The Pixy2

Relevant Documentation:

- *PixyMon Installation*
- *Pixy2 Porting Guide*
- *Building libpixy2 on Linux*

The Pixy2 (herein referred to as the Pixy) can be connected to a computer over USB. PixyMon is the official software for using the Pixy with a computer, with support for displaying video feed, and training it. The installer comes with drivers to use it.

Initializing the Pixy with a computer running PixyMon (although PixyMon does not need to be running for the Pixy to receive power) goes as follows:

- Plug the Pixy in.
- In the "Action" menu, enable the Pixy lamp. This will improve the recognition of colors.

The process for teaching the Pixy an object goes as follows:

- Start holding down the button. The LED will start cycling through various different colors. Each color represents a color signature. Release the button to select a color signature that will be written in to.
  - The exception to this is when the LED is white. If this is selected, the Pixy will enter automatic white balance mode.
- The Pixy enters "light pipe" mode, and will try to lock in on an object in the center of its field of view. When it does so, it will create a statistical region growing model of the object, and the LED will change depending on the color of the object.
  - The stronger the LED, the better the focus.
- When the Pixy has a good hold on the object, press the button once more, and it will save the color signature to the selected slot.
- In Pixymon, fine tune the signature and give it a name.

The Pixy2 has support for talking to a number of other controllers, over serial, analog/digital, or USB interfaces. The generic driver for USB, and Linux systems, is libpixy2. This is intended to be ported over to other microsystems, such as the roboRIO.

libpixy2

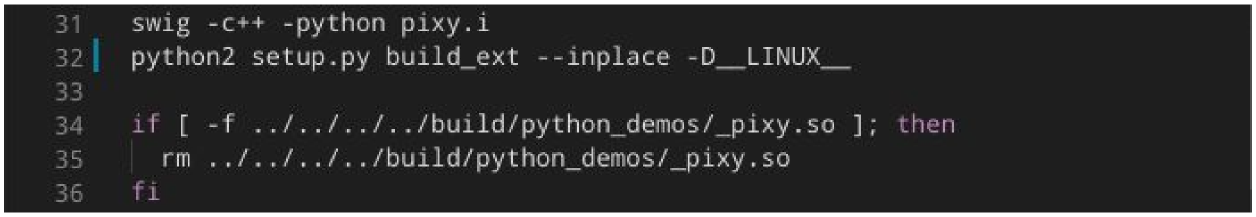
libpixy2 is written in C++, and such that is how its bindings are exposed. Therefore, to use it in Java robot code that will be running on the roboRIO, it is necessary to compile libpixy2 as a shared library, and link it to our Java code. Then, we must make our own

Java library that calls the functions exposed by libpixy2. Finally, our robot code will make use of our Java library. PixyUSB by Team 103 is an example that implements this method, using the original libpixyusb.

Charmed Labs provides in the official Pixy2 repository an example of using libpixy2 with Python. It does this by using SWIG, the Simplified Wrapper and Interface Generator. I was pretty excited upon making this discovery, because it means that we really will get to work with C++ in Java while learning a new tool.

I plan on reading through the SWIG documentation, but looking at the Python setup they have going, there are some things that can be assumed.

Educated Guesses
In regards to using this setup as to make some educated guesses about SWIG, the build script is the jackpot. I had to make a slight modification to it to make it run, making it use Python 2, but aside from that everything works out of the box.



```
swig -c++ -python pixy.i
python2 setup.py build_ext --inplace -D__LINUX__

if [ -f ../../../../build/python_demos/_pixy.so ]; then
  rm ../../../../build/python_demos/_pixy.so
fi
```

SWIG is called, with C++ and Python parameters passed in, as well as a "pixy.i" file. This input file appears to use a template language, with underlying syntax most similar to C++. Upon further inspection, though, aside from templated parts, this is actually a very normal C header, defining external functions. These function signatures match those of pixy_python_interface.cpp, suggesting that that file is the implementation of the functions. This file includes the libpixy2 header, and uses the *Pixy2* object inside it. Taking a peek at the output files, a "pixy.py" and "pixy_wrap.cxx" have been produced by a meta compiler. Or transpiler. Or whatever the buzz word is.

The CXX file contains the header-like part of the input file in its entirety among many, many other lines of code used for the C++ -> Python interface I assume. The Python file contains Python functions for each of the C++ functions that have been defined. **Suffice to say, the Pixy Python module and the Pixy C++ wrap have analogous definitions of functions.**

Accepting why the code works the way it does as black magic, we can now move onto the next, Python part of this process. I don't know how setup scripts work, but keeping everything previously discussed in mind, it can be figured out.

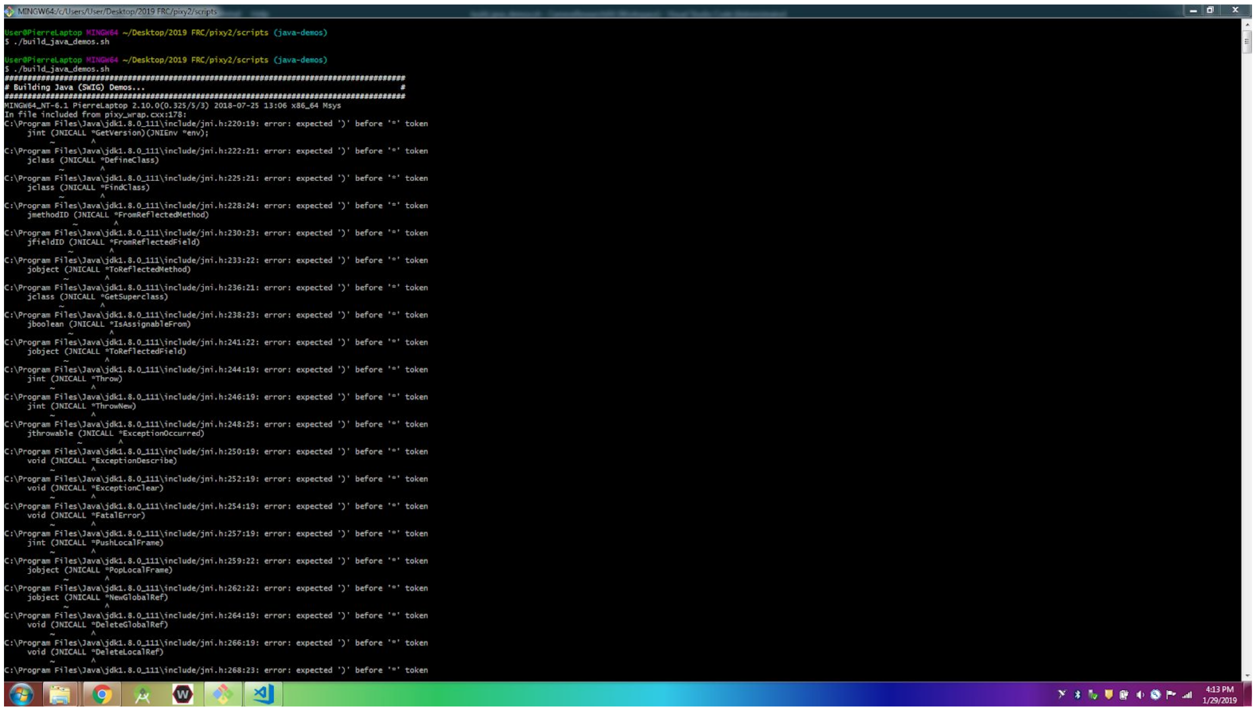
According to the description fields, this setup script builds a libpixy2 Python module. The extension (I'm not 100% if this is synonymous with module.) has a name of "_pixy". Sure enough, after going through the build process, we get a "_pixy.so" shared object. This script also requires the CXX file as a dependency. Currently, what I think is the most probably is that this script compiles the library, and links it to the C++ interface, which is compiled as the shared object, and somehow that's used in the Python script.

After looking around some more, that hypothesis appears to be not 100% accurate. The case actually seems to be that the Python module is not involved in the SWIG step of the build process. Instead, this is involved at runtime. The Pixy Python invokes SWIG, which in turn loads the "_pixy" shared object. The shared object is loaded up with the C++ interface, so now the Python module can use the C++ code.

Swig with Java

With this preliminary research done, I'm going to read through SWIG documentation, and begin prototyping a Java implementation of this all.

I began by making a copy of the Python build script, and made a "java_demos" directory. I copied over the "pixy.i' file, because I have a feeling we're going to be able to reuse most of, if not all of it for our Java purposes. I ran SWIG, rigged the ARM GCC compiler up to the Java include directories, libusb include directory, and the local Pixy libraries, and got a nightmare of error codes.



```
MINGW64:/c/Users/User/Desktop/2019 FRC/pixy2/scripts

User@PierreLaptop MINGW64 ~/Desktop/2019 FRC/pixy2/scripts (java-demos)
$ ./build_java_demos.sh

User@PierreLaptop MINGW64 ~/Desktop/2019 FRC/pixy2/scripts (java-demos)
$ ./build_java_demos.sh
################################################################################
# Building Java (SWIG) Demos...                                                #
################################################################################
MINGW64_NT-6.1 PierreLaptop 2.10.0(0.325/5/3) 2018-07-25 13:06 x86_64 Msys
In file included from pixy_wrap.cxx:178:
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:220:19: error: expected ')' before '*' token
     jint (JNICALL *GetVersion)(JNIEnv *env);
         ~        ^
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:222:21: error: expected ')' before '*' token
     jclass (JNICALL *DefineClass)
           ~        ^
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:225:21: error: expected ')' before '*' token
     jclass (JNICALL *FindClass)
           ~        ^
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:228:24: error: expected ')' before '*' token
     jmethodID (JNICALL *FromReflectedMethod)
              ~        ^
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:230:23: error: expected ')' before '*' token
     jfieldID (JNICALL *FromReflectedField)
             ~        ^
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:233:22: error: expected ')' before '*' token
     jobject (JNICALL *ToReflectedMethod)
            ~        ^
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:236:21: error: expected ')' before '*' token
     jclass (JNICALL *GetSuperclass)
           ~        ^
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:238:23: error: expected ')' before '*' token
     jboolean (JNICALL *IsAssignableFrom)
             ~        ^
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:241:22: error: expected ')' before '*' token
     jobject (JNICALL *ToReflectedField)
            ~        ^
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:244:19: error: expected ')' before '*' token
     jint (JNICALL *Throw)
         ~        ^
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:246:19: error: expected ')' before '*' token
     jint (JNICALL *ThrowNew)
         ~        ^
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:248:25: error: expected ')' before '*' token
     jthrowable (JNICALL *ExceptionOccurred)
               ~        ^
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:250:19: error: expected ')' before '*' token
     void (JNICALL *ExceptionDescribe)
         ~        ^
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:252:19: error: expected ')' before '*' token
     void (JNICALL *ExceptionClear)
         ~        ^
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:254:19: error: expected ')' before '*' token
     void (JNICALL *FatalError)
         ~        ^
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:257:19: error: expected ')' before '*' token
     jint (JNICALL *PushLocalFrame)
         ~        ^
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:259:22: error: expected ')' before '*' token
     jobject (JNICALL *PopLocalFrame)
            ~        ^
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:262:22: error: expected ')' before '*' token
     jobject (JNICALL *NewGlobalRef)
            ~        ^
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:264:19: error: expected ')' before '*' token
     void (JNICALL *DeleteGlobalRef)
         ~        ^
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:266:19: error: expected ')' before '*' token
     void (JNICALL *DeleteLocalRef)
         ~        ^
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:268:23: error: expected ')' before '*' token
```

The first of which is:

```
C:\Program Files\Java\jdk1.8.0_111\include/jni.h:220:19: error:
expected ')' before '*' token
     jint (JNICALL *GetVersion)(JNIEnv *env);
```

This would indicate that something is wrong with our JDK, because it's unable to find a "JNICALL" type, and that's just when compiling the JNI headers, before it gets to our code. As a sidenote, the compiler is one I found on my own time, on the ARM website. However, I made the discovery that this part isn't necessary, as a version of arm-none-eabi-gcc is provided with WPILib.

To get to the bottom of what's going on, I decided I would try making a desktop build. I installed Cygwin, and with it the Mingw-64 GCC compiler. I changed the compiler executable, popped the script into Cygwin (after messing with line endings), and it produced a "pixy_wrap.o" file. To get a better idea of what to expect, now I'm going to try making a Java demo that will run on PC.

I've now made a clone of the Python build script, for the Java SWIG library. It successfully runs SWIG, G++, and finally javac. This works pretty well, and on a Unix setup it succeeds at compiling the Java classes. However, the Java class can't access the bindings to C++, and since this side of the bindings are written in Java, now might be time to setup a proper build system. I tried many ways to get this working with javac alone, but that hasn't worked, so I'm going to learn Gradle, the choice of FIRST, and libraries used by it.

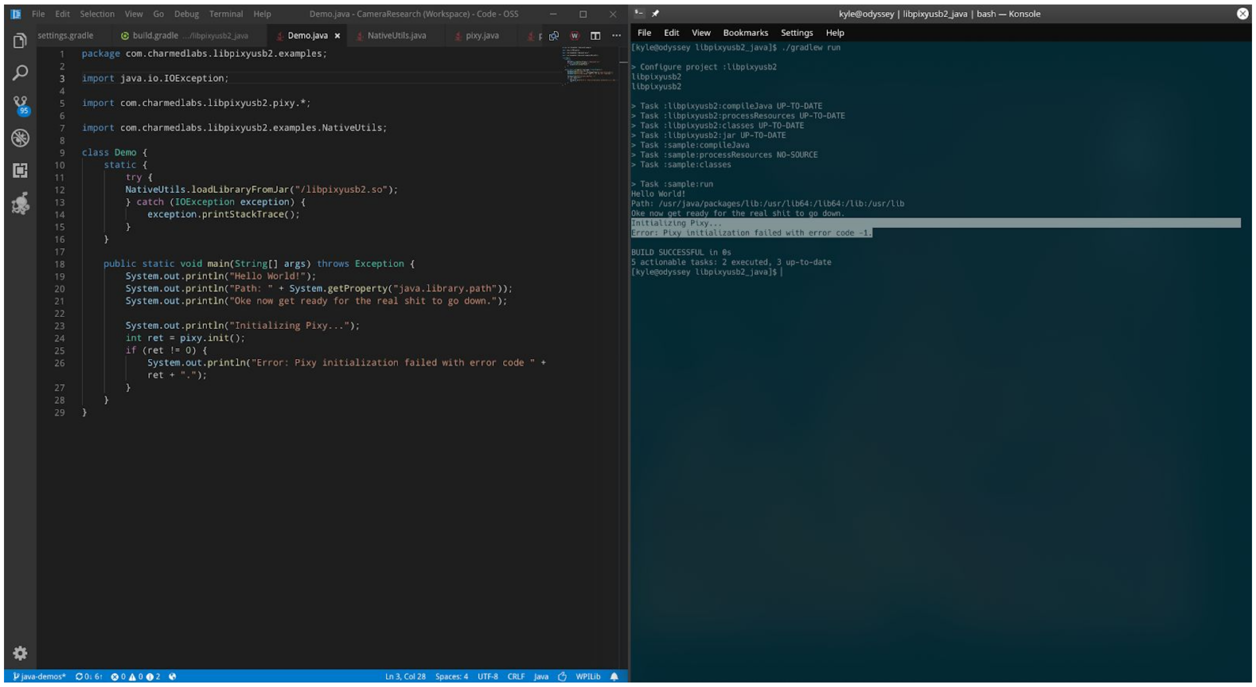
Breakthroughs

It's now a little over a week later. Since the last update here, I've made the discovery that another FRC team has found a way to make the Pixy work with the RoboRIO, here. An honorable mention is this repo, as well. These repos are ports of the C++ Arduino library, to Java. Upon seeing this, I thought that this was a wonderful idea, and was annoyed I hadn't thought of it myself. I looked around the former repo more, and found that, as opposed to working over USB, it utilizes the other ports. Therefore, it does not support a raw video feed, which we really want. With a little help, the USB can do this.

Moreover, we do not have a serial cable, or any other cable we could use with this library. Therefore, the method I picked out might be going to good use after all.

Over the past week, I have been doing additional maintenance and installing toolchains. Some of the discoveries I came across were that MinGW should be preferred over MinGW-64 here, because the latter is not supported by Gradle. Additionally, conveniently enough, an ARM toolchain is provided with WPILip. On the note of Gradle, I have been reading much, much documentation. Gradle is really hard to learn in such a short amount of time, but I know enough that today I was able to make a build system that successfully uses the shared object to call a libpixyusb2 method from a Java class.

The importance of this is that making this setup work with *javac* alone is very difficult, and at the end of the day, it will have to be integrated with the Gradle build system of the robot anyways. Gradle is *hard* though, the syntax feels oddly familiar for a language with its roots in Java.


Demo.java - CameraResearch (Workspace) - Code - OSS
settings.gradle
build.gradle .../libpixyusb2_java
Demo.java
NativeUtils.java
pixy.java
kyle@odyssey | libpixyusb2_java | bash — Konsole

```java
package com.charmedlabs.libpixyusb2.examples;

import java.io.IOException;

import com.charmedlabs.libpixyusb2.pixy.*;

import com.charmedlabs.libpixyusb2.examples.NativeUtils;

class Demo {
    static {
        try {
        NativeUtils.loadLibraryFromJar("/libpixyusb2.so");
        } catch (IOException exception) {
            exception.printStackTrace();
        }
    }

    public static void main(String[] args) throws Exception {
        System.out.println("Hello World!");
        System.out.println("Path: " + System.getProperty("java.library.path"));
        System.out.println("Oke now get ready for the real shit to go down.");

        System.out.println("Initializing Pixy...");
        int ret = pixy.init();
        if (ret != 0) {
            System.out.println("Error: Pixy initialization failed with error code " +
            ret + ".");
        }
    }
}
```

```
[kyle@odyssey libpixyusb2_java]$ ./gradlew run

> Configure project :libpixyusb2
libpixyusb2
libpixyusb2

> Task :libpixyusb2:compileJava UP-TO-DATE
> Task :libpixyusb2:processResources UP-TO-DATE
> Task :libpixyusb2:classes UP-TO-DATE
> Task :libpixyusb2:jar UP-TO-DATE
> Task :sample:compileJava
> Task :sample:processResources NO-SOURCE
> Task :sample:classes

> Task :sample:run
Hello World!
Path: /usr/java/packages/lib:/usr/lib64:/lib64:/lib:/usr/lib
Oke now get ready for the real shit to go down.
Initializing Pixy...
Error: Pixy initialization failed with error code -1.

BUILD SUCCESSFUL in 0s
5 actionable tasks: 2 executed, 3 up-to-date
[kyle@odyssey libpixyusb2_java]$ 
```

Pictured here is the first successfully running demo that I captured in action, a very big milestone in this project. From here, there is still a lot that has to be done though:

- Cross compile to ARM.
- Integrate with GradleRIO (Currently, this is tied into the *pixy2* repo.)
- Merge in raw video support.
- Make a *VideoSource* class for the Pixy.
- Expose the video to the *SmartDashboard*.
- Add swig integration to the build system.

For compiling libusb for ARM, I found the following to be useful:
/configure --host arm-frc2019-linux-gnueabi --enable-udev=no --enable-shared