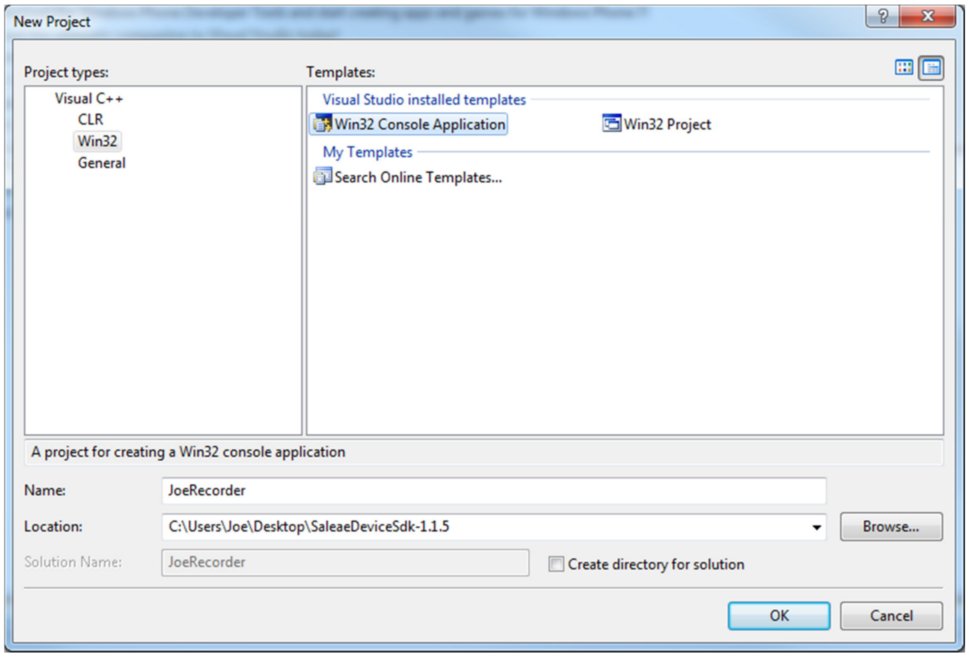
Contents

Starting a Device SDK Project on Windows

The instructions that follow are for setting up a Visual Studio project from scratch. Included in the SDK is a vs2008 project called *ConsoleDemo*, which should immediately build and run. If you like, you can skip the following section, and return to it if you have any questions about how a Visual Studio project should be set up.

1. If you haven't already, download the latest *SaleaeDeviceSdk-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). Customers have reported that using VS2010 will also work.
3. File->New->Project
   a. Visual C++; Win32
   b. Win32 Console Application (under Templates)
   c. Name: enter a name for the project, such as **MyRecorder**
   d. Location: *Desktop*\SaleaeDeviceSdk-1.1.x
   e. Make sure Create directory for solution is <u>not</u> checked.
   f. Press OK.



A project for creating a Win32 console application

Name: JoeRecorder

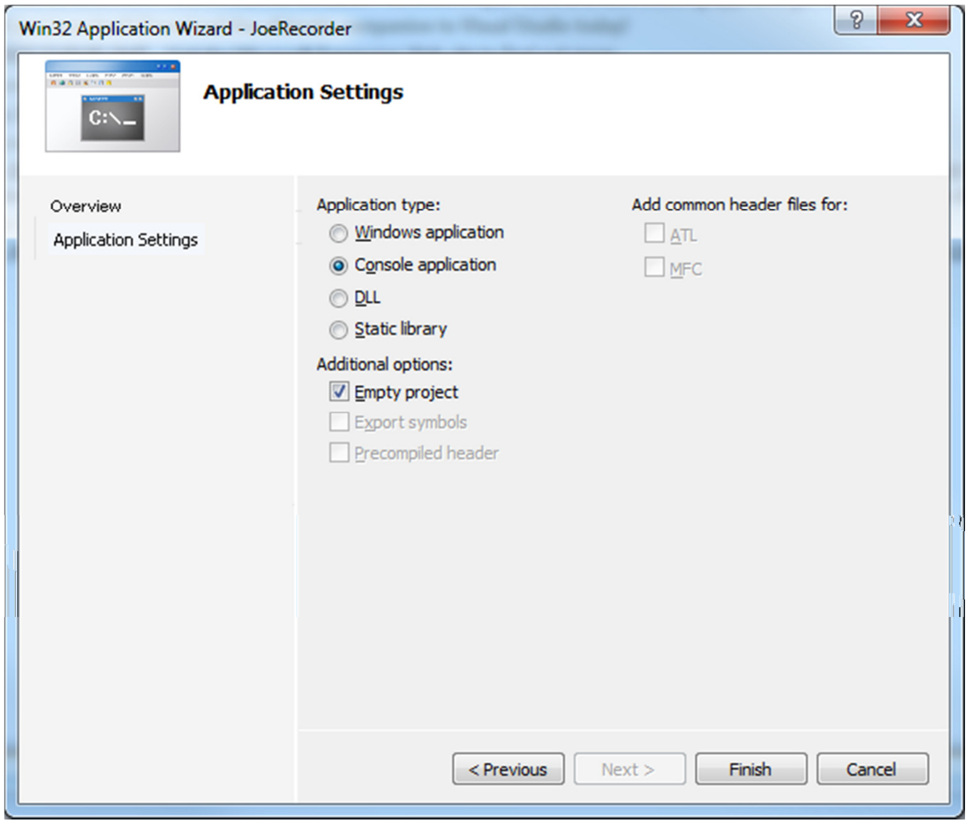
Location: C:\Users\Joe\Desktop\SaleaeDeviceSdk-1.1.5

Solution Name: JoeRecorder

☐ Create directory for solution

   g. Click Application Settings
      i. Application Type: Console application
      ii. Additional options:
         1. Precompiled header (not checked)
         2. Empty project (checked)

h. Click Finish.

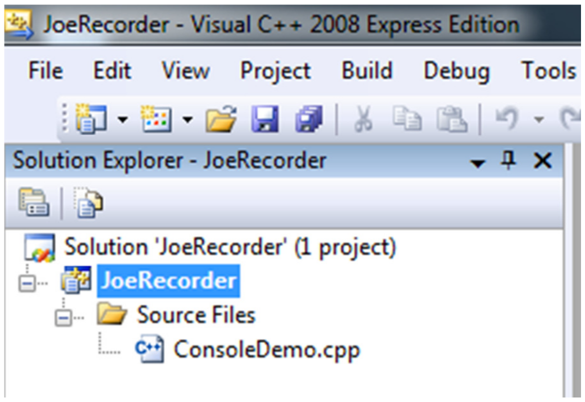


Application type:
- [ ] Windows application
- [✓] Console application
- [ ] DLL
- [ ] Static library

Additional options:
- [✓] Empty project
- [ ] Export symbols
- [ ] Precompiled header

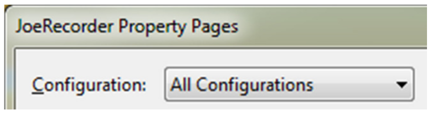
Add common header files for:
- [ ] ATL
- [ ] MFC

4. Delete the Header Files and Resource Files folders, from under our project item in the Solution Explorer.
5. Open *the SaleaeDeviceSdk-1.1.x* folder, and copy the *source* folder into your new project folder.
   a. Note: this will allow modifications to your own copy of the source, and leave the originals unchanged. If this is not desired, you can simply use the source files in their original location.
6. In Visual Studio, right click on the Source Files folder, and select Add->Existing Item.
7. Navigate to your new project folder, into the *source* folder (that we just copied) and select *ConsoleDemo.cpp*.

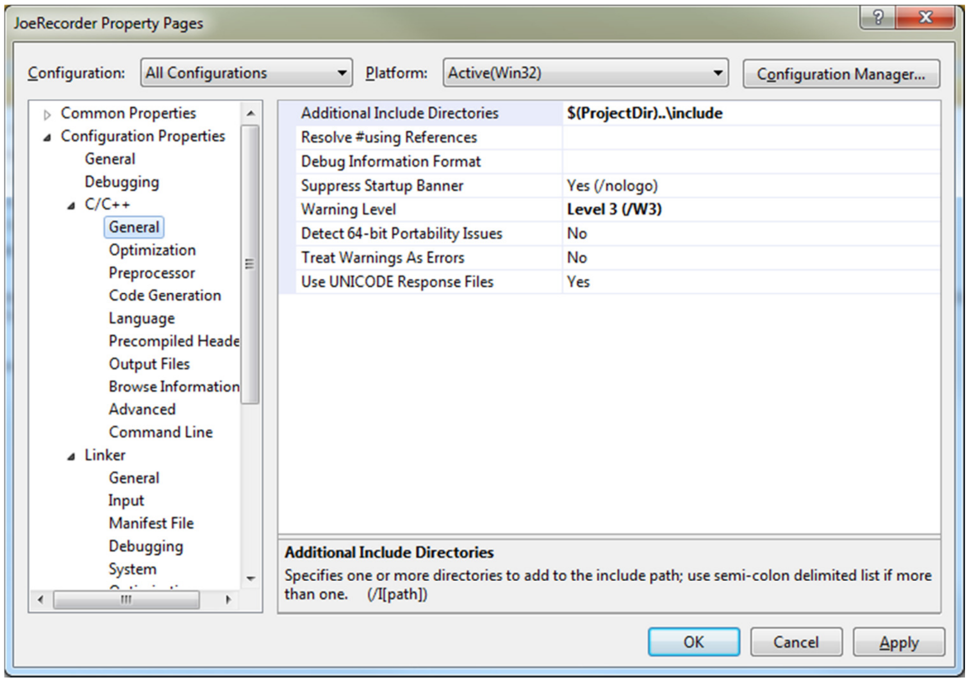

JoeRecorder - Visual C++ 2008 Express Edition
File
Edit
View
Project
Build
Debug
Tools
Solution Explorer - JoeRecorder

Solution 'JoeRecorder' (1 project)
JoeRecorder
Source Files
ConsoleDemo.cpp

8. In the Solution Explorer, right click on the project item (not the solution item) and choose Properties.
    a. Under Configuration, select All Configurations.



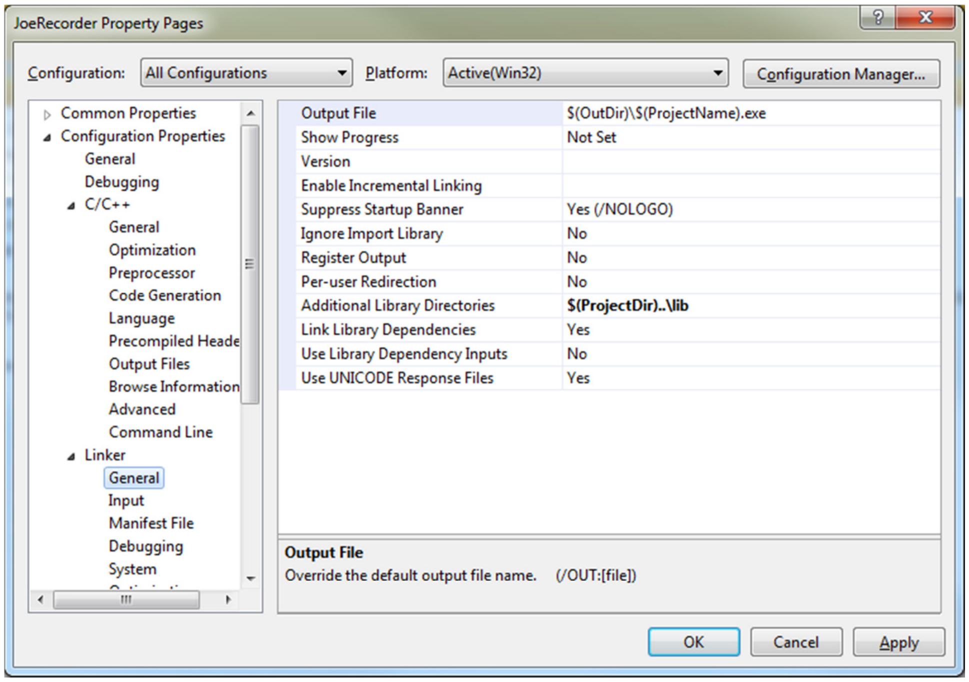
    b. Under C/C++, select General. Under Additional Include Directories, enter **$(ProjectDir)..\include**



Additional Include Directories	**$(ProjectDir)..\include**
Resolve #using References	
Debug Information Format	
Suppress Startup Banner	Yes (/nologo)
Warning Level	**Level 3 (/W3)**
Detect 64-bit Portability Issues	No
Treat Warnings As Errors	No
Use UNICODE Response Files	Yes

Additional Include Directories
Specifies one or more directories to add to the include path; use semi-colon delimited list if more than one. (/I[path])

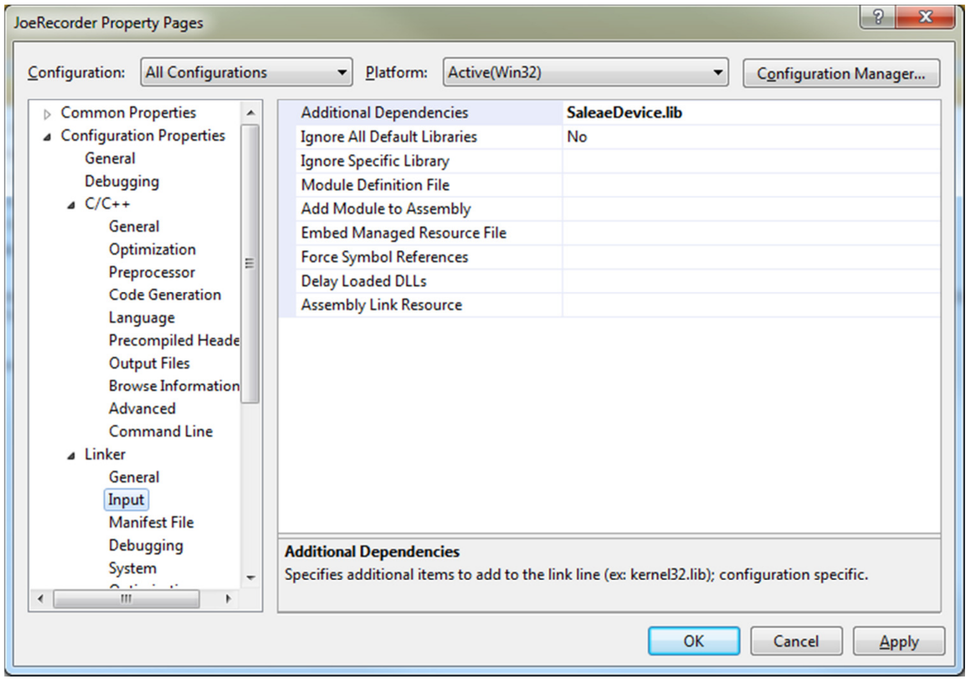
    c. Expand Linker and select General. Under Additional Library Directories, enter **$(ProjectDir)..\lib**



Output File	$(OutDir)\$(ProjectName).exe
Show Progress	Not Set
Version	
Enable Incremental Linking	
Suppress Startup Banner	Yes (/NOLOGO)
Ignore Import Library	No
Register Output	No
Per-user Redirection	No
Additional Library Directories	**$(ProjectDir)..\lib**
Link Library Dependencies	Yes
Use Library Dependency Inputs	No
Use UNICODE Response Files	Yes

Output File
Override the default output file name. (/OUT:[file])

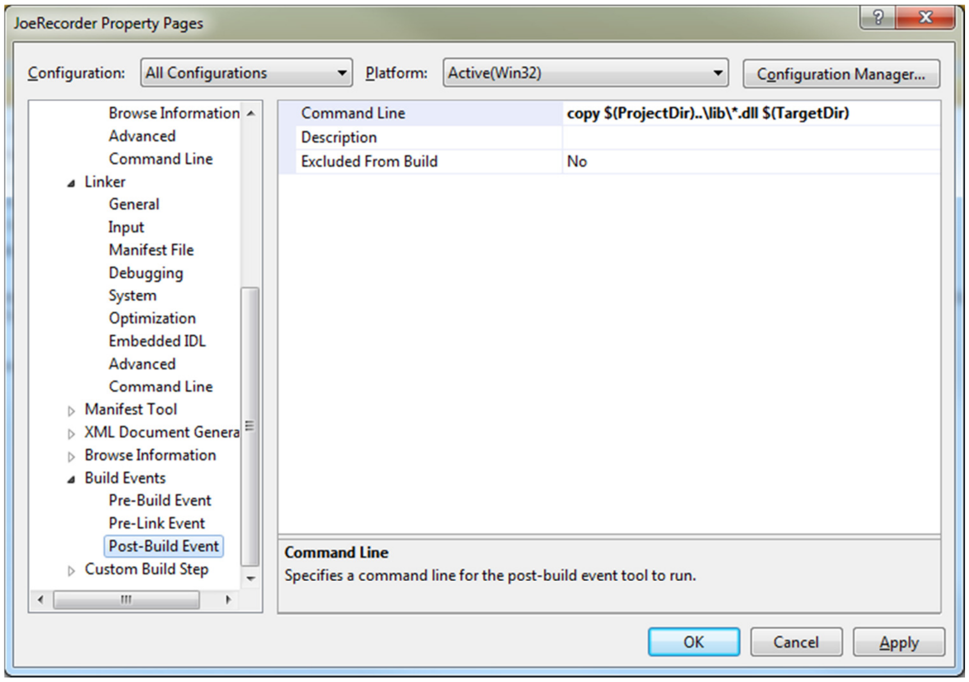
d. Under Linker, select Input. Under Additional Dependencies, enter **SaleaeDevice.lib**



Additional Dependencies	SaleaeDevice.lib
Ignore All Default Libraries	No
Ignore Specific Library	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded DLLs	
Assembly Link Resource	

Additional Dependencies
Specifies additional items to add to the link line (ex: kernel32.lib); configuration specific.

e. Expand Build Events, and select Post-build Event.
   i. Under Command Line, enter **copy $(ProjectDir)..\lib*.dll $(TargetDir)**
   ii. This will copy required DLLs to the same location as the executable. In production, make sure these DLLs are included with the executable, in the same directory.



Command Line	copy $(ProjectDir)..\lib*.dll $(TargetDir)
Description	
Excluded From Build	No

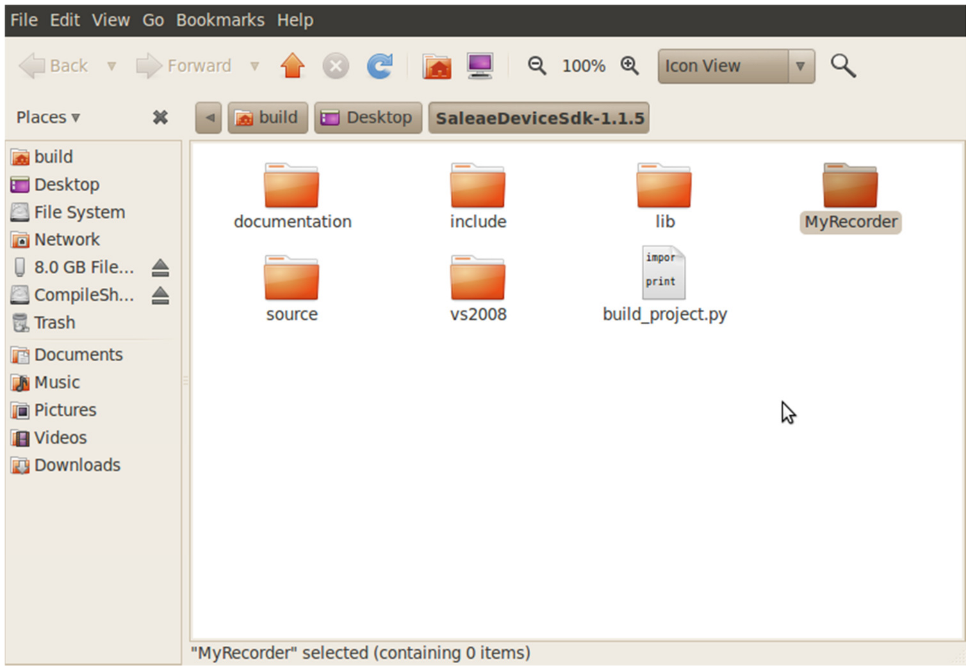
Command Line
Specifies a command line for the post-build event tool to run.

   f. Click OK.

9. You should now be able to build and run the application.

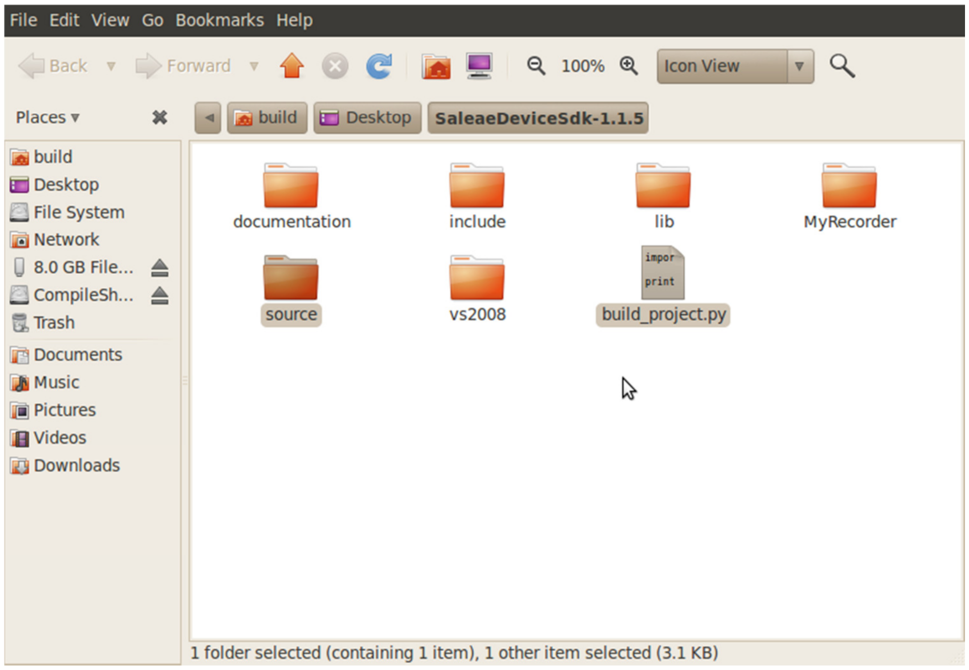
Starting a Device SDK Project on Linux

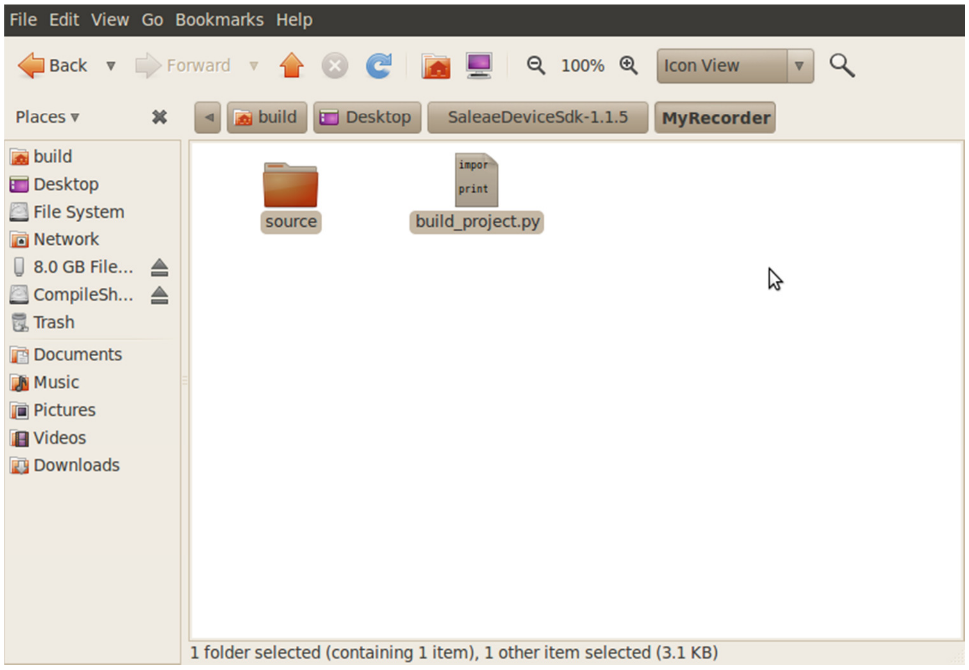
1. If you haven't already, download and extract the *SaleaeDeviceSdk-1.1.x* to your desktop, or other convenient location.
2. Decide on a name for your project, **MyRecorder**, for example. Open the *SaleaeDeviceSdk-1.1.x* folder and create a new folder with this name.



"MyRecorder" selected (containing 0 items)

3. Select the *source* folder, and file *build_project.py*. Copy and paste these into your new project folder.

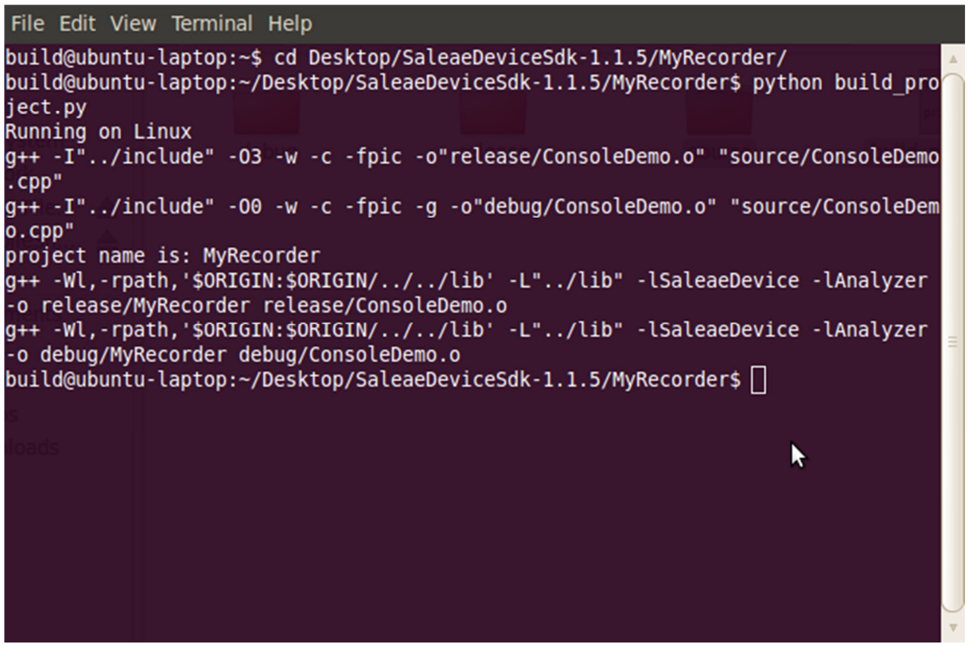




4. Open console and navigate to your new project folder. Something like: **cd Desktop\SaleaeDeviceSdk-1.1.5\MyRecorder**

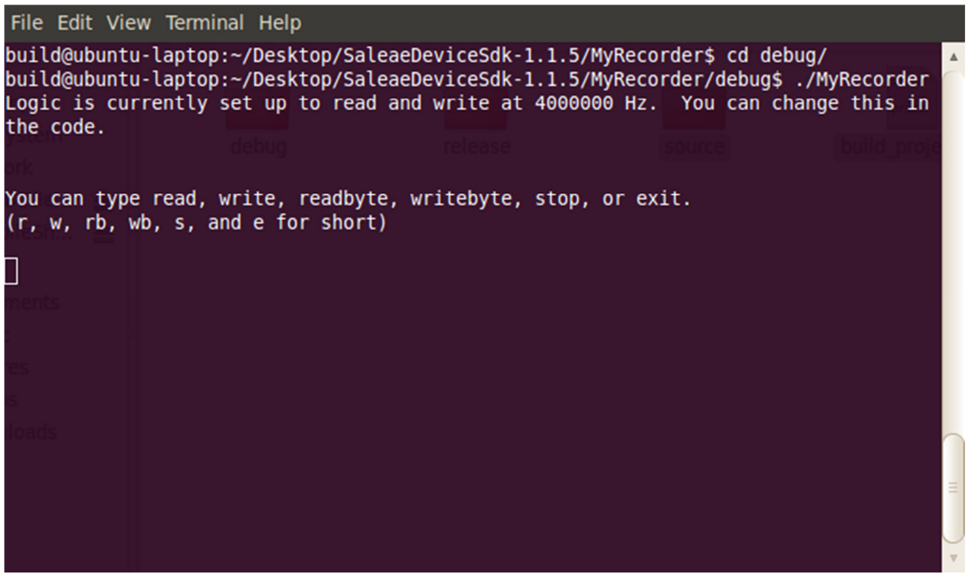
5. Inside the source folder we copied is a single cpp file, *ConsoleDemo.cpp*. You can modify this, or replace it, but for now we'll leave it as is and get it building.

6. From the console, type **python build_project.py**



```
build@ubuntu-laptop:~$ cd Desktop/SaleaeDeviceSdk-1.1.5/MyRecorder/
build@ubuntu-laptop:~/Desktop/SaleaeDeviceSdk-1.1.5/MyRecorder$ python build_pro
ject.py
Running on Linux
g++ -I"../include" -O3 -w -c -fpic -o"release/ConsoleDemo.o" "source/ConsoleDemo
.cpp"
g++ -I"../include" -O0 -w -c -fpic -g -o"debug/ConsoleDemo.o" "source/ConsoleDem
o.cpp"
project name is: MyRecorder
g++ -Wl,-rpath,'$ORIGIN:$ORIGIN/../../lib' -L"../lib" -lSaleaeDevice -lAnalyzer
-o release/MyRecorder release/ConsoleDemo.o
g++ -Wl,-rpath,'$ORIGIN:$ORIGIN/../../lib' -L"../lib" -lSaleaeDevice -lAnalyzer
-o debug/MyRecorder debug/ConsoleDemo.o
build@ubuntu-laptop:~/Desktop/SaleaeDeviceSdk-1.1.5/MyRecorder$
```

7. This will create two folders -- *debug* and *release* -- and build the application in each.
8. To execute the new program navigate to the *debug* folder and run the program:
    a. **cd debug**
    b. **./MyRecorder** (replace MyRecorder with the name of your project)



```
build@ubuntu-laptop:~/Desktop/SaleaeDeviceSdk-1.1.5/MyRecorder$ cd debug/
build@ubuntu-laptop:~/Desktop/SaleaeDeviceSdk-1.1.5/MyRecorder/debug$ ./MyRecorder
Logic is currently set up to read and write at 4000000 Hz.  You can change this in
the code.

You can type read, write, readbyte, writebyte, stop, or exit.
(r, w, rb, wb, s, and e for short)
```

9. To exit the program type **exit**, or **e**.

Debugging your Project with GDB

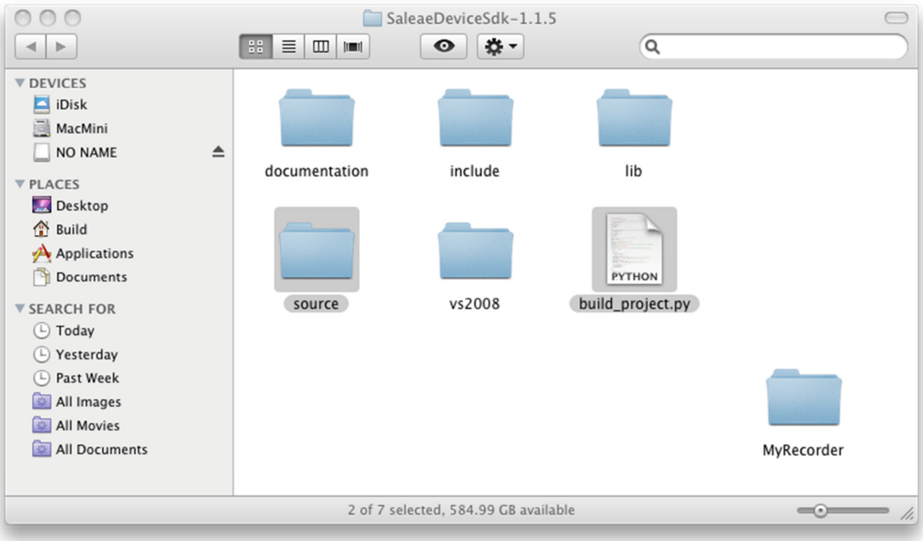
1. From your application's debug folder, type **gdb MyRecorder** (substitute your project name)
2. Set a breakpoint to be fired when Logic connects:

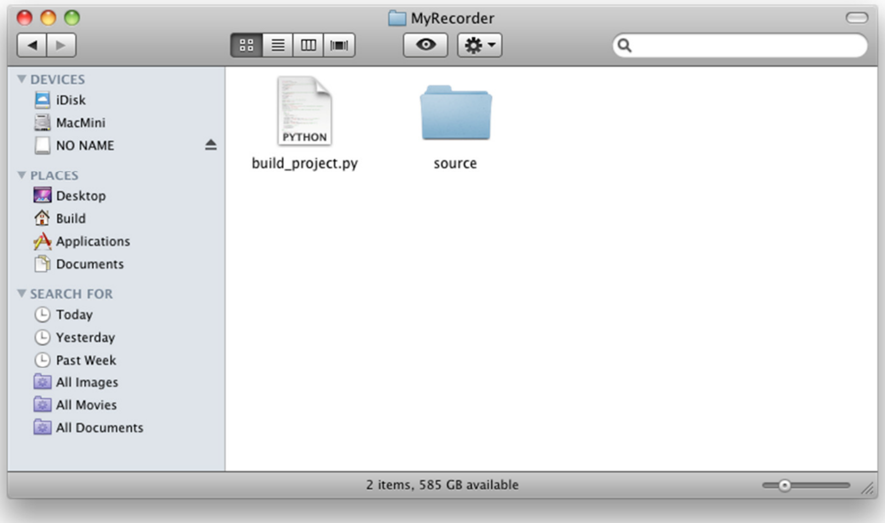
    a. type**: break ::OnConnect**
3. Start the program
    a. Type **run**
4. You should get a breakpoint when you connect Logic.
5. Type **step** to step line by line after a breakpoint has been hit, and **continue** to continue normal execution.
6. More in-depth use of GDB is outside the scope of this document.

Starting a Device SDK Project on Mac

Build Script / Command Line / GDB based Device Project

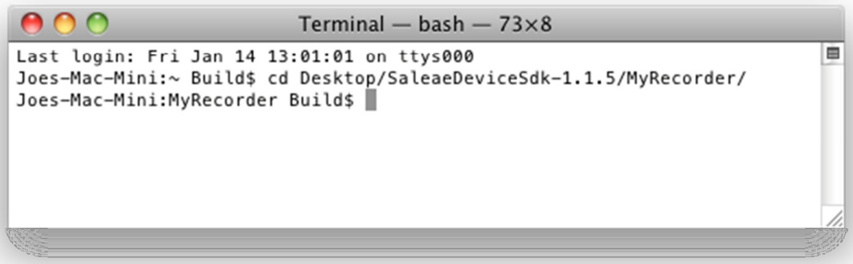
1. If you haven't already, download and extract the *SaleaeDeviceSdk-1.1.x* to your desktop, or other convenient location.
2. Decide on a name for your project, **MyRecorder**, for example. Open the *SaleaeDeviceSdk-1.1.x* folder and create a new folder with this name.
3. Select the *source* folder, and file *build_project.py*. Copy and paste these into your new project folder.





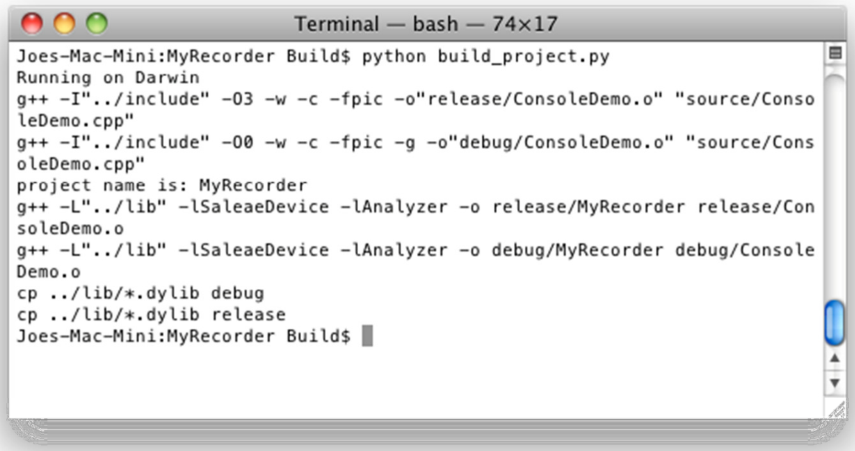
4. Open the terminal (under *Applications/Utilities*) and navigate to your new project folder.

   a. Something like **cd Desktop\SaleaeDeviceSdk-1.1.5\MyRecorder**



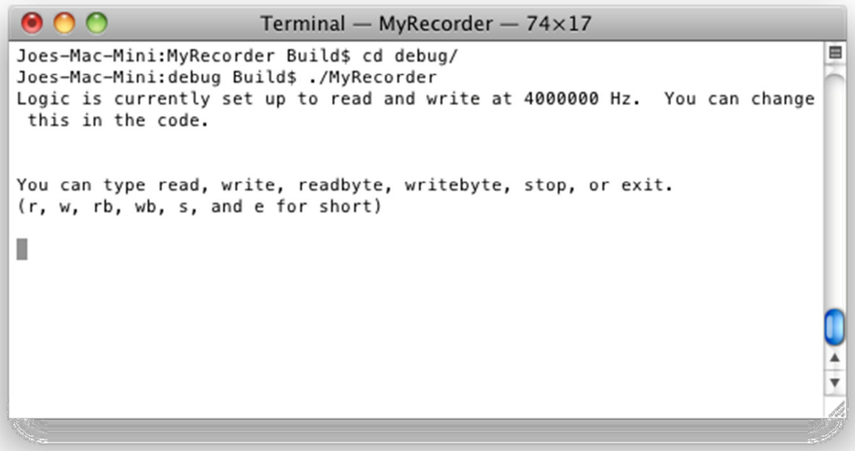
```
Last login: Fri Jan 14 13:01:01 on ttys000
Joes-Mac-Mini:~ Build$ cd Desktop/SaleaeDeviceSdk-1.1.5/MyRecorder/
Joes-Mac-Mini:MyRecorder Build$
```

5. Inside the source folder we copied is a single cpp file, *ConsoleDemo.cpp*. You can modify this, or replace it, but for now we'll leave it as is and get it building.
6. From the console, type **python build_project.py**



```
Joes-Mac-Mini:MyRecorder Build$ python build_project.py
Running on Darwin
g++ -I"../include" -O3 -w -c -fpic -o"release/ConsoleDemo.o" "source/Conso
leDemo.cpp"
g++ -I"../include" -O0 -w -c -fpic -g -o"debug/ConsoleDemo.o" "source/Cons
oleDemo.cpp"
project name is: MyRecorder
g++ -L"../lib" -lSaleaeDevice -lAnalyzer -o release/MyRecorder release/Con
soleDemo.o
g++ -L"../lib" -lSaleaeDevice -lAnalyzer -o debug/MyRecorder debug/Console
Demo.o
cp ../lib/*.dylib debug
cp ../lib/*.dylib release
Joes-Mac-Mini:MyRecorder Build$
```

7. This will create two folders -- *debug* and *release* -- and build the application in each. Note that the libraries the executable needs are also copied in from the Sdk's *lib* folder.
8. To execute the new program navigate to the *debug* folder and run the program
   a. Type **cd debug**
   b. Type **./MyRecorder** (replace MyRecorder with the name of your project)



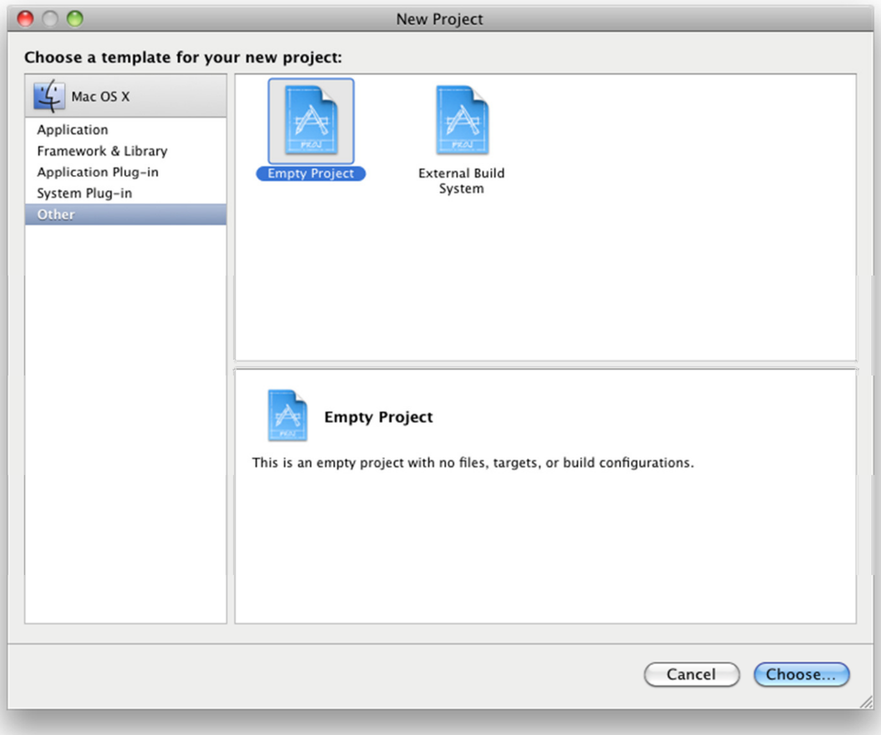
```
Joes-Mac-Mini:MyRecorder Build$ cd debug/
Joes-Mac-Mini:debug Build$ ./MyRecorder
Logic is currently set up to read and write at 4000000 Hz.  You can change
 this in the code.


You can type read, write, readbyte, writebyte, stop, or exit.
(r, w, rb, wb, s, and e for short)
```

9. To exit the program type **exit**, or **e**.

Making a SaleaeDevice Project with Xcode

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.

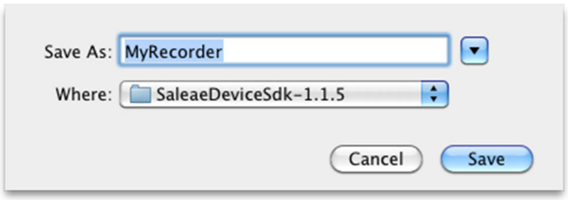
1. Start XCode
2. From the File menu, choose New Project
3. For Choose a template for your new Project select Other, Empty Project. Click Choose.



Empty Project

This is an empty project with no files, targets, or build configurations.

4. Choose a name for your project, such as **MyRecorder**, etc. It needs to be one word. Specify this as the project's name.

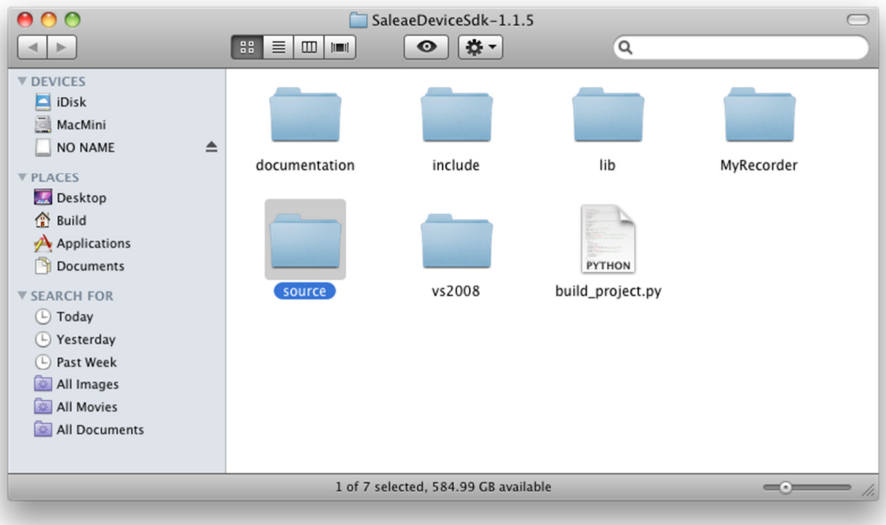


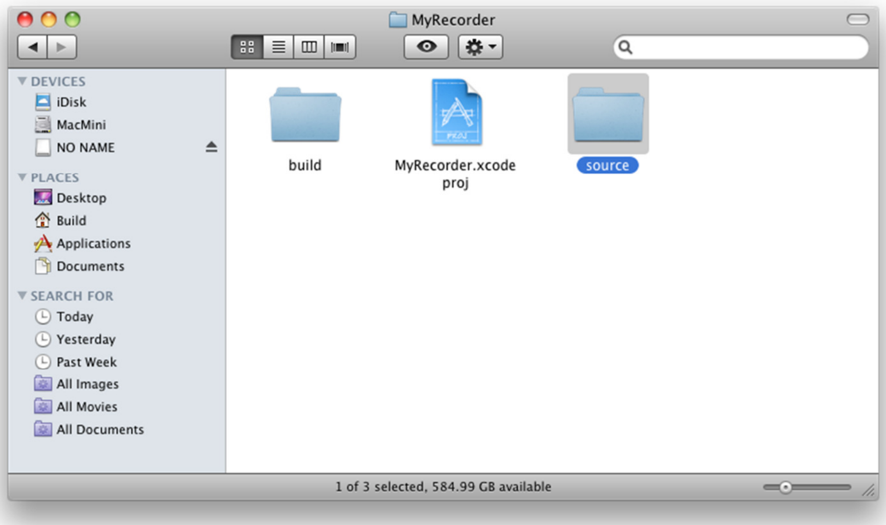
Save As: MyRecorder

Where: SaleaeDeviceSdk-1.1.5

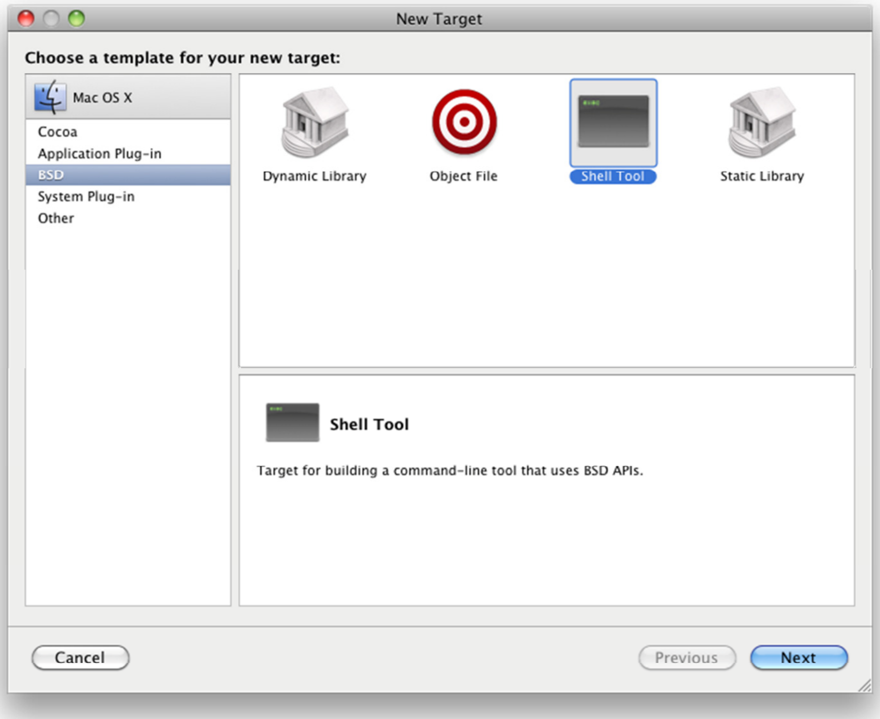
5. Save the project in the root of the *SaleaeDeviceSdk-1.1.x* folder. This will create a folder called *YourProjectName*

6. Open the folder *SaleaeDeviceSdk-1.1.x* in Finder. Copy the *source* folder and paste it into your newly created project folder.





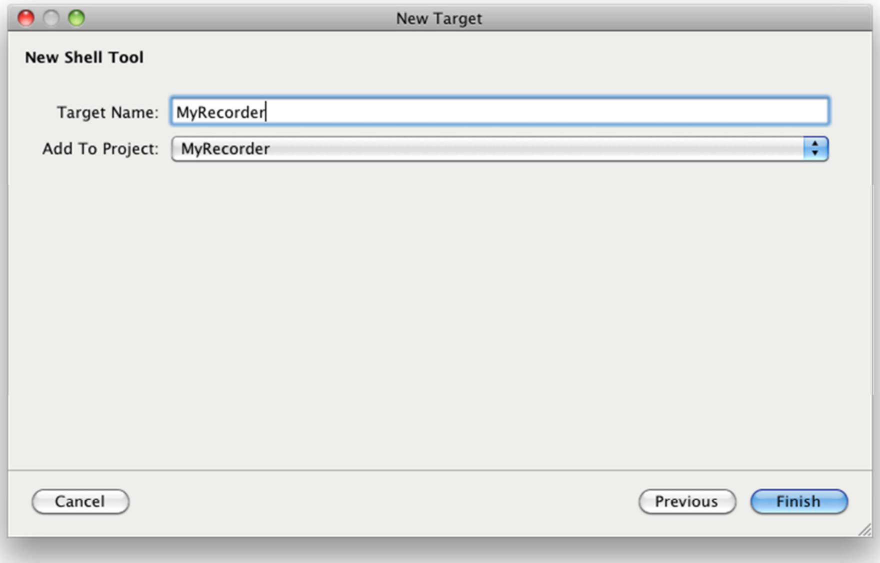
7. In XCode, under Groups & Files, right-click on Targets, and select Add->New Target
    a. Select BSD, Shell Tool, and click Next.



Shell Tool

Target for building a command-line tool that uses BSD APIs.

b. For Target Name, enter your project's name (**MyRecorder** as an example). Click Finish.



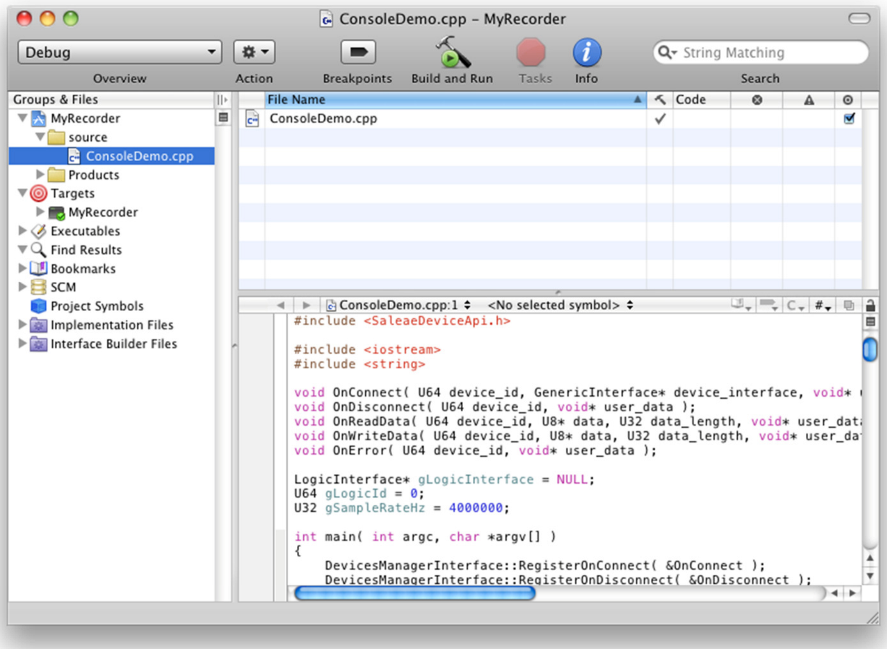
Target Name: MyRecorder

Add To Project: MyRecorder

c. This will open the Target Info window. Close this.

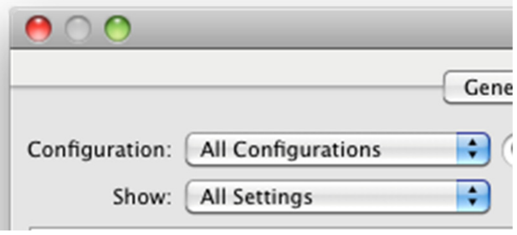
8. In the Groups & Files list, select the project item at the very top of the list. Right click and select Add->Existing Files.
    a. Navigate to your *source* folder and select it. Click Add.

b. The defaults should be fine. Click Add.



```
#include <SaleaeDeviceApi.h>

#include <iostream>
#include <string>

void OnConnect( U64 device_id, GenericInterface* device_interface, void* 
void OnDisconnect( U64 device_id, void* user_data );
void OnReadData( U64 device_id, U8* data, U32 data_length, void* user_data
void OnWriteData( U64 device_id, U8* data, U32 data_length, void* user_da
void OnError( U64 device_id, void* user_data );

LogicInterface* gLogicInterface = NULL;
U64 gLogicId = 0;
U32 gSampleRateHz = 4000000;

int main( int argc, char *argv[] )
{
    DevicesManagerInterface::RegisterOnConnect( &OnConnect );
    DevicesManagerInterface::RegisterOnDisconnect( &OnDisconnect );
```

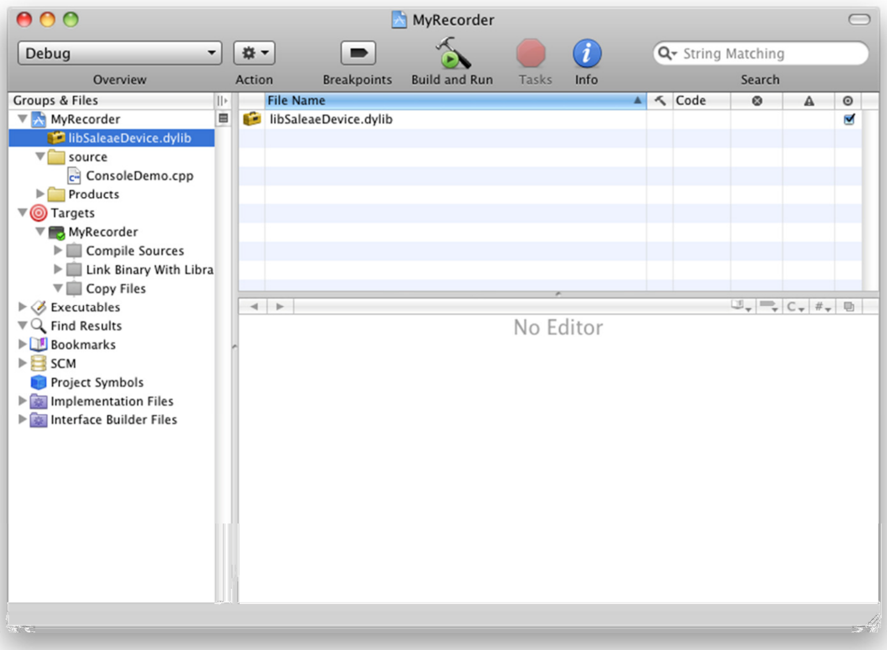
9. Select the project item (at the top of the list), and click the Info button on the main toolbar.
    a. Click the Build tab.
    b. Set Configuration to All Configurations, and set Show to All Settings

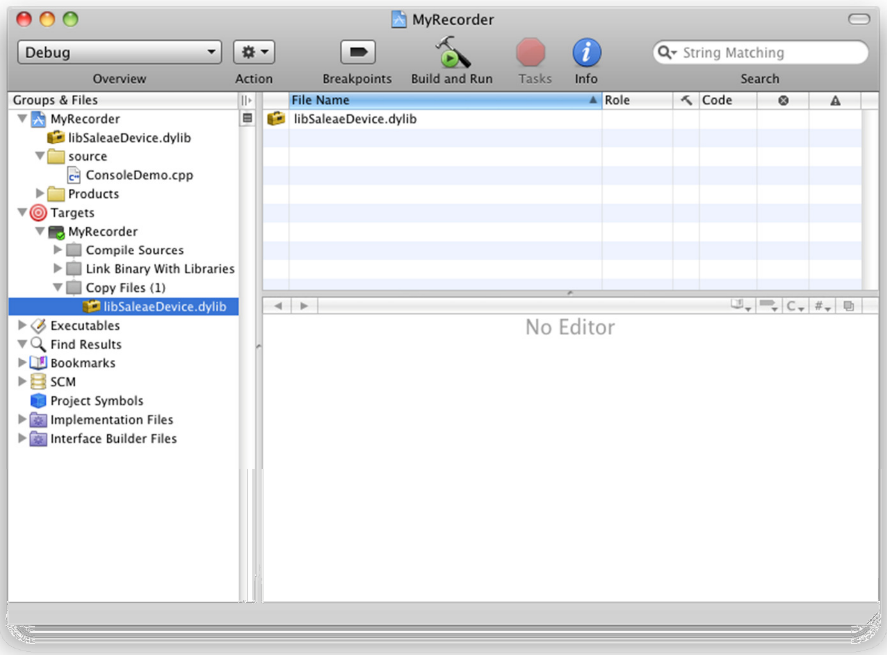


Configuration: All Configurations
Show: All Settings

    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 this item and select Add->Existing Files.
    b. Navigate to the *lib* folder, in *SaleaeDeviceSDK-1.1.5*. Select this file:
        i. libSaleaeDevice.dylib
    c. The defaults should be fine. Click Add.
11. From the main menu, select Build->Build. Your project should build completely. However, it won't run because we need to copy the dylib into the same folder as the executable.

12. Under the Targets item select your executable (MyRecorder, in our case). Right-click on this and select Add->New Build Phase->New Copy Files Build Case.
    a. For Destination, choose Executables. Close the window.
    b. Notice that under the project item is the library we're linking against: *libSaleaeDevice.dylib*. Drag these to the new Copy Files item we just created under Target. (Under Groups & Files).





13. From the main menu, select Build->Build. This should copy the required libraries to the same location as our executable.

Running & Debugging your Project

14. In XCode, open the source file (*ConsoleDemo.cpp* if you're using the original file). This will be in the source folder, under the project item.
15. Go down to the first line on *OnConnect* and click in the margin to create a breakpoint.
16. From the Build menu (at the top of the Mac desktop), select Build and Debug.
17. Click the little GDB Icon (a little black console with the letters "gdb" on it).
18. The application should run, and XCode should break execution at your breakpoint when you plug in Logic.

ConsoleDemo.cpp – a Walkthrough

Include Files

We'll need to include The *SaleaeDeviceApi.h* file – this is the SDK header file.

```
#include <SaleaeDeviceApi.h>
```

In *ConsoleDemo*, we'll also need *iostream* and *string*. These aren't needed for anything SDK specific.

```
#include <iostream>
#include <string>
```

Logic16 vs Logic support

While the SDK can manage more than one device, and more than one type of device – to keep things simple we've added a define to control whether the demo works with Logic, or with Logic16.

Static callback functions

The Saleae Device SDK uses callback functions extensively. Internally we use a variety of callback wrappers (boost, QT), but to reduce library dependencies to virtually nil, we'll just be using regular old c/c++ callbacks in our Device SDK.

```
void OnConnect( U64 device_id, GenericInterface* device_interface, void* user_data );
void OnDisconnect( U64 device_id, void* user_data );
void OnReadData( U64 device_id, U8* data, U32 data_length, void* user_data );
void OnWriteData( U64 device_id, U8* data, U32 data_length, void* user_data );
void OnError( U64 device_id, void* user_data );
```

As you can see, we can get called when a device is connected or disconnected, when it sends us data, when we need to provide it with data, or when something goes wrong (Generally the "I couldn't keep up at this sample rate error").

Notice that each of these functions provides a *user_data* parameter. When you register to receive a callback, you can specify what you want passed there. This is often used in c++ so you can provide a pointer to your class. In standard c/c++, you can't register a non-static member function to receive a callback. Passing a pointer to your object via *user_data* provides an effective, if cumbersome and crude, way around this limitation.

Notice that each of these callbacks also provides a *device_id*. This is a 64-bit number that uniquely identifies a particular Saleae device. You could manage several devices, and communicate with all of them for instance. Note however that Logic does not provide a means to synchronize captures from multiple Logics, so in practice combining Logics may not be highly useful. The ability to work with several deferent Saleae devices – including ones of different types, is provided for future compatibility.

Global Variables

ConsoleDemo is pretty simple. We're not defining any classes, like you might want to do. We're just defining standard static functions, and therefore we'll need a handful of global variables.

```
        LogicInterface* gLogicInterface = NULL;
        U64 gLogicId = 0;
        U32 gSampleRateHz = 4000000;
```

In particular, we'll hold on to a pointer to Logic, its Device ID, and the sample rate we'll be using. If we wanted to work with multiple devices, we'd have to do a better job keeping track of the different devices.

Main

The first thing we'll do is call functions in *DevicesManagerInterface* to register *OnConnect* and *OnDisconnect* callbacks. As soon as we call *BeginConnect*, we can get called on those functions. Note that the calls will occur on another thread.

```
        int main( int argc, char *argv[] )
        {
                DevicesManagerInterface::RegisterOnConnect( &OnConnect );
                DevicesManagerInterface::RegisterOnDisconnect( &OnDisconnect );
                DevicesManagerInterface::BeginConnect();
```

The next thing we do is start prompting for console input.

Before we get into that however, let's handle *OnConnect* and *OnDisconnect.*

OnConnect Callback

```
        void OnConnect( U64 device_id, GenericInterface* device_interface, void* user_data )
        {
                if( dynamic_cast<LogicInterface*>( device_interface ) != NULL )
                {
                        std::cout << "A Logic device was connected (id=0x" << std::hex <<
        device_id << std::dec << ")." << std::endl;


                        gLogicInterface = (LogicInterface*)device_interface;
                        gLogicId = device_id;


                        gLogicInterface->RegisterOnReadData( &OnReadData );
                        gLogicInterface->RegisterOnWriteData( &OnWriteData );
                        gLogicInterface->RegisterOnError( &OnError );


                        gLogicInterface->SetSampleRateHz( gSampleRateHz );
```

```
		}
	}
```

The SDK allows for different Saleae devices to be supported. In *OnConnect*, we need to test the *device_interface* pointer to see what type it is. If *device_interface* is a pointer to a *LogicInterface* object, then we can start using it.

We set our global *gLogicInterface* pointer to this new pointer (again, this isn't very sophisticated, but it'll work for our purposes). We set our *gLogicId* variable. Then we register the three callbacks that are specific to the *LogicInterface* class: *OnReadData*, *OnWriteData*, and *OnError*.

Lastly we set the sample rate. The device can now be used.

Note that technically we should be using mutexes when we access our global variables as we're actually accessing them on two different threads. In this simple application we can get away with not bothering. A more robust application would need to take care to prevent simultaneous access of any member variables that are accessed from multiple threads.

OnDisconnect Callback

```
	void OnDisconnect( U64 device_id, void* user_data )
	{
		if( device_id == gLogicId )
		{
			std::cout << "A Logic device was disconnceted (id=0x" << std::hex << 
	device_id << std::dec << ")." << std::endl;
			gLogicInterface = NULL;
		}
	}
```

OnDisconnect provides us with the *device_id* of the Saleae device that has been disconnected. Note that once this function ends, the underlying *LogicInteface* object will be destroyed soon thereafter, so this is another case where thread safety is important. For our non-mission-critical little demo however, we'll be fine.

We simply set our global pointer to *NULL* so that code elsewhere can tell that they can't access the device any more.

ReadByte and WriteByte

Reading a single byte from Logic or having it output a byte (on its pins) is easiest thing you can do with the SDK.

```
	if( command == "readbyte" || command == "rb" )
	{
		std::cout << "Got value 0x" << std::hex << U32( gLogicInterface->GetInput() ) << 
	std::dec << std::endl;
```

```
        }
        else
        if( command == "writebyte" || command == "wb" )
        {
                static U8 write_val = 0;

                gLogicInterface->SetOutput( write_val );
                std::cout << "Logic is now outputting 0x" << std::hex << U32( write_val ) <<
        std::dec << std::endl;
                write_val++;
        }
```

By default Logic's pins are inputs (high-z). As soon as you issue a write command, they change to outputs. If you subsequently read, they immediately change back to inputs. There isn't any way to read on some pins and write on others unless you had two Logics connected, and were writing on one and reading on the other exclusively.

Reading or writing a byte in this way is extraordinarily slow compared to streaming, which is what Logic is primarily designed to do. However, it might be all you need depending on your application.

Please note that Logic has series 510-ohm resistors on its input pins, so output mode is limited to very low currents. You may need to buffer the outputs to do things like drive LEDs.

Reading and Writing, and the OnReadData and OnWriteData Callbacks

What Logic is primarily designed to do is to read or write high-bandwidth streams. When reading, the *OnReadData* callback is called with an array of data. You can expect this callback to be called at on the order of 20Hz. The amount of data included in each callback will depend on the sample rate. The higher the sample rate, the more data will arrive per callback. *OnReadData* is called sequentially – in the order data is sampled. Data near the beginning of the array was collected before data later in the array. *OnReadData* will not be called again until you return from the previous *OnReadData* callback, so you should be careful not to do too much work in *OnReadData* – spending too much time in *OnReadData* will cause the collection system to back up and ultimately *OnError* will be called when it overflows. If you need to do heavy lifting, consider holding on to the data's pointer, and providing it to a worker thread in a thread-safe manner. You could also keep the data, and process it after you stop the read process.

OnReadData provides the *device_id* of the device the data is coming from, a pointer to an array of bytes, and the size of that array. You own the data, and are responsible for eventually freeing it, although this could be much later.

To free the memory, use *DevicesManagerInterface::DeleteU8ArrayPtr.* This is because depending on the operating system, it's likely that memory was allocated on a different heap, and not one that you have direct access to.

```
void OnReadData( U64 device_id, U8* data, U32 data_length, void* user_data )
{
        std::cout << "Read " << data_length << " bytes, starting with 0x" << std::hex <<
(int)*data << std::dec << std::endl;

        DevicesManagerInterface::DeleteU8ArrayPtr( data );
}
```

OnWriteData is similar to *OnReadData*, only instead of providing you with data, it's asking for you to provide it with data. The data itself is already allocated, you just to fill the contents. Note that as soon you start the writing process, *OnWriteData* will be immediately called a couple dozen times, to make sure the USB doesn't get ahead of us and run out of data before we can provide it with more. As with *OnReadData*, don't spend too much time in *OnWriteData*. Note also that due to limitations with WinUSB on Windows, you probably won't be able to output data at more than 4MHz. Other platforms don't have this limitation, and should be able to write just as fast as they can read (24MHz).

```
void OnWriteData( U64 device_id, U8* data, U32 data_length, void* user_data )
{
        static U8 dat = 0;

        for( U32 i=0; i<data_length; i++ )
        {
                *data = dat;
                dat++;
                data++;
        }

        std::cout << "Wrote " << data_length << " bytes of data." << std::endl;
}
```

To start reading or writing, call *StartRead*, or *StartWrite*. This assumes you've already registered the appropriate callback functions.

OnError

A lot of SDK users are looking to set up some sort of automated test or data logger. Unfortunately Logic is better suited to being a Logic analyzer, because it has a very limited hardware-side buffer, and can easily overflow if the USB isn't constantly pulling data from it. Don't count on a capturing data for many hours or days without interruption, even at slow sample rates. Your application must allow for the possibility that a capture could fail at any time. When a capture fails, you won't get bad data. All the data that arrives at *OnReadData* is sequential and has no gaps. When an error happens, you'll stop getting *OnReadData* callbacks and *OnError* will be called. Once *OnError* is called, you won't get any more *OnReadData* calls.

What you can do – if your application is tolerant enough – is to simply handle this condition: make a note that there was a break in the data at such and such a time, and start up the read again and continue on.

However – you can’t restart the data collection from inside the OnError callback. You’ll need to do this from a different thread (typically you would do this from your main thread).

C#.NET Support

In 1.1.8 we've added .NET support for Logic and Logic16. Inside the *SaleaeDeviceSdk-1.1.x* folder, there will be a C#.NET folder. In here you will find the *ConsoleDemo* project. You should be able to build and run this project right from its current location.

A few pointers for creating .NET projects

First, you'll need to reference the *SaleaeDeviceSdkDotNet.dll* assembly. This is in *lib\C#.NET.*

You'll also need the *SaleaeDevice.dll* – the native dll that you would use if you were using c++. Like the .NET dll above, this will ultimately need to end up in the same folder as your exe. The easiest way to do this is add it to your project: In the Solution Explorer, right click on your project and select Add->Existing Item. In the file types drop-down, select *.*, and then navigate to and select the *SaleaeDevice.dll* (in the *lib* folder). Once added to your project, set the item's Copy to Output Directory property to Copy Always.

The last thing to mention is you can't restart a capture from inside the *OnError* callback. Instead, this needs to be done from the main thread. You can use .NET's *Invoke* functionally to do this.