Just wanted to document the steps I had to take to get the ePrint and Linea SDK’s playing nicely in the same project for future reference if needed.  The original error we were receiving when immediately trying to launch the app is “[Printer sharedDevice]: unrecognized selector sent to class”, which is called in the SurveyAppDelegate class as soon as the application starts.

The root cause of the issue is both the ePrint and Linea SDK libraries have classes with the same object name, causing a conflict when the application is compiled.  With Objective-C, when compiling the application (including any imported libraries), a symbol is created for each class.  That class name is used to distinguish between them at runtime, so if you have any class names that are the same (even in an imported library/framework), you can run into this issue.

Originally the thought was to contact Linea and have them rename their class name to something unique to get rid of the conflict, but I was able to find steps online to perform this step manually on an already compiled static library.  This issue is something that happens enough that there are utilities already out there to handle this for x86 architectures, but nothing was ever created for ARM for some reason.  Because of this, I ended up having to find a solution for the two different architecture types.  Steps below handle splitting the “libdtdev.a” file into its respective architecture types, renaming the conflicting class, then re-compiling both a separate Simulator and Device library for XCode.

* **Split “libdtdev.a” file into its respective build type libraries (convert the “fat” library into it’s respective “slim” libraries)**
  1. In Mac OS X, copy the “libdtdev.a” file to a working folder by itself, then navigate into this folder from a terminal
     + Note: I ended up just using the Downloads folder, so the following command will take you to this:
       - “cd /Users/[username]/Downloads”
  2. Run the following command in the Terminal in that folder:
     + “lipo libdtdev.a –info”
  3. That command will spit out the included architectures in the “fat” library
  4. For each architecture type provided, run the following command, replacing “[arch]” with each Architecture included
     + “lipo libdtdev.a –thin [arch] –output libdtdev.[arch]”
  5. You should now have the original “libdtdev.a” library file, as well as 4 or 5 new files generated for each architecture
  6. We will be modifying these separate library files based on if it’s an ARM architecture (armv7/armv7s/arm64/etc.) or if it’s an x86 architecture (i386/x86\_64/etc.)
* **Modify ARM architecture library files (for the Device)**
  1. ARM architecture does not currently have a nice utility for renaming classes in a static library, so we have to manually do this using a HEX Editor
     + I ended up using this HEX Editor for Mac OS X: <http://ridiculousfish.com/hexfiend/>
  2. Open your HEX Editor, and open the “Find/Replace” utility and set it to “Text” mode
  3. Do a search for each of the following items, and do a “Replace All” with the updated text:
     + Find: “\_Printer” | Replace: “\_LinPrtr”
       - This takes care of a number of “symbol definitions” such as “\_OBJC\_CLASS\_$\_Printer” / “\_OBJC\_METACLASS\_$\_Printer” / “l\_OBJC\_CLASS\_RO\_$\_Printer” / etc.
     + Find: “[Printer ” | Replace: “[LinPrtr ”
     + **Please Note: When renaming the class this way, you need to make sure the number of characters match what you’re replacing it with.  So if the original Class Name is 10 characters long, make sure the new one is also exactly 10 characters long.**
  4. Make sure to do this for each different ARM build, and then save the file once you’re done replacing.
  5. Once completed, follow the steps to re-combine each architecture into a “fat” library
* **Modify x86 architecture library files (for the Simulator)**
  1. For this one, I ended up finding a utility in Windows to perform this action.
     + Download the “Object file converter” utility from: <http://www.agner.org/optimize/#objconv>
     + There are also utilities out there to install the linux utility “objcopy” into Mac OS X, but decided this port for Windows was an easier/quicker solution.
  2. Extract that downloaded utility anywhere and open a command window in that folder (Shift + Right Click -> “Open Command Window Here”)
  3. Also copy the “libdtdev.i386” and “libdtdev.x86\_64” files into this same directory.
  4. Run the following commands in the order provided for each individual architecture file:
     + For each command, replace [arch] with the architecture file we’re modifying.  Make sure to do one architecture file at a time, since each command is dependent on the one in front of it.
     + Command: “objconv –nr:\_OBJC\_CLASS\_$\_Printer:\_OBJC\_CLASS\_$\_LinPrtr libdtdev.[arch] libdtdev.[arch].new”
     + Command: “objconv –nr:\_OBJC\_METACLASS\_$\_Printer:\_OBJC\_METACLASS\_$\_LinPrtr libdtdev.[arch].new libdtdev.[arch].new2”
  5. At the end of it, you should have a “.new” and “.new2” file.  Go ahead and delete the “.new” and original file, and remove “.new2” at the end of the last one.  The very final file is the one you want to keep.
  6. Once completed, follow the steps to re-combine each architecture into a “fat” library
* **Re-Combine different Architecture files back into a “fat” (.a) library (make it usable for XCode)**
  1. These steps can be used to re-combine separate architecture files into either a “simulator” or “device” .a library file.  From the tests I ran, it seems best to split the architectures up due to the different way we went about renaming the conflicting class.  I was receiving strange errors if I used a completely combined one.
  2. Determine what architecture files to include, then run the following command
     + Replace [arch1] / [arch2] / [arch…] with the different architecture types.  You can include as many as you have.
     + Replace [device or sim] with the library type this is, so that it is either “libdtdev-device.a” or “libdtdev-sim.a”.
     + Command: “lipo libdtdev.[arch1] libdtdev.[arch2] libdtdev.[arch…] –create –output libdtdev-[device or sim].a”
  3. And that’s it!  The filename you passed in for the “-output” flag is the file to import back into the XCode project.

That’s pretty much it.  The only other difference was since it’s using a separate library file for the Simulator and device, you have to handle passing in the simulator file into the “Other Linker Flags” property for Debug only.