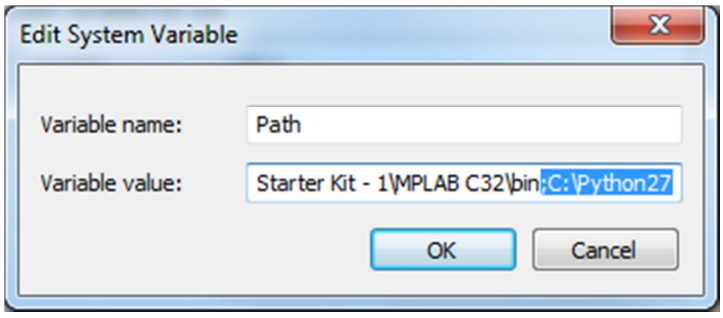


4. Next, let's get our source files in order. In the root of the *SaleaeAnalyzerSdk* folder there is a folder called *source*, and a file called *rename_analyzer.py*, copy and paste both of these into your new project folder.





a. Automatically changing the file names and contents to match the name of your analyzer is very helpful and saves a bunch of time and effort. The *rename_analyzer.py* script does exactly this. Unfortunately to run it you'll have to install Python 2.x. We'll do that now.
b. Go to http://www.python.org/download/ and download the latest 2.x Windows installer, and install it. Note that Python also has a 3.x version, but this is not as widespread yet and will not be compatible.
c. We'll want to add Python to our path environment variable.
    - On Windows 7 or Windows Vista:
        1. Start Menu->(right click on) Computer->Properties
        2. Choose Advanced System Settings. It'll be on the left sidebar.
        3. Click the Environment Variables… button at the bottom of the Advanced tab.
        4. Under System Variables, scroll down and double click the variable *Path*.
        5. At the end of the Variable value string, add **;C:\Python27** (a semicolon, and the absolute path of your Python installation)

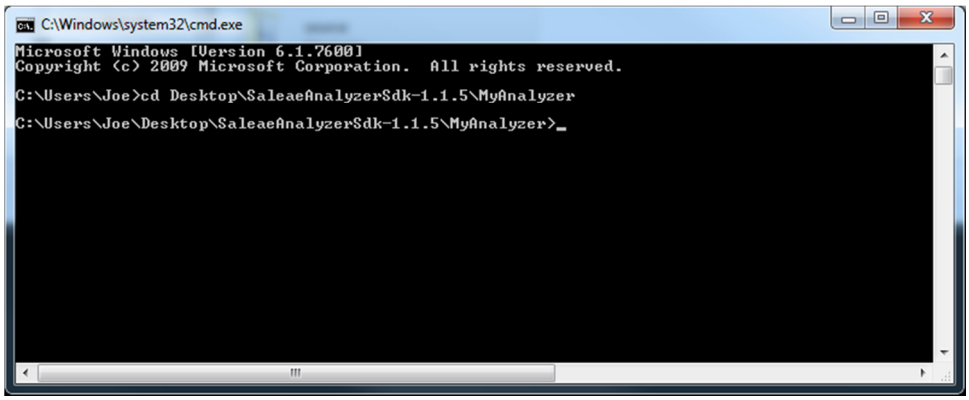


    - On Windows XP:
        1. Right click on My Computer, and then click Properties.
        2. Click Environment variables, on the Advanced tab.

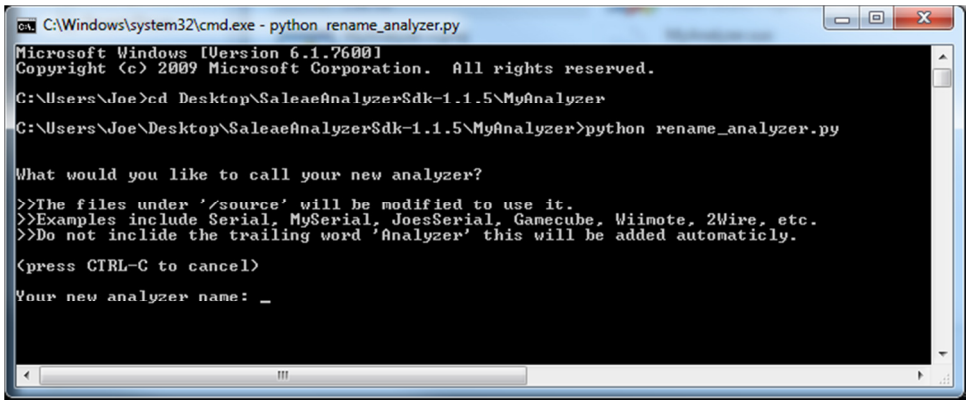
3. Under System Variables, scroll down and double click the variable Path.
4. At the end of the Variable value string, add **;C:\Python27** (a semicolon, and the absolute path of your Python installation)

d. Now we'll run the *rename_analyzer.py* script. Open a console (Start Menu ->Run, type **cmd**, press enter (Windows XP), or Start Menu->Search programs and files (text box), type **cmd** and press enter (Windows Vista & 7).

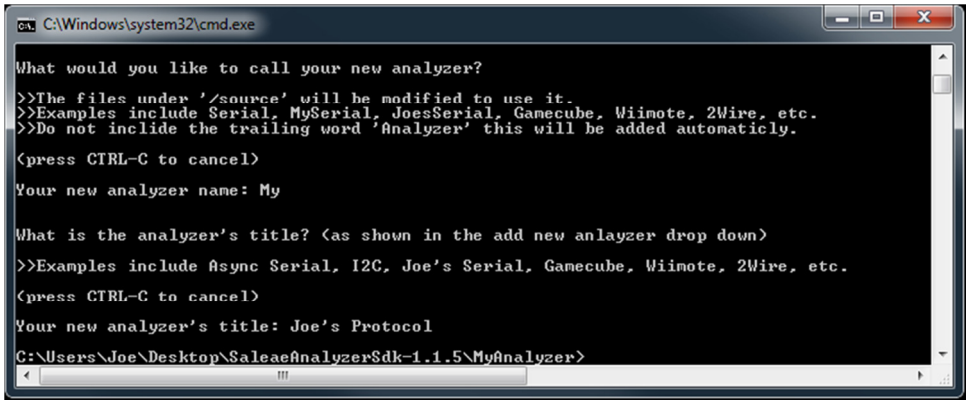
- Navigate to your new project folder: **cd Desktop\SaleaeAnalyzerSdk-1.1.x\YourProject**



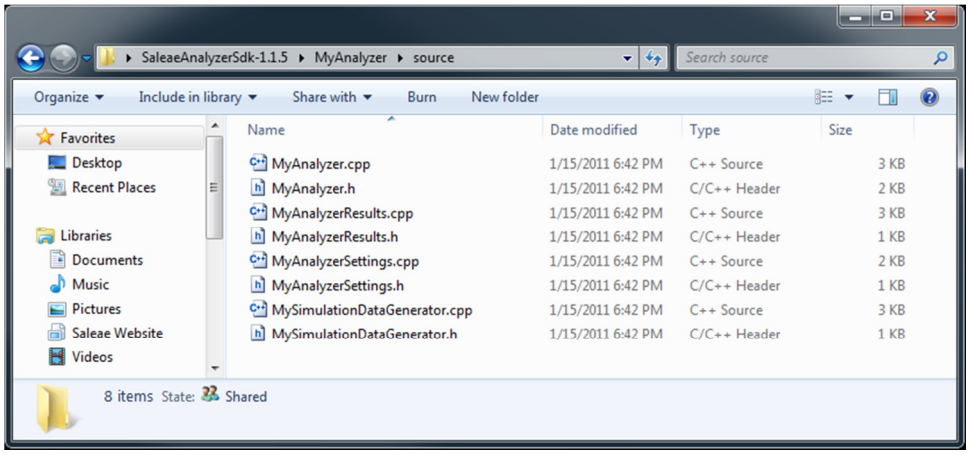
- Type: **python rename_analyzer.py**



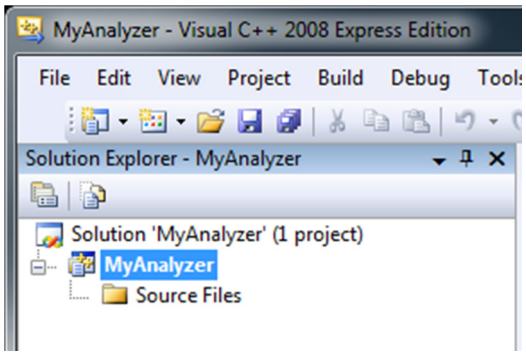
- Follow the instructions – enter your analyzer name (this should match your project name you created earlier). Do not enter "Analyzer" at the end – this is implied, and will be added automatically. For example, for *SimpleSerialAnalyzer*, enter **SimpleSerial**.
- Enter the title to display in the Add Analyzer dropdown in the Logic software. This is really easy to change later.



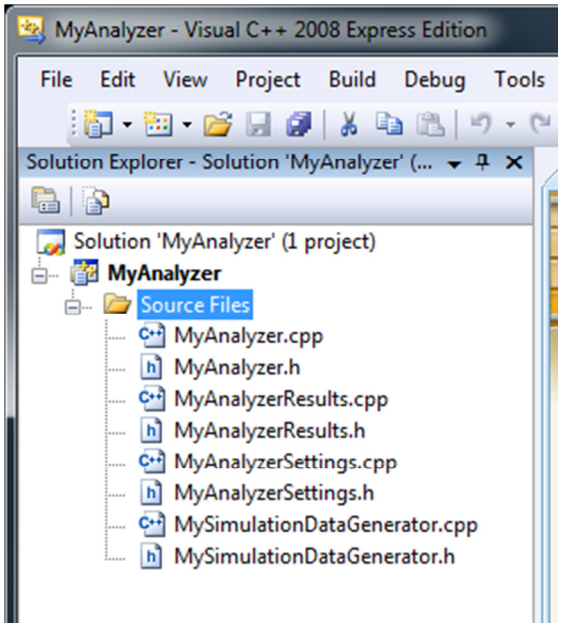
e. Close the console, and open the source folder inside your project folder. Notice that the file names now match your project. Internal changes have also been made – class names, etc.



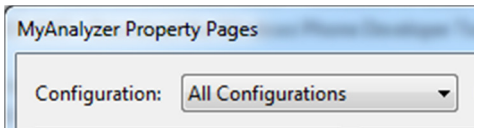
5. Next we'll add our source files to Visual Studio
   a. Under Solution Explorer, delete the folders Header Files and Resource Files. We won't be using them.



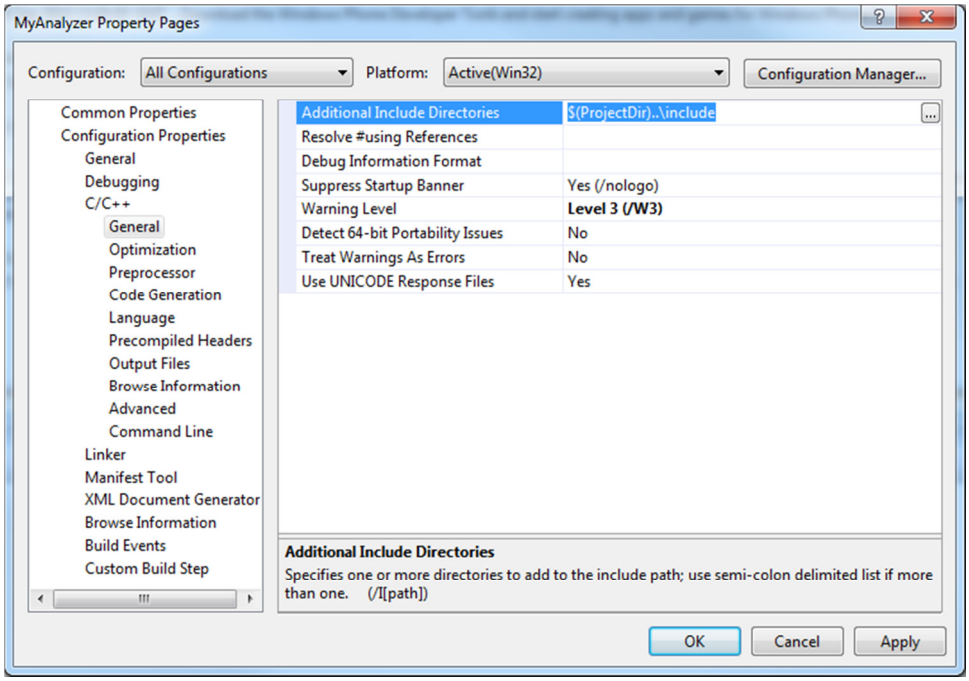
   b. Right click on Source Files and choose Add->Existing Item.
   c. Navigate to your source file, and select everything, including the headers



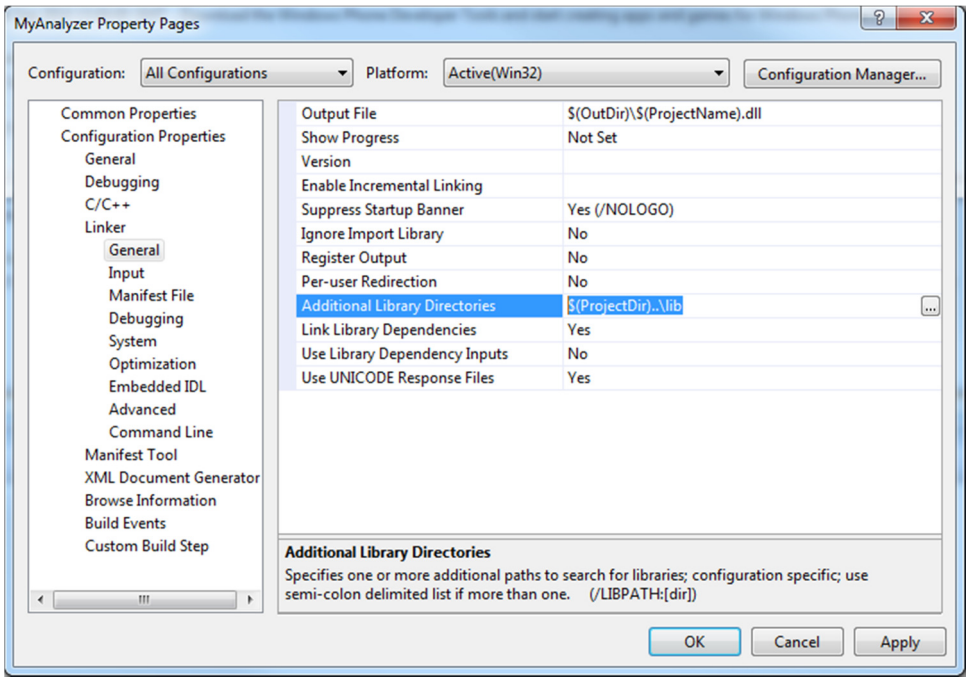
6. Now we must tell Visual Studio where to look for the SDK's header files.
    a. Right click on the project name, and select Properties.
    b. In the Configuration: drop-down, choose All Configurations (This saves us from needing to set it up for Release and Debug modes separately.)



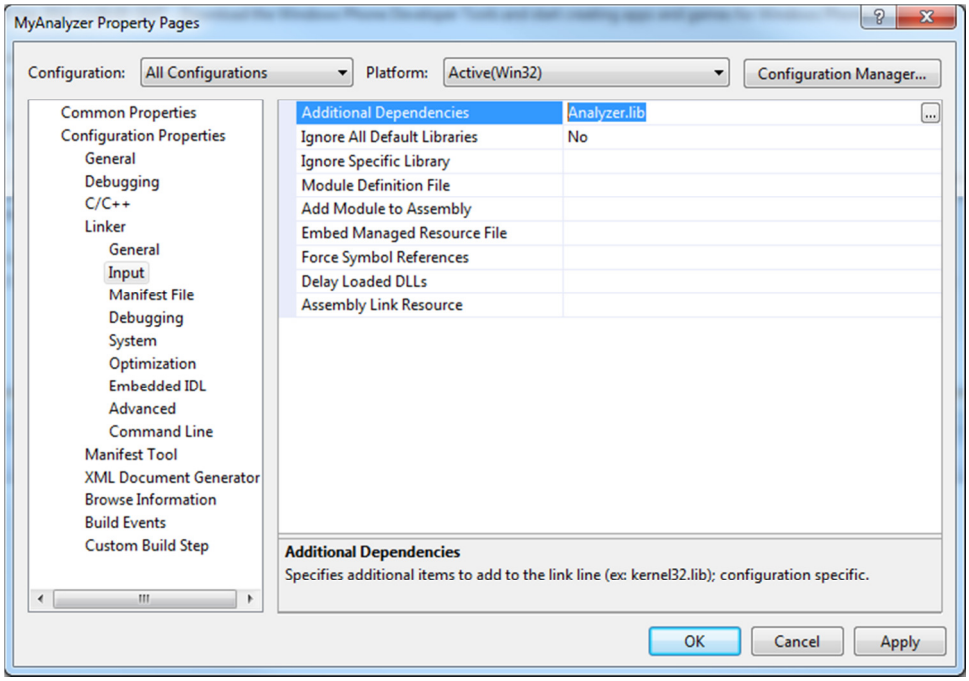
    c. Under C/C++, select the General Item.
    d. In Additional Include Directories, type **$(ProjectDir)..\include**



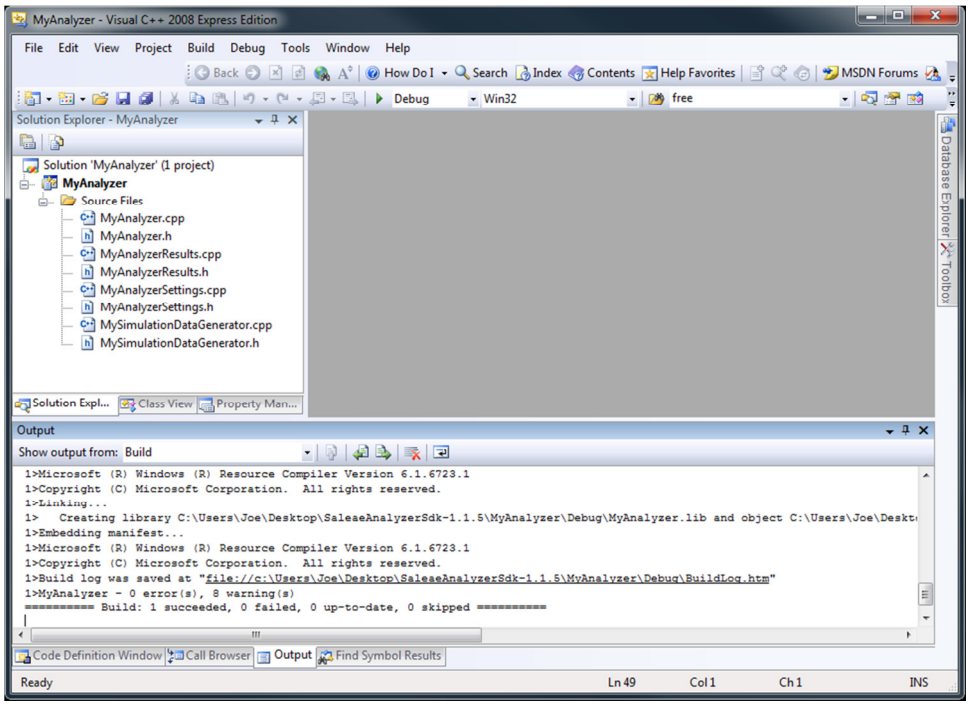
7. Next, let's point Visual Studio to the *Analyzer.dll* file we'll be linking against.
    a. Expand the Linker item and select General.
    b. Under Additional Library Directories enter **$(ProjectDir)..\lib**



    c. Under the Linker item, select Input.
    d. Under Additional Dependencies enter **Analyzer.lib**



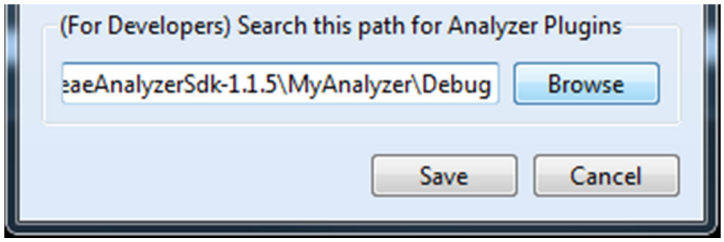
Ok! At this point everything is set up and you should be able to build and develop your code. You should be able to build for release, or debug. Use debug for development (see further down for how to debug), but always use release for real world use as it will run much faster.



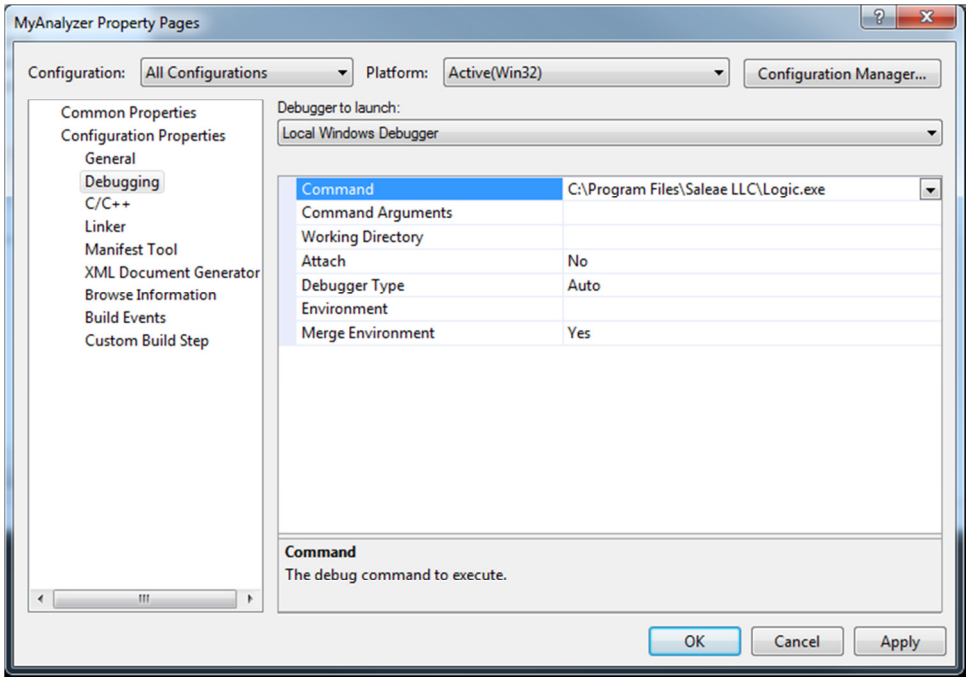
Debugging an Analyzer with Visual Studio

Now let's debug your project. You'll need to have the Logic software installed, and typically the same or higher version as the SDK you're using. We'll try to maintain binary compatibility in new releases to the extent possible, but there may be occasional breaking changes.

1. Launch the Logic software
    a. Options->Preferences
    b. Under Search this path for Analyzer Plugins browse to the *Debug* (or Release) folder in your project.

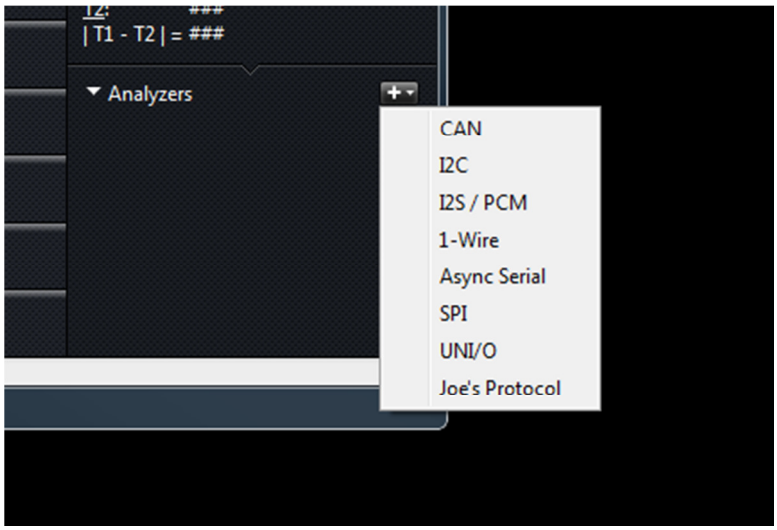


    c. Click Save and then close the Logic software.
2. Next, we need to tell Visual Studio to use the Logic software in its debug session.
    a. In Visual Studio, right click in the Project item (under Solution Explorer) and select Properties.
    b. Click the Debugging Item
    c. Under Command click in the field, click the down arrow that appears, and select Browse…
    d. Navigate to the *Logic.exe* program. This is typically located at *C:\Program Files\Saleae LLC* (Note that the older 1.0.33 version, if it is installed, is located in the folder *C:\Program Files\Logic* – be sure not to accidently use this version).



e. Press OK.

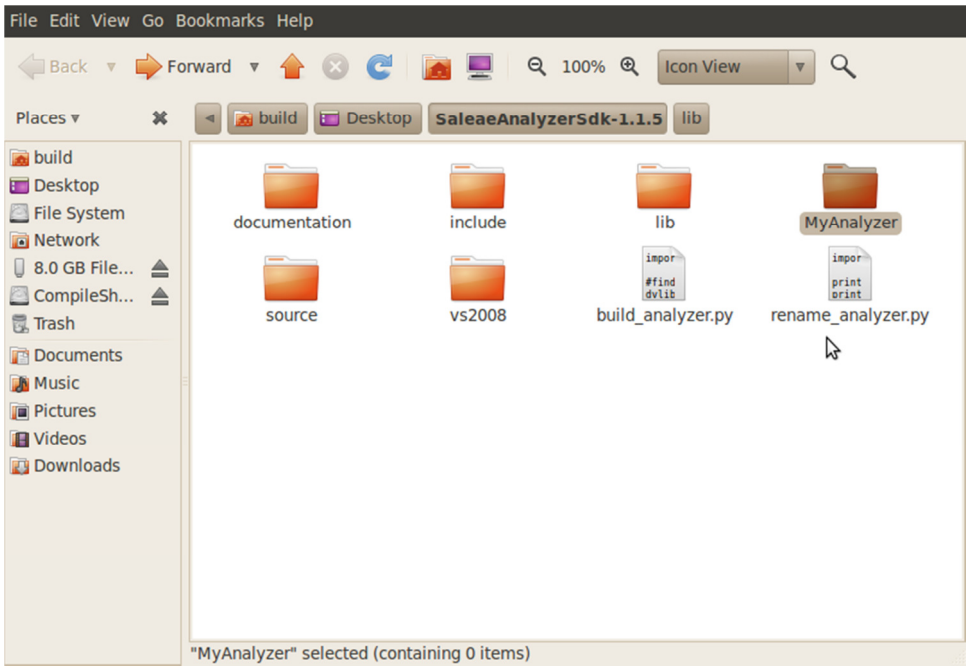
3. Debug the application (F5). This will launch the Logic software. If all has gone well, you should be able to see your analyzer in the analyzers drop down list.



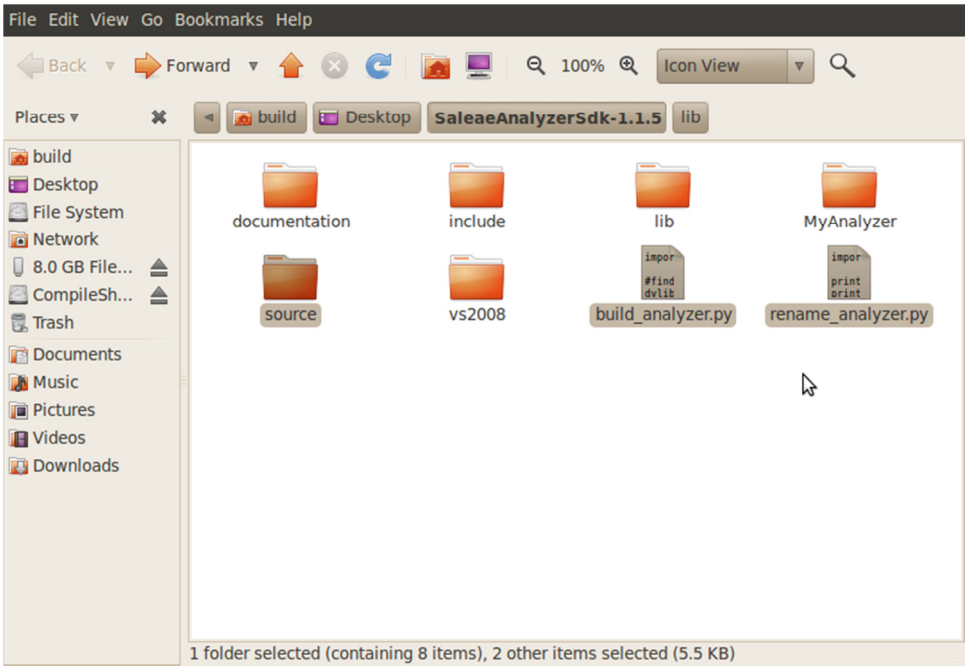
To make sure debugging is working, put a breakpoint in your Analyzer's constructor. Then, from the Logic software, select your analyzer from the list. Your breakpoint should fire -- you're off and running!

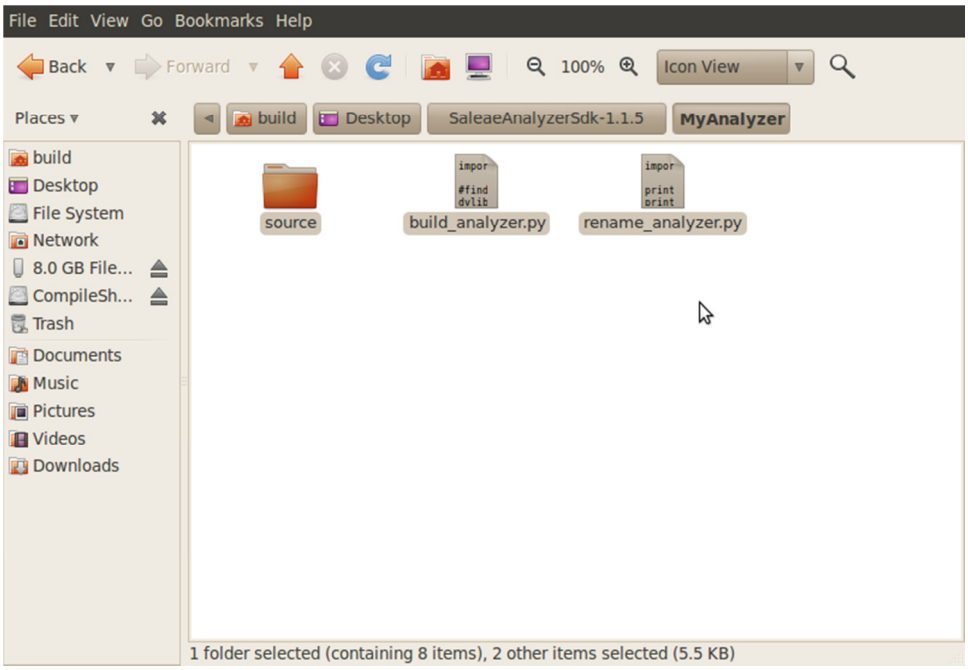
Linux

1. If you haven't already, download and extract the *SaleaeAnalyzerSdk-1.1.x* to your desktop, or other convenient location.
2. Inside the *SaleaeAnalyzerSdk-1.1.x* folder, create a new folder for your new analyzer project. Name it something like MyAnalyzer, SuperSerialAnalyzer, etc.

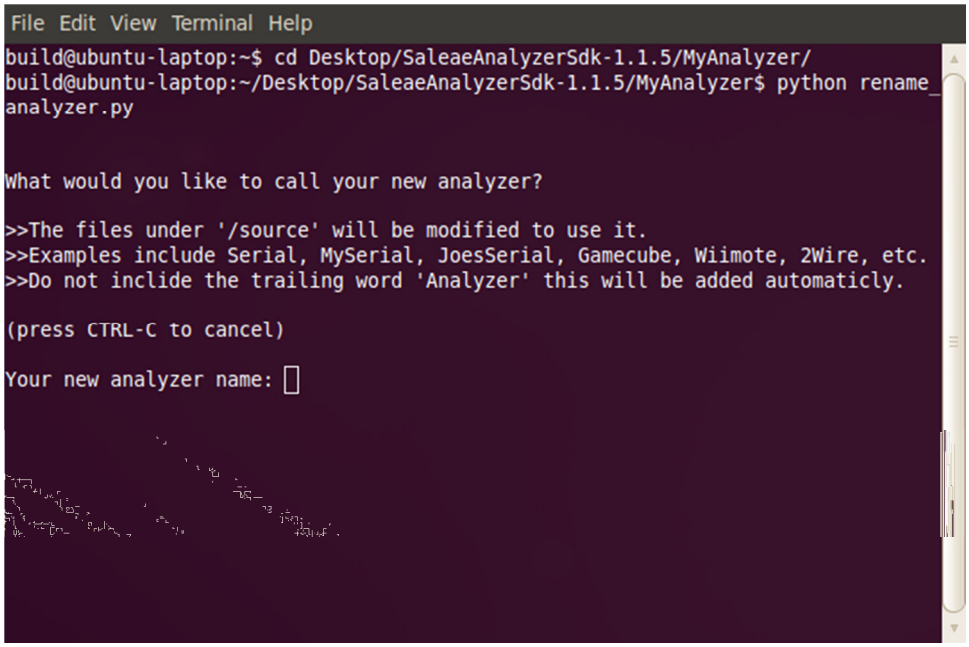


3. From the root of *SaleaeAnalyzerSdk-1.1.x* folder, copy *rename_analyzer.py*, *build_analyzer.py*, and the *source* folder. Paste these inside your new project folder.



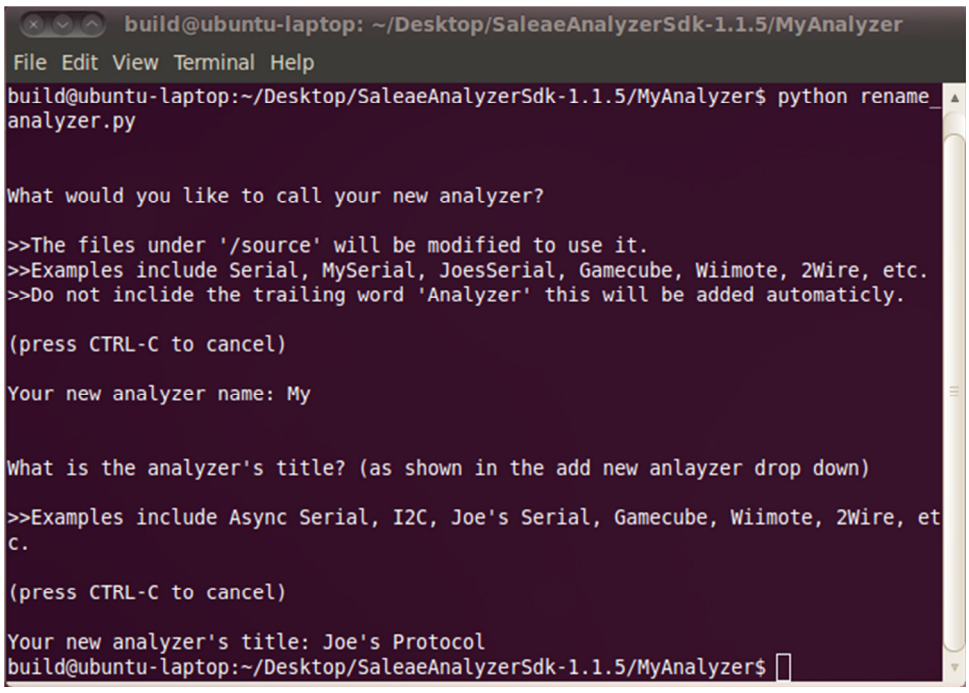


4. Next, we'll modify the source files so their names -- and everything in them -- reflect your new analyzer name. Otherwise it's a bit of a pain to modify by hand.
    a. Open a terminal (console) and navigate to your *MyAnalyzer* folder. (typically **cd Desktop\SaleaeAnalyzerSdk-1.1.x\MyAnalyzer**)
    b. Type **python rename_analyzer.py**



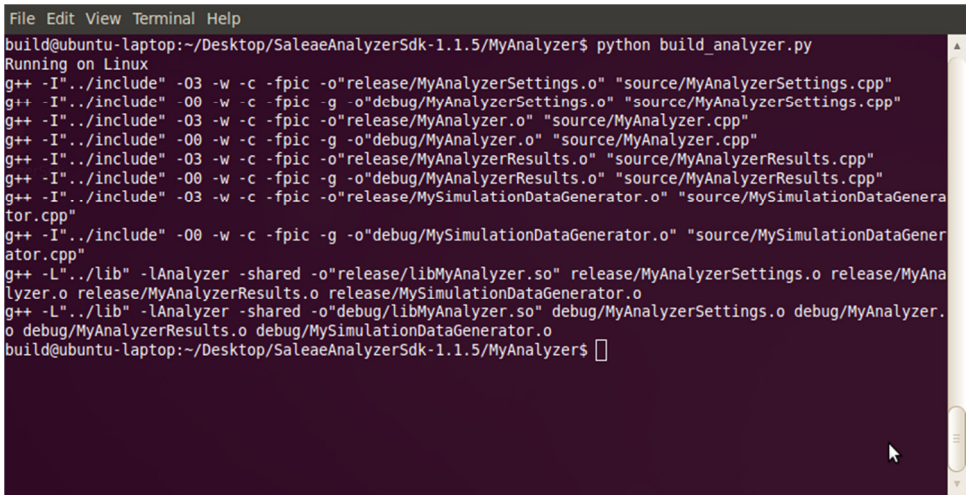
c. You will be prompted for the name of your analyzer – be sure to enter this without the Analyzer suffix; this will be added automatically. For example, for *MyAwesomeAnalyzer*, enter **MyAwesome**. Also note that this needs to be a valid variable name, so it should be one word.

d. Next you will be prompted for the menu title to display in the Add Analyzer drop down menu in the Logic software. This is really easy to change later, so don't worry about it too much. For example, you could enter **Joe's Analyzer**



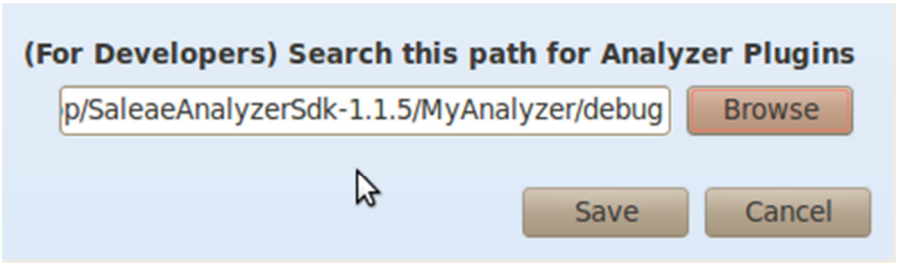
e. The script should have renamed your project files in your source folder, as well as changed their contents to match your analyzer's name.

5. Lastly, build the analyzer with the build_analyzer build script. At the command line, type **python build_analyzer.py**

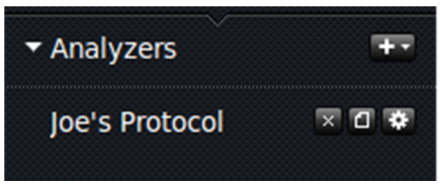


Next, let's setup Logic to find your new analyzer

1. Launch the Logic software.
2. Select Options->Preferences
3. Under Developer, click Browse. Navigate new analyzer project's debug (or release) folder, at *SaleaeAnalyzerSdk-1.1.x/ MyAnalyzer/debug*

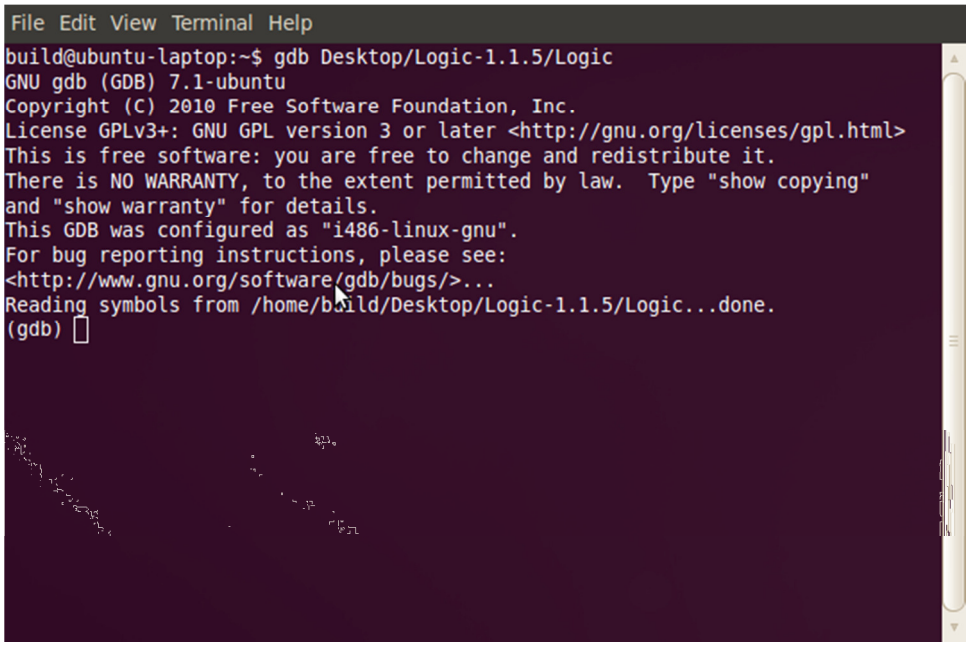


4. Press Save, and then close the Logic software.
5. Launch the Logic software again. Your analyzer should show up the Add Analyzer drop-down list.

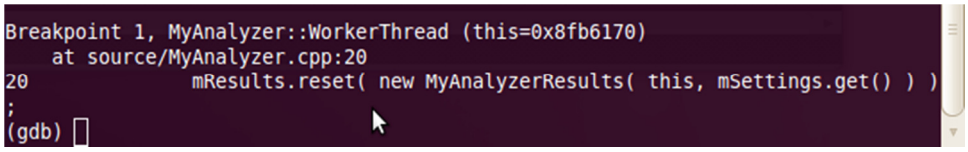


Lastly, let's debug our new analyzer with GDB

1. Open a terminal (console)
2. Run GDB on the Logic application: type something like **gdb /Home/YourUserName/Desktop/Logic-1.1.x/Logic**



3. Set a breakpoint to get fired when your analyzer is loaded: type **break MyAnalyzer::WorkerThread**
    a. Because your analyzer hasn't been loaded yet, GDB will notify you that it can't find this function, and ask if you want to automatically set this breakpoint if a library with a matching function is loaded in the future. Type **y** <enter>
4. Launch the application. Type **run**
5. Now select your analyzer from the Add Analyzer drop-down list, and start a collection. GDB should break execution upon entering your *WorkerThread* function.



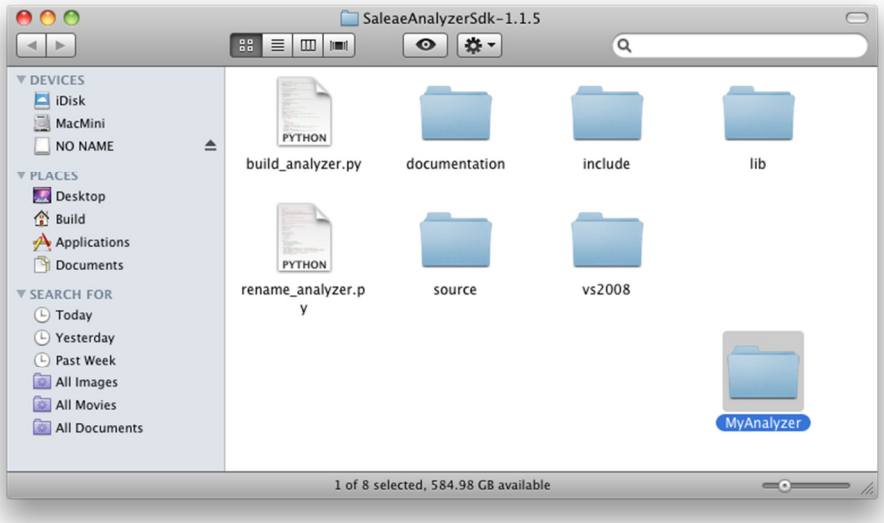
6. You can now type **step** to single step, and **continue** to resume running.

More in-depth use of GDB is outside the scope of this document.

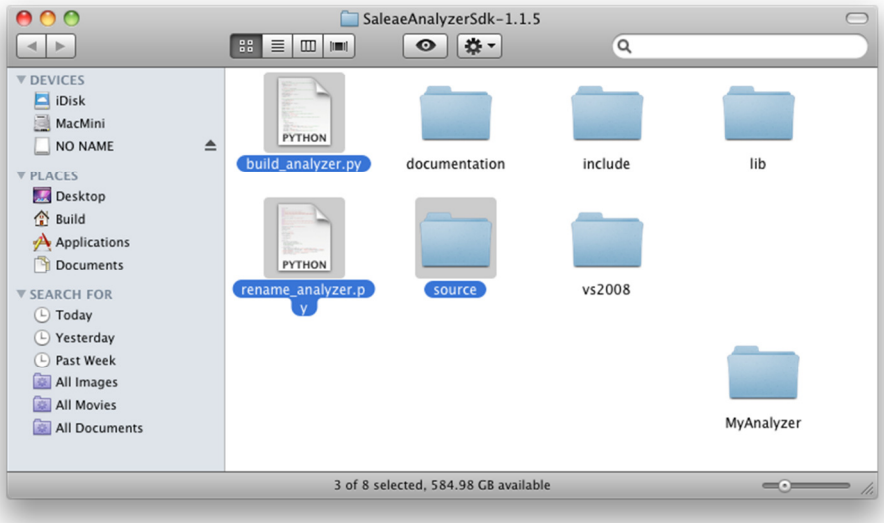
Mac OSX

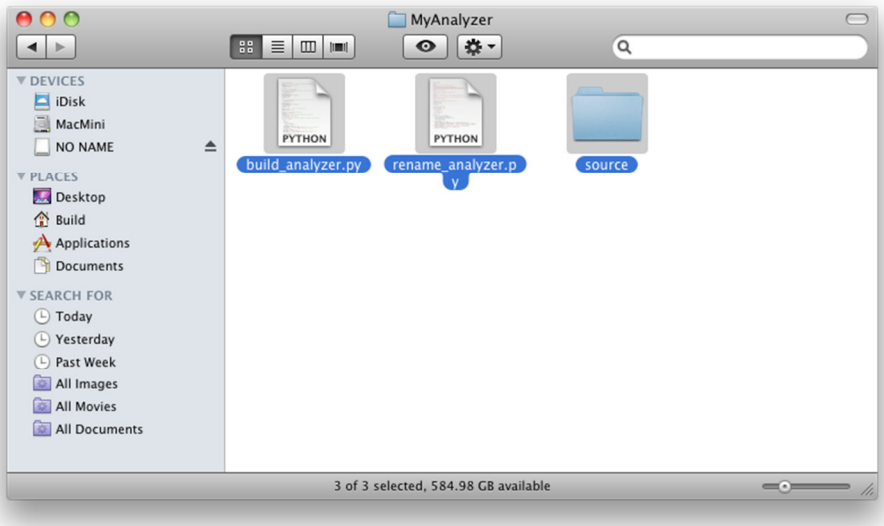
Build Script Based Project

1. If you haven't already, download and extract *SaleaeAnalyzerSdk-1.1.x* to your desktop, or to another convenient location.
2. Decide on the name of your project -- *MyAnalyzer*, *SuperSerialAnalyzer*, etc. Open the *SaleaeAnalyzerSdk-1.1.x* folder and create a new folder with this name. We'll call ours *MyAnalyzer*

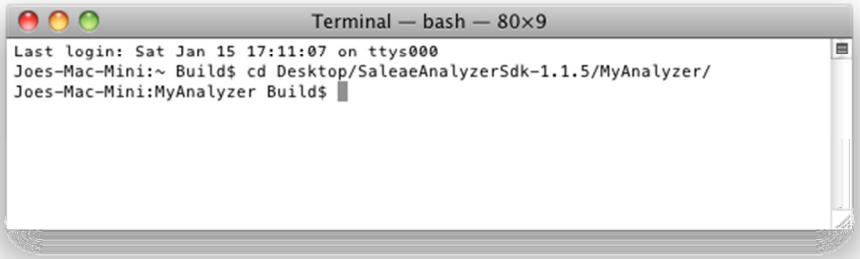


3. Select the *source* folder, and the files *build_analyzer.py* and *rename_analyzer.py.* Copy and paste these into your new analyzer folder.

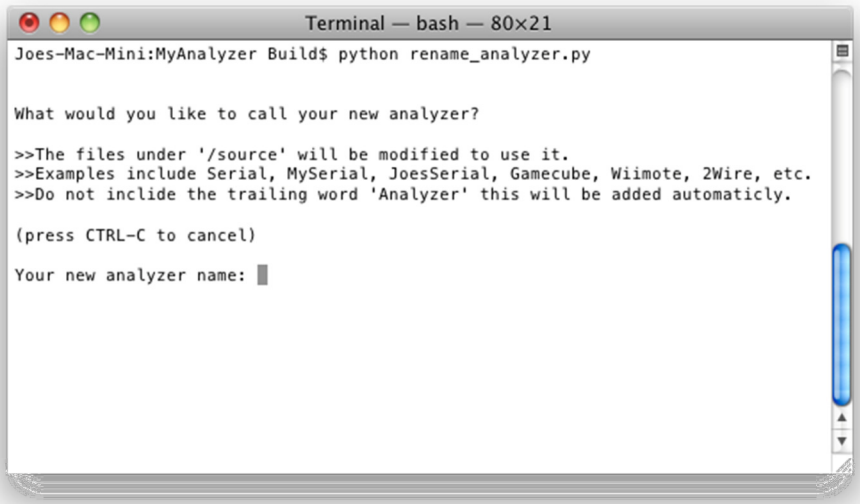




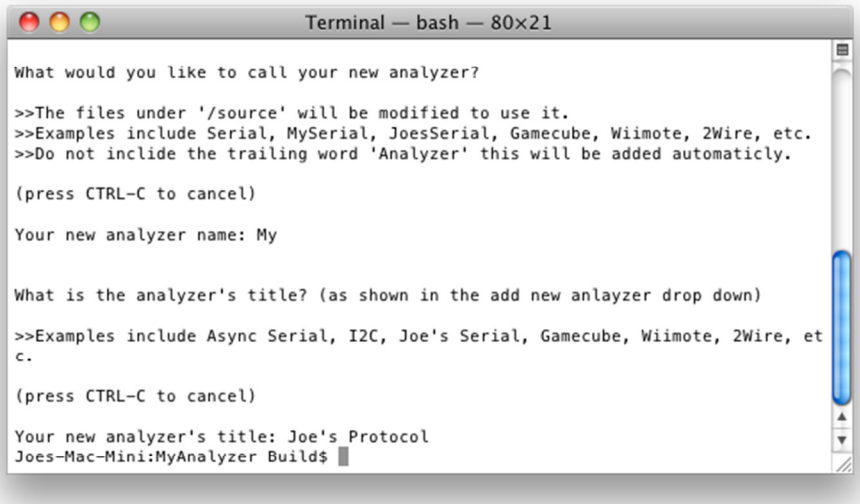
4. Open a terminal (under *Applications/Utilities*) and navigate to your new analyzer folder. Type something like **cd Desktop\SaleaeAnalyzerSdk-1.1.5\MyAnalyzer**



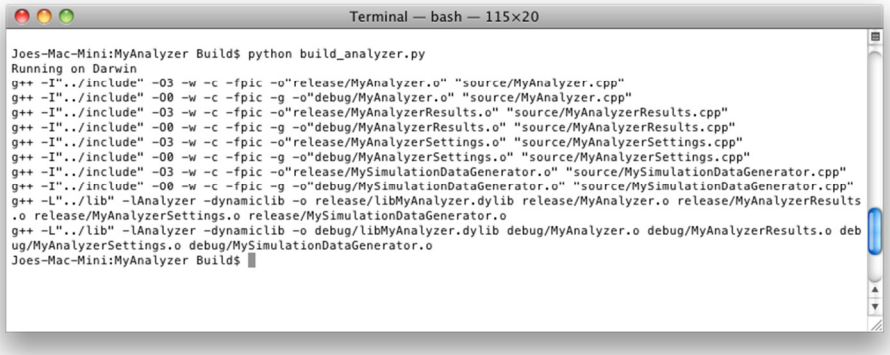
5. Next we'll rename and modify the source files so they will be all set up for your analyzer. We'll use the *build_analyzer.py* script to do this.
6. Type **python rename_analyzer.py**



7. Follow the prompts
    a. You will be prompted for the name of your analyzer – be sure to enter this without the Analyzer suffix; this will be added automatically. For example, for *MyAwesomeAnalyzer*, enter **MyAwesome**. Also note that this needs to be a valid variable name, so it should be one word.
    b. Next you will be prompted for the menu title to display in the Add Analyzer drop down menu in the Logic software. This is really easy to change later, so don't worry about it too much. For example, you could enter **Joe's Analyzer**
    c. The script should have renamed your project files in your source folder, as well as changed their contents to match your analyzer's name.

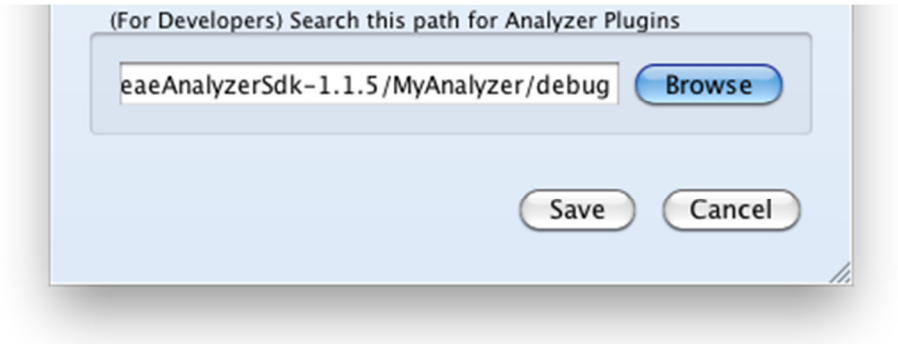


8. Next we'll attempt to build the project. Type **python build_analyzer.py**

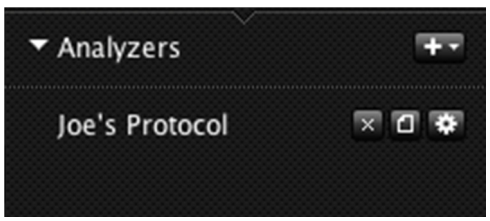


Your analyzer should now be built. Let's try to run it from the Logic software.

6. Launch the Logic software.
7. *Select Options->Preferences*
8. Under Developer, click Browse. Navigate new analyzer project's debug (or release) folder, at *SaleaeAnalyzerSdk-1.1.x/ MyAnalyzer/debug*

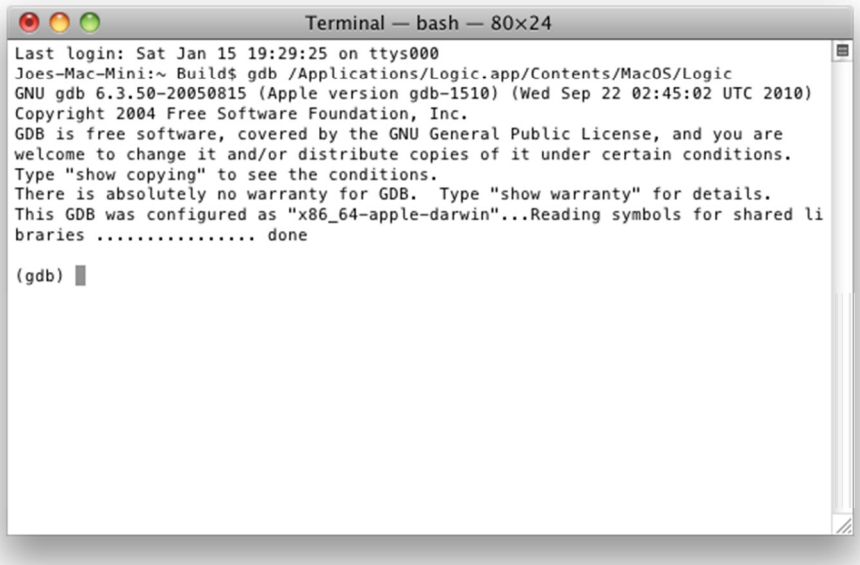


9. Press Save, and then close the Logic software.
10. Launch the Logic software again. Your analyzer should show up the Add Analyzer drop-down list.

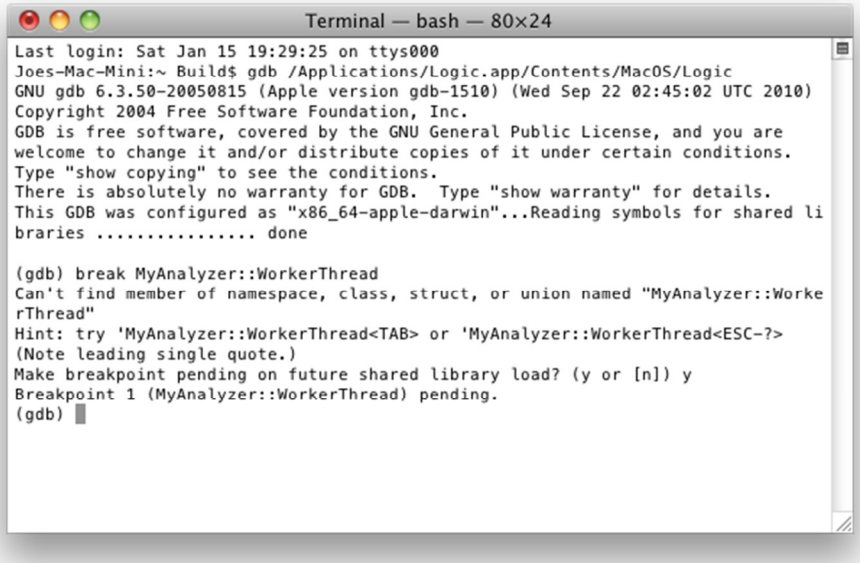


Debugging with GDB

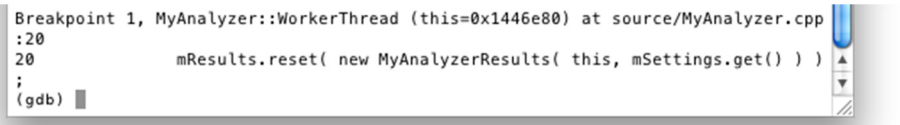
7. Open a terminal (under *Applications/Utilities*)
8. Run GDB on the Logic application: type **gdb /Applications/Logic.app/Contents/MacOS/Logic**



9. Set a breakpoint to get fired when your analyzer is loaded: type **break MyAnalyzer::WorkerThread**
    a. Because your analyzer hasn't been loaded yet, GDB will notify you that it can't find this function, and ask if you want to automatically set this breakpoint if a library with a matching function is loaded in the future. Type **y** <enter>



10. Launch the application. Type **run**
11. Now select your analyzer from the Add Analyzer drop-down list, and start a collection. GDB should break execution upon entering your *WorkerThread* function.



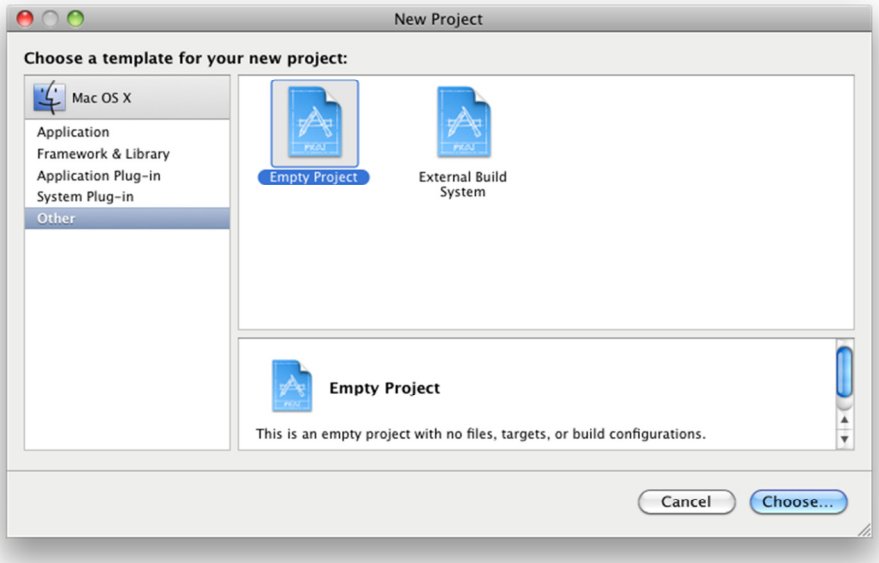
12. You can now type **step** to single step, and **continue** to resume running.

More in-depth use of GDB is outside the scope of this document. For a GUI based debugging solution, we recommend using XCode.

XCode based Project

Note that in this walkthrough we are using XCode on Snow Leopard. Your experience may be somewhat different if you are in Tiger or Leopard.

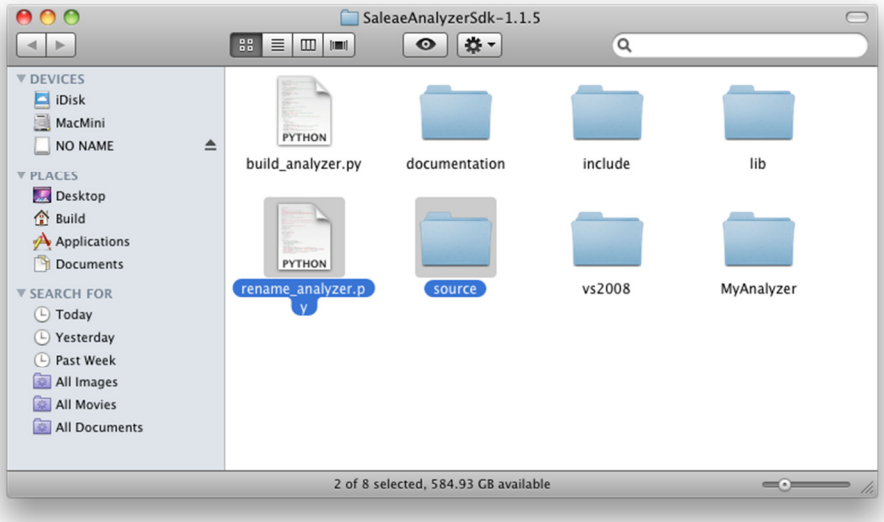
1. Start XCode
2. From the File menu, choose New Project
3. For Choose a template for your new Project select Other, Empty Project.

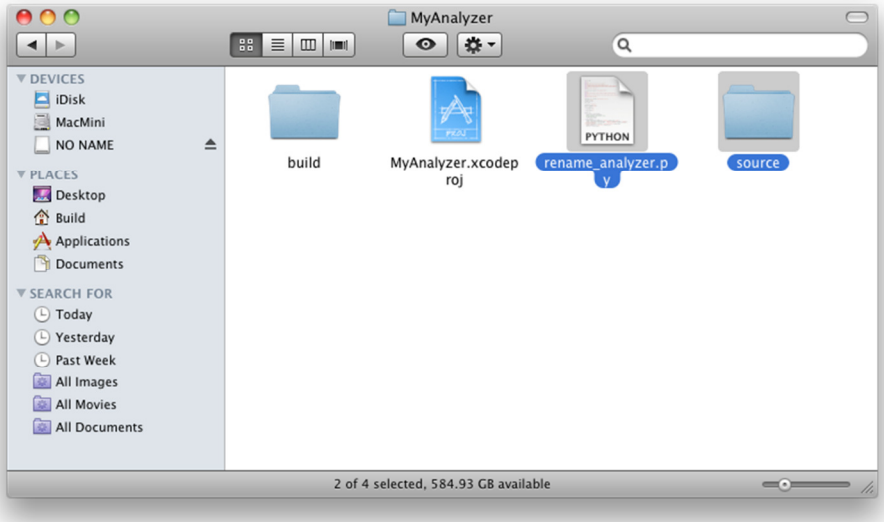


4. Choose a name for your analyzer, such as *MyAnalyzer*, *SuperSerialAnalyzer*, etc. It should be one word. Specify this as the project's name.
5. Save the project in the root of the *SaleaeAnalyzerSdk-1.1.x* folder. This will create a folder called *YourProjectName*.

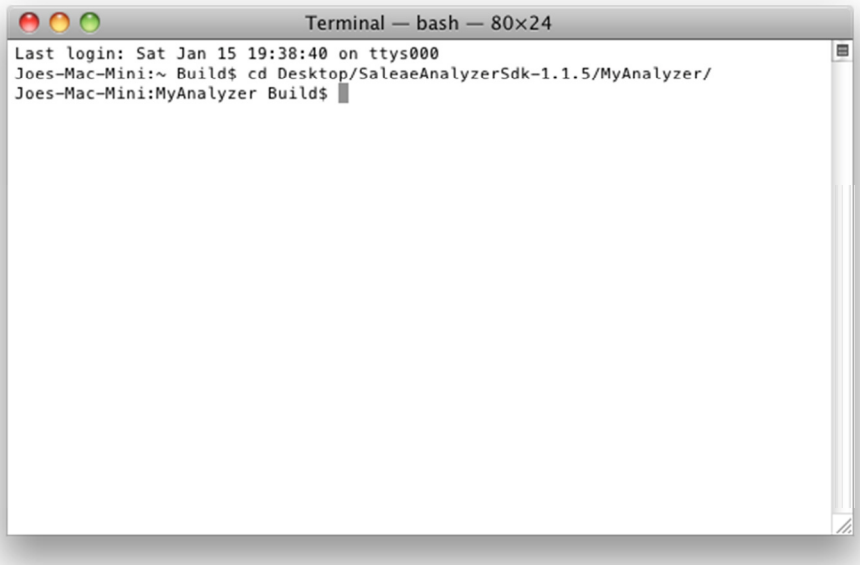
Close XCode. Next we'll get our source files in order.

1. With Finder, open the *SaleaeAnalyzerSdk-1.1.x* folder. Select the *source* folder, and the file *rename_analyzer.py*, and copy and paste these into your new analyzer project folder.

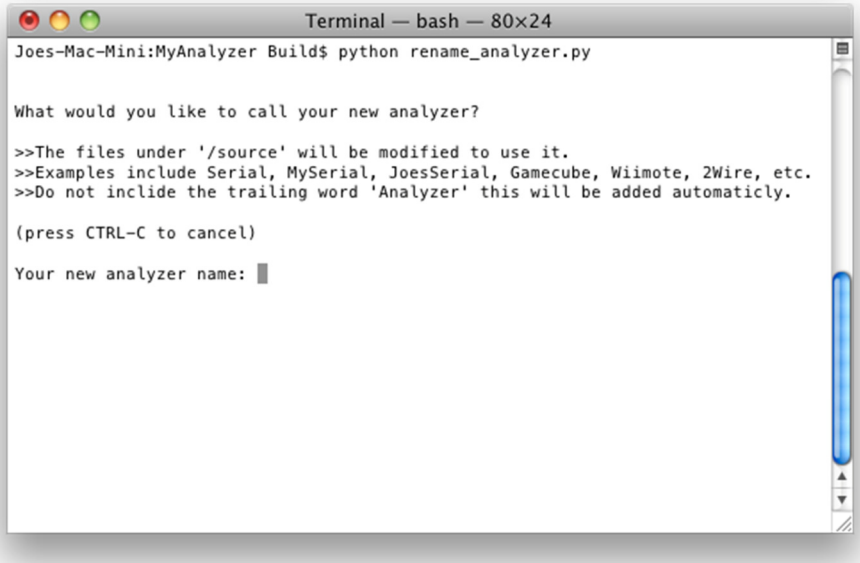




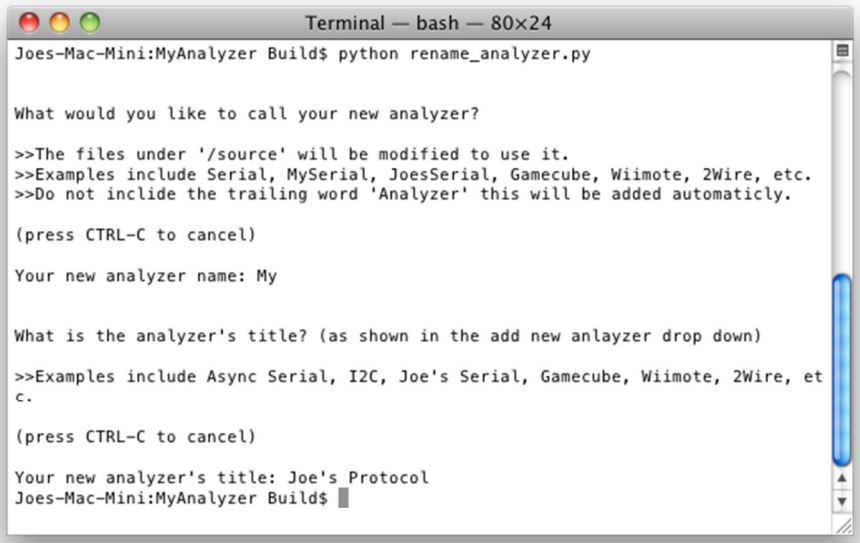
2. Launch Terminal. (This is in *Application/Utilities*)
3. Navigate to your analyzer folder by typing something like **cd Desktop/SaleaeAnalyzerSdk-1.1.x/MyAnalyzer**



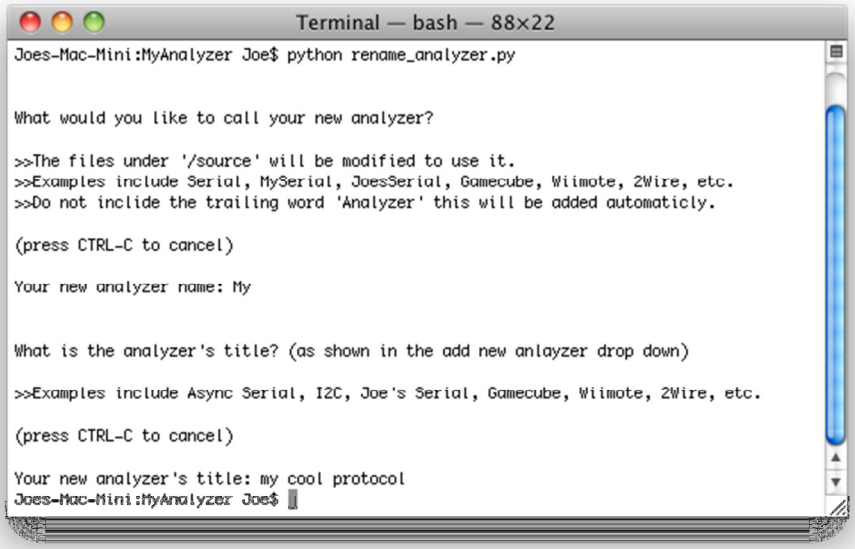
4. Type **python rename_analyzer.py** and follow the prompts.



a. You will be prompted for the name of your analyzer – be sure to enter this without the Analyzer suffix; this will be added automatically. For example, for *MyAwesomeAnalyzer*, enter **MyAwesome**. Also note that this needs to be a valid variable name, so it should be one word.
b. Next you will be prompted for the menu title to display in the Add Analyzer drop down menu in the Logic software. This is really easy to change later, so don't worry about it too much. For example, you could enter **Joe's Analyzer**

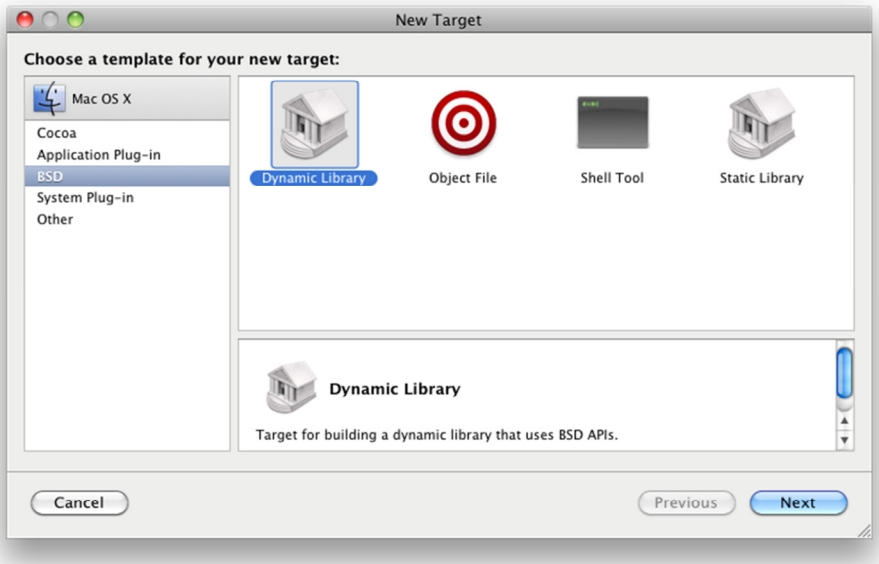


c. The script should have renamed your project files in your source folder, as well as changed their contents to match your analyzer's name.

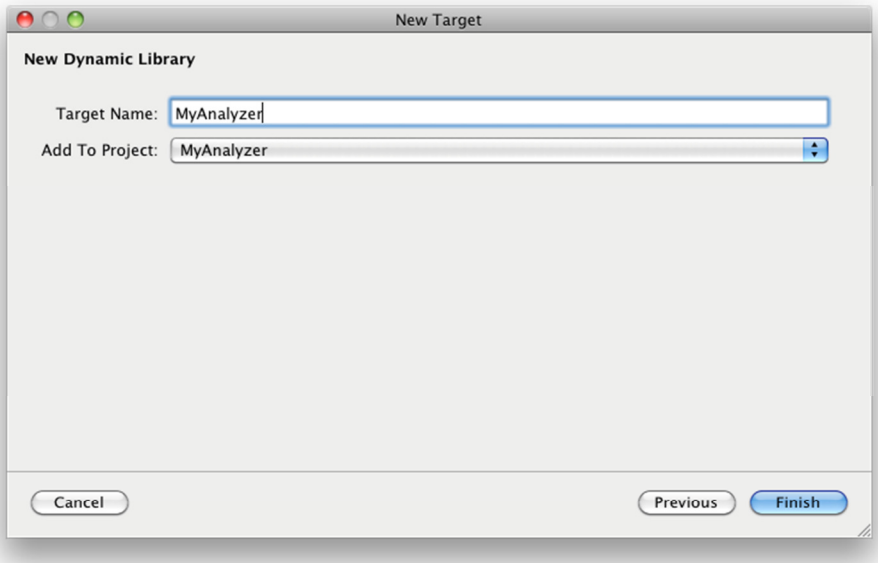


Back to XCode

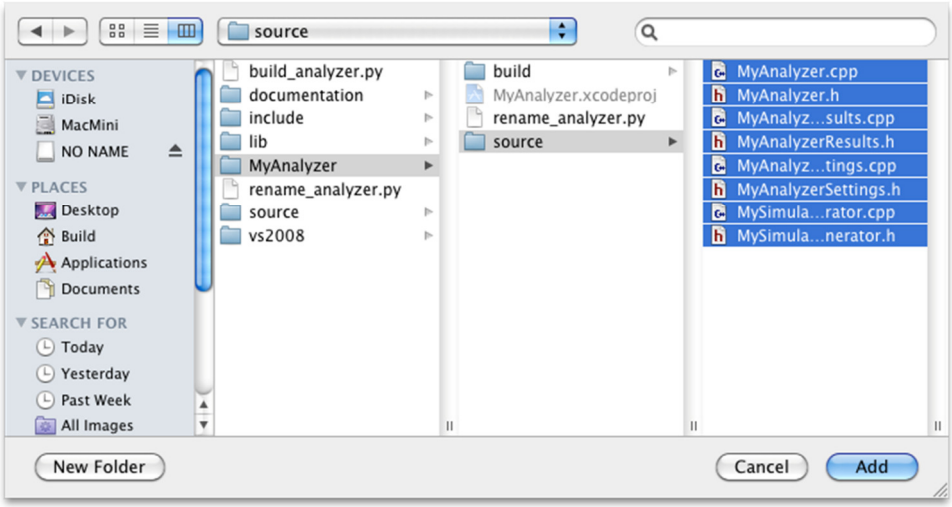
1. In your new project folder, double-click the XCode project. (starting XCode)
2. Under Groups & Files, right-click on Targets, and select Add->New Target
3. Select BCD, Dynamic Library template, and click Next.



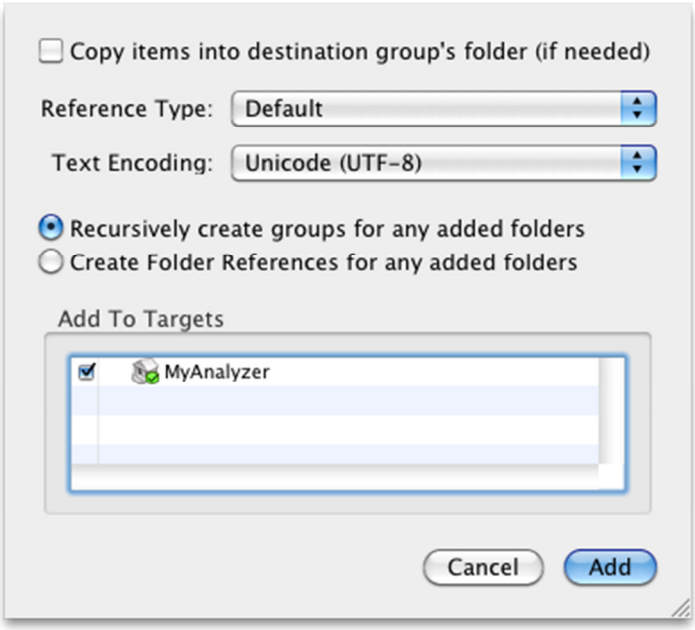
4. For Target Name, enter your analyzer's name (*MyAnalyzer* for example). Click Finish.



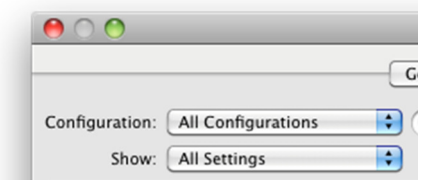
5. This will open the Target Info window. Close this.
6. In the Groups & Files list, select the project item at the very top of the list. Right click and select Add->Existing Files.
7. Navigate to your source folder and select all the files, including the headers. Click Add.



8. The defaults should be fine. Click Add.

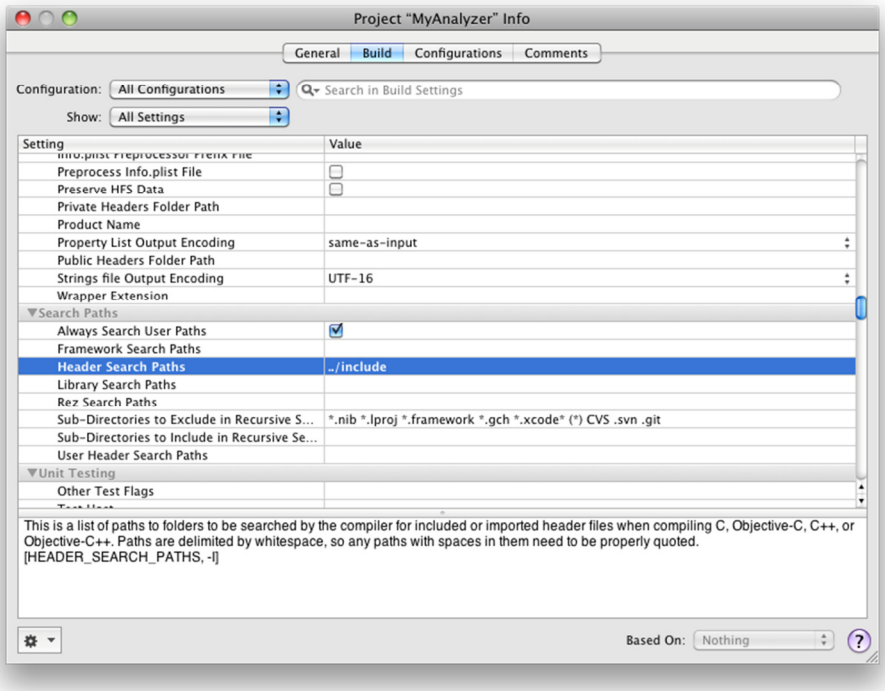


9. Select the project item (at the top of the list), and click the Info button on the main toolbar.
    a. Click the Build tab.
    b. Set Configuration to All Configurations, and set Show to All Settings



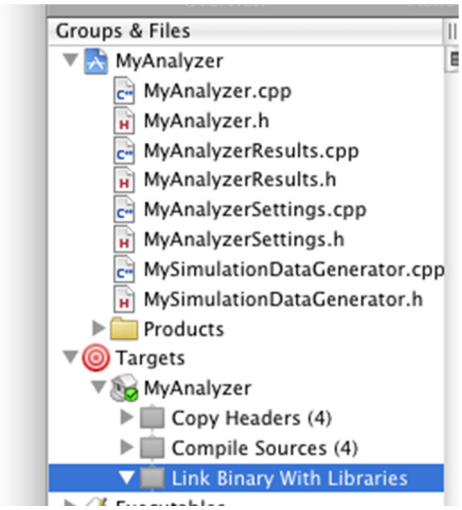
c. Scroll down to Search Paths section

d. For Header Search Paths, enter **../include**



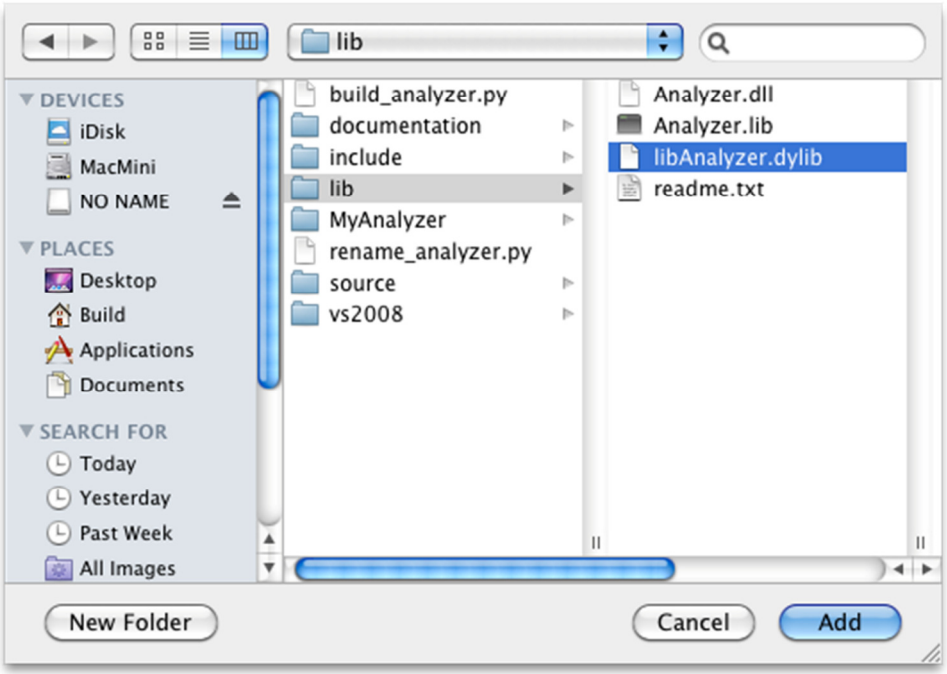
e. Close the Project Info window.

10. Expand the Targets item until you see the Link Binary With Libraries item.

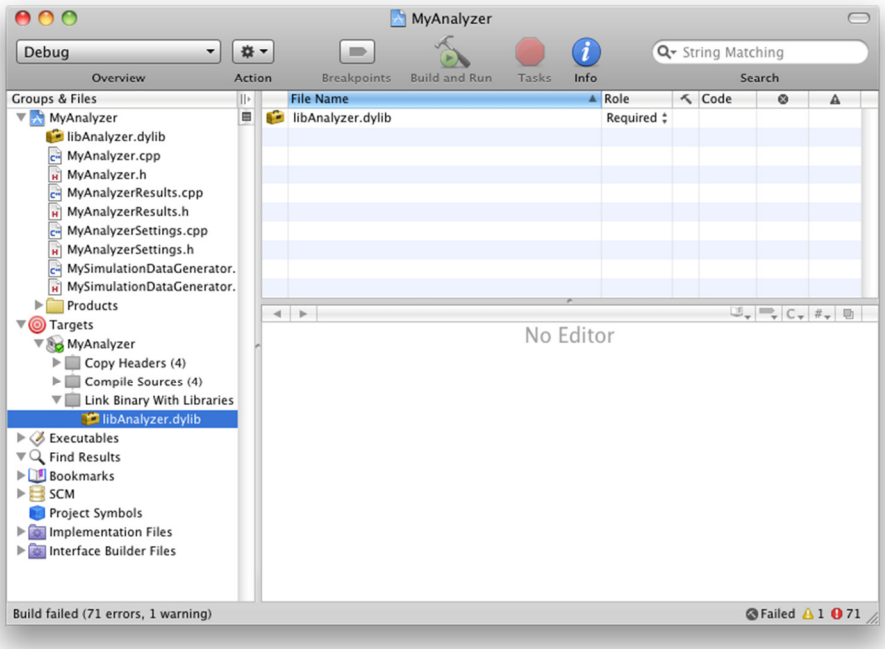


a. Right click on Link Binary With Libraries and select Add->Existing Files.

b. Navigate to and select *libAnalyzer.dylib* from *lib* folder in the *SaleaeAnalyzerSdk-1.1.x* folder.



c. Click Add, and except the defaults by click Add again.

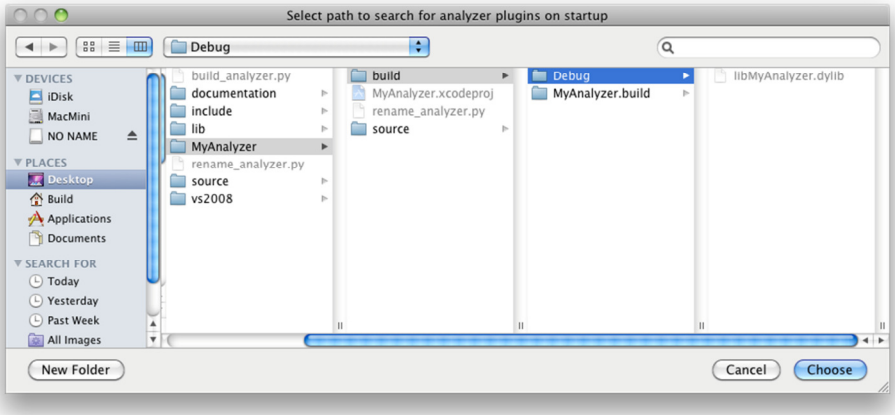


From the Build menu (at the top of the screen), select Build. Your project should build completely.

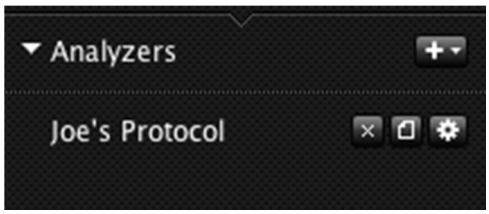
Running & Debugging your Analyzer

1. Launch the Logic software.

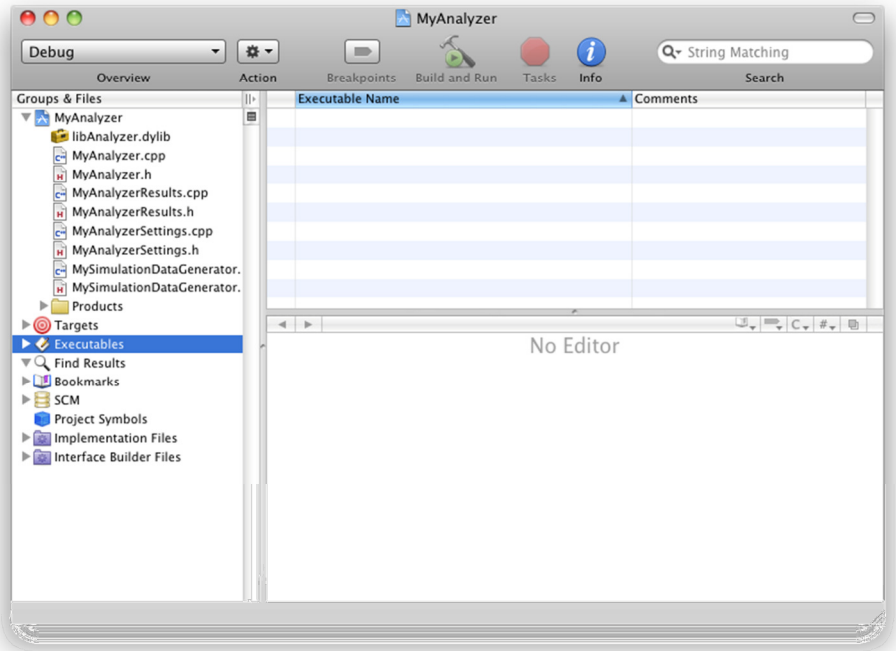
a. Select Options->Preferences
b. Under Developer, click Browse. Navigate new analyzer project's debug (or release) folder, at *SaleaeAnalyzerSdk-1.1.x/ MyAnalyzer/build/debug*



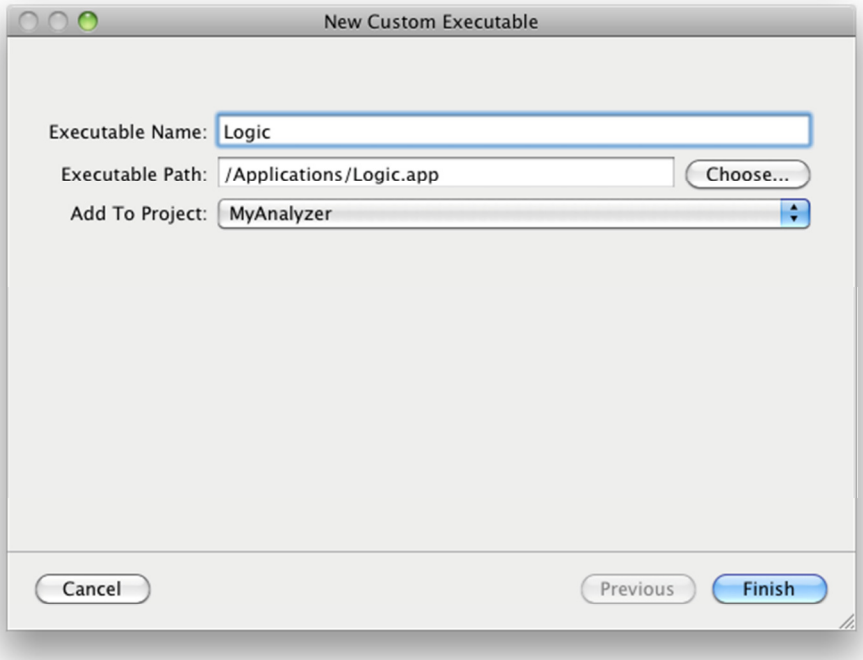
c. Press Save, and then close the Logic software.
d. Launch the Logic software again. Your analyzer should show up the Add Analyzer drop-down list.



2. Bring up your analyzer project in XCode
   a. Right click on the Executables item, and select Add->New Custom Executable.



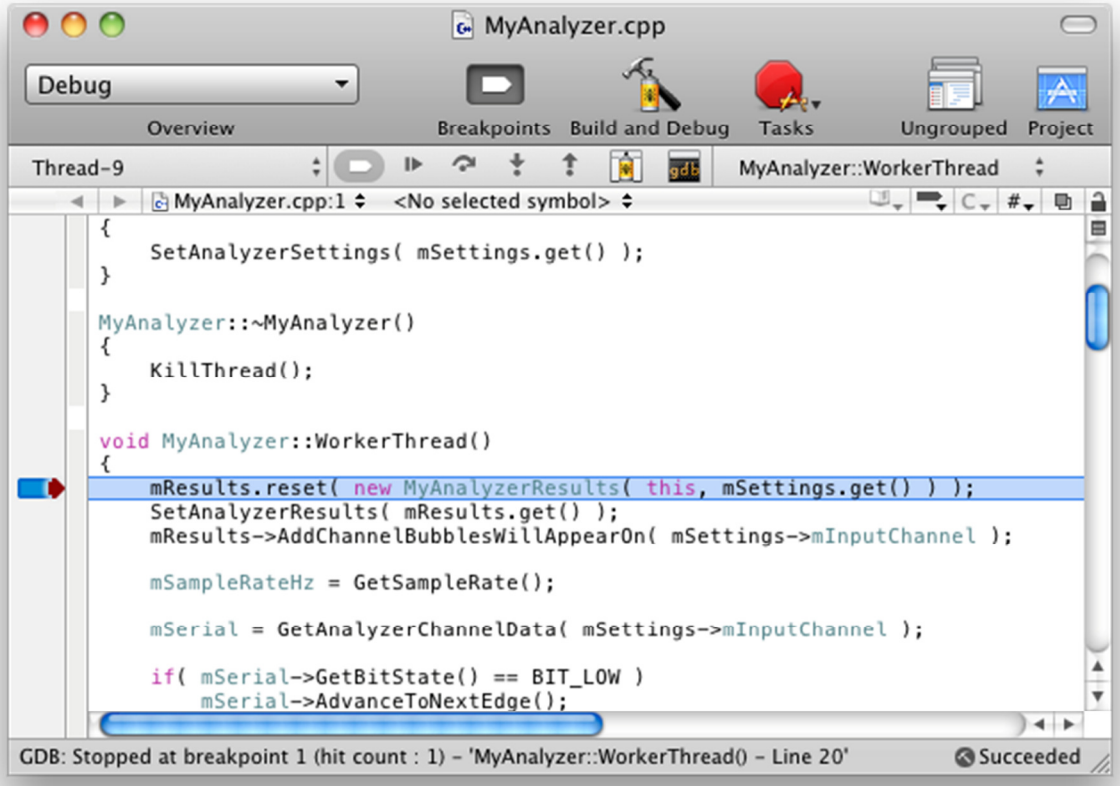
i. Executable Name: **Logic**
ii. Executable Path: **/Applications/Logic.app** (or similar)
iii. Click Finish.



b. From XCode, open the primary cpp file (*MyAnalyzer.cpp* or similar).

c. Go down to the first line on *WorkerThread* and click in the margin to create a breakpoint.
d. From the Build menu, select Build and Debug.
e. Select your new analyzer from the Add Analyzer drop-down.

Start a data collection. XCode should break execution at your breakpoint.



Writing your Analyzer's Code

This second part of the document deals with writing the code for your analyzer.

There are 4 c++ files and 4 header files that you will implement to create your analyzer. If you followed the procedure in the first part, you already have a working analyzer, and will be modifying that code to suit your needs.

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 Logic software 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 this 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!

Analyzer Settings

After setting up your analyzer project, and renaming the source files to match your project, the first step is to implement/modify your analyzer's *AnalyzerSettings*-derived class.

{YourName}AnalyzerSettings.h

In this file, you provide a declaration for your *{YourName}AnalyzerSettings* class. This class must inherit from *AnalyzerSettings*, and should include the *AnalyzerSettings.h* header file.

We'll start with this

```
#ifndef SIMPLESERIAL_ANALYZER_SETTINGS
#define SIMPLESERIAL_ANALYZER_SETTINGS

#include <AnalyzerSettings.h>
#include <AnalyzerTypes.h>

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

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

#endif //SIMPLESERIAL_ANALYZER_SETTINGS
```

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

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

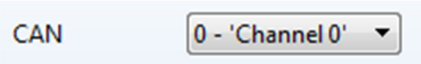
- Enable line polarity
- Etc – anything you need for your specific protocol. If you like, start with just the *Channel* variable(s), and you can add more later to make your analyzer more flexible.

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.

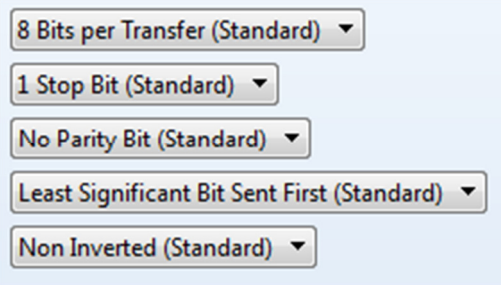
AnalyzerSettingsInterfaces

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:

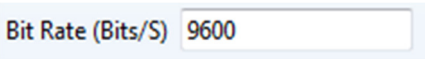
- AnalyzerSettingInterfaceChannelData: Used exclusivly for input channel selection.



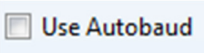
- AnalyzerSettingInterfaceNumberList: Used to provide a list of numerical options for the user to choose from. Note that this can be used to select from several enum types as well, as illustrated below. (Each dropdown below is implemented with its own interface object)



- 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_ptr* type, which largly acts like a standard (raw) pointer. It's a simple form of what's called a "smart pointer" and it automaticly calls *delete* on its contents when it goes out of scope.)

{YourName}AnalyzerSettings.cpp

The Constructor

In your constructor, we'll first initilize all your settings variables to their default vaules. Second, we'll setup each variable's corresponding interface object.

Note that if the user has previously entered values for this analyzer, these will be loaded at a later time. Be sure to initialize the variables to values that you want to be defaults. These will show up when a user adds a new instance of your analyzer.

Setting up each AnalyzerSettingInterface object

First, we create the object (call new) and assign the value to the interface's pointer. Note that we're using *std::auto_ptr*, so this means calling the member function *reset*. For standard (raw pointers), you would do something like:

```
mInputChannelInterface = new AnalyzerSettingInterfaceChannel();
```

Next, we call the member function *SetTitleAndTooltip*. 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 );
mInputChannelInterface->SetTitleAndTooltip( "Serial", "Standard Async Serial" );
```

We'll want to set the value. The interface object is, well, an interface to our settings variables. When setting up the interface, we copy the value from our settings variable to the interface. When the user makes a change, we copy the value in the interface to our settings variable. The function names for this differ depending on the type of interface.

```
void SetChannel( const Channel& channel );
void SetNumber( double number );
void SetInteger( int integer );
void SetText( const char* text );
void SetValue( bool value );
```

We'll want to specify the allowable options. This depends on the type of interface.

AnalyzerSettingInterfaceChannel

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

Some channels can be optional , but typically they are not. By default, the user must select a channel.

AnalyzerSettingInterfaceNumberList

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

Call AddNumber for every item you want in the dropdown. *number* is the value assiciated with the selection; it is not displayed to the user.

AnalyzerSettingInterfaceInteger

```
void SetMax( int max );
void SetMin( int min );
```

You can set the allowable range for the integer the user can enter.

AnalyzerSettingInterfaceText

```
void SetTextType( TextType text_type );
```

By default, this interface just provides a simple textbox for the user to enter text, but you can also specify that the text should be a path, which will cause a browse button to appear. The options are *NormalText*, *FilePath*, or *FolderPath*.

AnalyzerSettingInterfaceBool

There are only two allowable options for the bool interface (checkbox).

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 );
```

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 );
AddExportOption( 0, "Export as text/csv file" );
AddExportExtension( 0, "text", "txt" );
AddExportExtension( 0, "csv", "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, "Serial", 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 thatn *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.

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.

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 *SimpleSerialAnalyzerSettings*.

```
bool SimpleSerialAnalyzerSettings::SetSettingsFromInterfaces()
{
        mInputChannel = mInputChannelInterface->GetChannel();
        mBitRate = mBitRateInterface->GetInteger();

        ClearChannels();
        AddChannel( mInputChannel, "Simple Serial", true );
```

```
        return true;
}
```

void {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 *SimpleSerialAnalyzerSettings*.

```
void SimpleSerialAnalyzerSettings::UpdateInterfacesFromSettings()
{
        mInputChannelInterface->SetChannel( mInputChannel );
        mBitRateInterface->SetInteger( mBitRate );
}
```

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*.

```
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. This operator returns *bool* – it will return *false* if the requested type is not exactly in the right place in the archive. This could happen if you change the settings variables over time, and a user tries to load an old settings string. If loading fails, you can simply not update that settings variable (it will retain its default value).

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 *SimpleSerialAnalzerSettings*.

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

        text_archive >> mInputChannel;
        text_archive >> mBitRate;

        ClearChannels();
        AddChannel( mInputChannel, "Simple Serial", true );

        UpdateInterfacesFromSettings();
}
```

const char {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 *SimpleSerialAnalyzerSettings*:

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

	text_archive << mInputChannel;
	text_archive << mBitRate;

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


SimulationDataGenerator

The next step after creating your *{YourName}AnalyzerSettings* files, is to create your *SimulationDataGenerator*.

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.

{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 *SimpleSerialSimulationDataGenerator.h*

```
#ifndef SIMPLESERIAL_SIMULATION_DATA_GENERATOR
#define SIMPLESERIAL_SIMULATION_DATA_GENERATOR

#include <SimulationChannelDescriptor.h>
class SimpleSerialAnalyzerSettings;
class SimpleSerialSimulationDataGenerator
{
public:
	SimpleSerialSimulationDataGenerator();
	~SimpleSerialSimulationDataGenerator();

	void Initialize( U32 simulation_sample_rate, SimpleSerialAnalyzerSettings*
settings );
	U32 GenerateSimulationData( U64 newest_sample_requested, U32 sample_rate,
SimulationChannelDescriptor** simulation_channel );

protected:
	SimpleSerialAnalyzerSettings* mSettings;
	U32 mSimulationSampleRateHz;
	SimulationChannelDescriptor mSerialSimulationData;
};
#endif //SIMPLESERIAL_SIMULATION_DATA_GENERATOR
```

Overview

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.

We'll go over this in detail in a minute.

{YourName}SimulationDataGenerator.cpp

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.

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.
    - We repeatedly do this

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

A common issue with converting exact timing values into numbers-of-samples, is that you lose some precision. This isn't always a problem, but it's nice to have a way to keep track of how much error is building up, and then, just at the right times, add an extra sample in so that on average, the timing is exact.

ClockGenerator is a class provided in *AnalyzerHelpers.h* which will let you enter time values, rather than numbers-of-samples. For example, instead of figuring out how many samples are in 500ns, you can just use *ClockGenerator* to both figure it out and manage the error, so that on average, your timing is perfect.

Initially we created *ClockGenerator* to create clock-like signals, but now you can use and time value, any time. Here's how:

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 exaple, 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.

Note that the number of samples for a specific time period may change slightly every once in a while. This is so that on average, timing will be exact.

You may want to have a *ClockGenerator* as a member of you class. This makes it easy to use from any helper functions you might create.

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

Let's take another look at the *Initialize* function, now that we have an idea what's going on. This example is from *SimpleSerialSimulationDataGenerator.cpp*.

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

	mSerialSimulationData.SetChannel( mSettings->mInputChannel );
	mSerialSimulationData.SetSampleRate( simulation_sample_rate );
	mSerialSimulationData.SetInitialBitState( BIT_HIGH );
}
```

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

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 primary task of the function is to generate the simulation data, which we typically do in a loop – checking until we have generated enough data. A clean way of doing this is to generate a complete piece (possibly a full transaction) of your protocol in a helper function. Then just repeatedly call this function until enough data has been generated. You can also add spacing between the elements of your protocol as you like.

Here is an example from *SimpleSerialSimulationDataGenerator.cpp*. We're going to be outputting chars from a string, which we initialized in our constructor as shown.

```
SimpleSerialSimulationDataGenerator::SimpleSerialSimulationDataGenerator()
:	mSerialText( "My first analyzer, woo hoo!" ),
	mStringIndex( 0 )
{
}
U32 SimpleSerialSimulationDataGenerator::GenerateSimulationData( U64
largest_sample_requested, U32 sample_rate, SimulationChannelDescriptor**
simulation_channel )
{
	U64 adjusted_largest_sample_requested =
AnalyzerHelpers::AdjustSimulationTargetSample( largest_sample_requested, sample_rate,
mSimulationSampleRateHz );

	while( mSerialSimulationData.GetCurrentSampleNumber() <
adjusted_largest_sample_requested )
	{
		CreateSerialByte();
	}

	*simulation_channel = &mSerialSimulationData;
	return 1;
}

void SimpleSerialSimulationDataGenerator::CreateSerialByte()
{
	U32 samples_per_bit = mSimulationSampleRateHz / mSettings->mBitRate;

	U8 byte = mSerialText[ mStringIndex ];
	mStringIndex++;
	if( mStringIndex == mSerialText.size() )
		mStringIndex = 0;

	//we're currenty high
	//let's move forward a little
	mSerialSimulationData.Advance( samples_per_bit * 10 );

	mSerialSimulationData.Transition();  //low-going edge for start bit
	mSerialSimulationData.Advance( samples_per_bit );  //add start bit time

	U8 mask = 0x1 << 7;
	for( U32 i=0; i<8; i++ )
```

```
		{
			if( ( byte & mask ) != 0 )
				mSerialSimulationData.TransitionIfNeeded( BIT_HIGH );
			else
				mSerialSimulationData.TransitionIfNeeded( BIT_LOW );

			mSerialSimulationData.Advance( samples_per_bit );
			mask = mask >> 1;
		}

		mSerialSimulationData.TransitionIfNeeded( BIT_HIGH ); //we need to end high

		//lets pad the end a bit for the stop bit:
		mSerialSimulationData.Advance( samples_per_bit );
	}
```

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.

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

void SerialSimulationDataGenerator::CreateSerialByte(U64 value)

```
void SerialSimulationDataGenerator::CreateSerialByte( U64 value )
{
	//assume we start high

	mSerialSimulationData.Transition();  //low-going edge for start bit
	mSerialSimulationData.Advance( mClockGenerator.AdvanceByHalfPeriod() );  //add start bit time

	if( mSettings->mInverted == true )
		value = ~value;

	U32 num_bits = mSettings->mBitsPerTransfer;

	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 ); //we want to add a zero bit
		else
			mSerialSimulationData.TransitionIfNeeded( mBitHigh ); //we want to add a one bit
```

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

        }else
        if( mSettings->mParity == AnalyzerEnums::Odd )
        {

                if( AnalyzerHelpers::IsOdd( AnalyzerHelpers::GetOnesCount( value ) ) ==
true )
                        mSerialSimulationData.TransitionIfNeeded( mBitLow ); //we want to
add a zero bit
                else
                        mSerialSimulationData.TransitionIfNeeded( mBitHigh );

                mSerialSimulationData.Advance( mClockGenerator.AdvanceByHalfPeriod() );

        }

        mSerialSimulationData.TransitionIfNeeded( mBitHigh ); //we need to end high

        //lets pad the end a bit for the stop bit:
        mSerialSimulationData.Advance( mClockGenerator.AdvanceByHalfPeriod( mSettings-
>mStopBits ) );
}
```

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, grouped under the class *AnalyzerHelpers*, include:

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


void I2cSimulationDataGenerator::CreateBit(BitState bit_state)

```
void I2cSimulationDataGenerator::CreateBit( BitState bit_state )
{
	if( mScl->GetCurrentBitState() != BIT_LOW )
		AnalyzerHelpers::Assert( "CreateBit expects to be entered with scl low" );

	mI2cSimulationChannels.AdvanceAll( mClockGenerator.AdvanceByHalfPeriod( 0.5 ) );

	mSda->TransitionIfNeeded( bit_state );

	mI2cSimulationChannels.AdvanceAll( mClockGenerator.AdvanceByHalfPeriod( 0.5 ) );

	mScl->Transition(); //posedge

	mI2cSimulationChannels.AdvanceAll( mClockGenerator.AdvanceByHalfPeriod( 1.0 ) );

	mScl->Transition(); //negedge
}
void I2cSimulationDataGenerator::CreateI2cByte( U8 data, I2cResponse reply )

void I2cSimulationDataGenerator::CreateI2cByte( U8 data, I2cResponse reply )
{
	if( mScl->GetCurrentBitState() == BIT_HIGH )
	{
		mI2cSimulationChannels.AdvanceAll( mClockGenerator.AdvanceByHalfPeriod(
1.0 ) );
		mScl->Transition();
		mI2cSimulationChannels.AdvanceAll( mClockGenerator.AdvanceByHalfPeriod(
1.0 ) );
	}

	BitExtractor bit_extractor( data, AnalyzerEnums::MsbFirst, 8 );

	for( U32 i=0; i<8; i++ )
	{
		CreateBit( bit_extractor.GetNextBit() );
	}

	if( reply == I2C_ACK )
		CreateBit( BIT_LOW );
	else
		CreateBit( BIT_HIGH );
```

```
        mI2cSimulationChannels.AdvanceAll( mClockGenerator.AdvanceByHalfPeriod( 4.0 ) );
}
```

void I2cSimulationDataGenerator::CreateI2cTransaction(U8 address, I2cDirection direction, U8 data)

```
void I2cSimulationDataGenerator::CreateI2cTransaction( U8 address, I2cDirection
direction, U8 data )
{
        U8 command = address << 1;
        if( direction == I2C_READ )
                command |= 0x1;

        CreateStart();
        CreateI2cByte( command, I2C_ACK );
        CreateI2cByte( data, I2C_ACK );
        CreateI2cByte( data, I2C_NAK );
        CreateStop();
}
U32 I2cSimulationDataGenerator::GenerateSimulationData( U64 largest_sample_requested, U32
sample_rate, SimulationChannelDescriptor** simulation_channels )
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 )
        {
                CreateI2cTransaction( 0xA0, I2C_READ, mValue++ );

                mI2cSimulationChannels.AdvanceAll( mClockGenerator.AdvanceByHalfPeriod(
10.0 ) ); //insert 10 bit-periods of idle
        }

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

void SpiSimulationDataGenerator::OutputWord_CPHA1(U64 mosi_data, U64 miso_data)

```
void SpiSimulationDataGenerator::OutputWord_CPHA1( U64 mosi_data, U64 miso_data )
{
        BitExtractor mosi_bits( mosi_data, mSettings->mShiftOrder, mSettings-
>mBitsPerTransfer );
```

```
        BitExtractor miso_bits( miso_data, mSettings->mShiftOrder, mSettings-
>mBitsPerTransfer );

        U32 count = mSettings->mBitsPerTransfer;
        for( U32 i=0; i<count; i++ )
        {
                mClock->Transition();  //data invalid
                mMosi->TransitionIfNeeded( mosi_bits.GetNextBit() );
                mMiso->TransitionIfNeeded( miso_bits.GetNextBit() );

                mSpiSimulationChannels.AdvanceAll( mClockGenerator.AdvanceByHalfPeriod( .5
) );
                mClock->Transition();  //data valid

                mSpiSimulationChannels.AdvanceAll( mClockGenerator.AdvanceByHalfPeriod( .5
) );
        }

        mMosi->TransitionIfNeeded( BIT_LOW );
        mMiso->TransitionIfNeeded( BIT_LOW );

        mSpiSimulationChannels.AdvanceAll( mClockGenerator.AdvanceByHalfPeriod( 2.0 ) );
}
```

void SpiSimulationDataGenerator::CreateSpiTransaction()

```
void SpiSimulationDataGenerator::CreateSpiTransaction()
{
        if( mEnable != NULL )
                mEnable->Transition();

        mSpiSimulationChannels.AdvanceAll( mClockGenerator.AdvanceByHalfPeriod( 2.0 ) );

        if( mSettings->mDataValidEdge == AnalyzerEnums::LeadingEdge )
        {
                OutputWord_CPHA0( mValue, mValue+1 );
                mValue++;

                OutputWord_CPHA0( mValue, mValue+1 );
                mValue++;

                OutputWord_CPHA0( mValue, mValue+1 );
                mValue++;

                if( mEnable != NULL )
```

```
                        mEnable->Transition();


                OutputWord_CPHA0( mValue, mValue+1 );
                mValue++;
        }else
        {
                OutputWord_CPHA1( mValue, mValue+1 );
                mValue++;

                OutputWord_CPHA1( mValue, mValue+1 );
                mValue++;

                OutputWord_CPHA1( mValue, mValue+1 );
                mValue++;

                if( mEnable != NULL )
                        mEnable->Transition();

                OutputWord_CPHA1( mValue, mValue+1 );
                mValue++;
        }
}
```

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

```
U32 SpiSimulationDataGenerator::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( mClock->GetCurrentSampleNumber() < adjusted_largest_sample_requested )
        {
                CreateSpiTransaction();

                mSpiSimulationChannels.AdvanceAll( mClockGenerator.AdvanceByHalfPeriod(
10.0 ) ); //insert 10 bit-periods of idle
        }

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


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.

{YourName}AnalyzerResults.h

In addition to the constructor and destructor, there are 5 functions we'll need to implement. *AnalyzerResults* is fairly straightforward, so typically we won't need much in the way of helper functions or member variables.

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

```
#ifndef SIMPLESERIAL_ANALYZER_RESULTS
#define SIMPLESERIAL_ANALYZER_RESULTS

#include <AnalyzerResults.h>

class SimpleSerialAnalyzer;
class SimpleSerialAnalyzerSettings;

class SimpleSerialAnalyzerResults : public AnalyzerResults
{
public:
	SimpleSerialAnalyzerResults( SimpleSerialAnalyzer* analyzer,
SimpleSerialAnalyzerSettings* settings );
	virtual ~SimpleSerialAnalyzerResults();

	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
        SimpleSerialAnalyzerSettings* mSettings;
        SimpleSerialAnalyzer* mAnalyzer;
};

#endif //SIMPLESERIAL_ANALYZER_RESULTS
```

{YourName}AnalyzerResults.cpp

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 *SimpleSerialAnalyzerResults.cpp:*

```
SimpleSerialAnalyzerResults::SimpleSerialAnalyzerResults( SimpleSerialAnalyzer* analyzer,
SimpleSerialAnalyzerSettings* settings )
:       AnalyzerResults(),
        mSettings( settings ),
        mAnalyzer( analyzer )
{
}
```

Frames, Packets, and Transactions

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( const Frame& 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*.

```
#define DISPLAY_AS_ERROR_FLAG ( 1 << 7 )
#define DISPLAY_AS_WARNING_FLAG ( 1 << 6 )
```

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. In most situations, this can simply be ignored. If you need to use it, just compare it to the channels saved in your *mSettings* object to see which bubble should be generated – for example, for the MISO or MOSI channel.

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 };

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 ); //multiple
strings will be concatinated
```

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;

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

        char result_str[128];

        if( ( parity_error == true ) || ( framing_error == true ) )
        {
                AddResultString( "!" );

                sprintf( result_str, "%s (error)", number_str );
                AddResultString( result_str );

                if( parity_error == true && framing_error == false )
                        sprintf( result_str, "%s (parity error)", number_str );
                else
                if( parity_error == false && framing_error == true )
                        sprintf( result_str, "%s (framing error)", number_str );
                else
                        sprintf( result_str, "%s (framing error & parity error)", number_str );

                AddResultString( result_str );

        }else
        {
                AddResultString( number_str );
        }
}
```

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. (See *GenerateBubbleText* for more detail)

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


        U64 trigger_sample = mAnalyzer->GetTriggerSample();

        U32 sample_rate = mAnalyzer->GetSampleRate();


        file_stream << "Time [s],Value,Parity Error,Framing Error" << std::endl;


        U64 num_frames = GetNumFrames();

        for( U32 i=0; i < num_frames; i++ )

        {

                Frame frame = GetFrame( i );


                //static void GetTimeString( U64 sample, U64 trigger_sample, U32
sample_rate_hz, char* result_string, U32 result_string_max_length );

                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 );
```



```
                file_stream << time_str << "," << number_str;

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

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

                file_stream << std::endl;

                if( UpdateExportProgressAndCheckForCancel( i, num_frames ) == true )
                {
                        file_stream.close();
                        return;
                }
        }

        file_stream.close();
}
```

void SerialAnalyzerResults::GenerateFrameTabularText(U64 frame_index, DisplayBase display_base)

GenerateFrameTabularText is for producing text for tabular display which is not yet implemented as of 1.1.5. 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 )
{
        Frame frame = GetFrame( frame_index );
        ClearResultStrings();

        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;


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


        char result_str[128];


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

void SerialAnalyzerResults::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.

void SerialAnalyzerResults::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.

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.

{YourName}Analyzer.h

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

```
		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();

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.

{YourName}Analyzer.cpp

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.

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.

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.

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.

*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 );

}
```

U32 SerialAnalyzer::GetMinimumSampleRateHz()

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

For Serial, for instance, we would like the sample rate to be x4 higher that the serial bit rate.

For other, typically synchronous, protocols, you may not ask the user to enter the data's bit rate – therefore you can't know ahead of time what sample rate is required. In that case, you can either return the smallest sample rate (25000), or return a value that will be fast enough for your simulation. However, your simulation really should adjust its own rate depending on the sample rate – for example, when simulation SPI you should probably make the bit rate something like 4x the sample rate. This will allow the simulation to work perfectly no matter what the sample rate is.

The rule of thumb is to require oversampling by x4 if you know the data's bit rate, otherwise just return 25000.

Here's what we do in *SerialAnalyzer.cpp*

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

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

Simply return the name you would like to see in the "Add Analyzer" drop down.

```
return "Async Serial";
```

const char GetAnalyzerName()*

Return the same string as in the previous function.

```
return "Async Serial";
```

Analyzer CreateAnalyzer()*

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

```
return new {YourName}Analyzer();
```

void DestroyAnalyzer(Analyzer analyzer)*

Simply call *delete* on the provided pointer.


```
delete analyzer;
```

void {YourName}Analyzer::WorkerThread()

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.

AnalyzerChannelData – Keeping track of the smallest pulse.

When we were implementing Serial's "autobaud" it was clear that keeping track of the minimum pulse length over the entire stream was overly cumbersome. If you need this capability for some reason, these functions will provide it for you (it's turned off by default)

```
void TrackMinimumPulseWidth();
U64 GetMinimumPulseWidthSoFar();
```

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 (*DISPLAY_AS_WARNING_FLAG*, and *DISPLAY_AS_ERROR_FLAG*).

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 );
```

The *transaction_id* is an ID you generate yourself.

The analyzers created by Saleae do not yet use *Transactions*, and the current Analyzer probably never will. *Transactions* are provided for higher-level protocols, and you may not want to bother, especially since they aren't used in the Logic software yet. We will use *Transactions* in analyzers for more sophisticated protocols in the future.

Packets on the other hand tend to be fairly applicable for lower level protocls, although not in entirly the same ways. For example:

- Serial Analyzer – no packet support makes sense at this level. (there are many more structured protocols that use asynchronous serial where packets would be applicable)
- SPI Analyzer – packets are used to delimit between periods when the enable line is active.
- I2C Analyzer – packets are used to delimit periods between a start/restart and a stop.
- CAN Analyzer – packets are used to represent, well, CAN packets.
- UNI/O – packets are used to group Frames in a UNI/O sequence.
- 1-Wire – packets are used to group 1-Wire sequences.
- I2S/PCM – packets aren't used.

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.

