Hacking the KIPR Link: Using the Pixy Low-Latency USB Color Camera
Jeremy Rand
Team SNARC (Sooners / Norman Advanced Robotics Coalition)
jeremy@veclabs.net

Hacking the KIPR Link: Using the Pixy Low-Latency USB Color Camera

1 Introduction

The color cameras that have been used in Botball since 2009 are reasonably high quality, but they have two significant problems: they have visible latency (lag), and they use significant CPU time to perform tracking. These problems make them unsuitable for some applications. Botballers may find themselves wishing for a camera that doesn't have these drawbacks. However, there is a solution already in existence, which has just been waiting for a hacker to integrate and document.

The Pixy [1] is a color camera designed for blob tracking with extremely low latency. Instead of making the controller's CPU perform the computations, the Pixy has an on-board processor which handles color tracking, and outputs blob data in the form of numerical coordinates. The Pixy is capable of handling huge numbers of blobs simultaneously at 50 frames per second, has incredibly low lag, and can also detect multi-colored blobs with orientation. While the Pixy can output in a number of formats, including as standard analog and digital sensors, it is most flexible and user-friendly when plugged into a USB port.

The Pixy is engineered by Carnegie-Mellon University and Charmed Labs. Charmed Labs was the developer of the Xport Botball Controller (XBC), which first introduced low-latency color tracking to Botball in 2005 and has occupied the position of “favorite Botball controller of all time” of a number of long-time Botballers.

WARNING: The instructions contained in this paper involve hacking your Link using SSH and replacing critical system files. If you do this wrong, your Link **will** be bricked. We bricked a Link a couple times while fine-tuning this research, and while we were able to recover by reflashing a stock firmware (and expect you will be able to do the same), we offer no warranty on these instructions and offer them at your own risk. Don't expect us or KIPR to fix your Link if your Link gets bricked or displays ultra-realistic smoke effects, and don't use this in a competition robot unless you're willing to risk a failed run when the instructions contained in this paper fail to work as intended. If this concerns you, that should be a hint that hacking the Link is not for you.

2 Prerequisites

We're going to assume that you know how to use a command line on Linux (including SSH), and that you are reasonably skilled at coding in C (with a little C++). We're also going to assume that you are capable of reading the Pixy manual for things like setting color models (which is not best done on the Link). If any of these don't apply to you, then you may want to rethink your plans to hack the Link to use the Pixy.

3 Getting the Pixy Source Code

The Pixy libraries from Charmed Labs can be obtained from the Charmed Labs GitHub [2]. We used the revision from June 13, 2015 [3]; other versions might have differences. You can use the "Download ZIP" button at that link to get the code. The contents of the Git repository should be placed on a USB flash drive in a folder called "`pixy-master`".

Additionally, you'll need to add two files to the Pixy code.

The first file is at `pixy-master/src/host/hello_pixy/build_snarc.sh`, and has the following contents (all one line):

```
g++ -fpic -c -D__LINUX__ -D__LIBPIXY_VERSION__=`cat
../libpixyusb/cmake/VERSION` -I/usr/include/libusb-1.0
-I../libpixyusb/include -I../../common -I../libpixyusb/src
-I../libpixyusb/src/utils
../libpixyusb/src/chirpreceiver.cpp
../libpixyusb/src/pixy.cpp
../libpixyusb/src/pixyinterpreter.cpp
../libpixyusb/src/usblink.cpp
../libpixyusb/src/utils/timer.cpp ../../common/chirp.cpp
```

The second file is at `pixy-master/src/host/hello_pixy/build_snarc_main.sh`, and has the following contents:

```
echo "Building program..."

mkdir -p /kovan/snarc/pixy_main

g++ -D__LINUX__ -D__LIBPIXY_VERSION__=`cat
../libpixyusb/cmake/VERSION` -I/usr/include/libusb-1.0
-I../libpixyusb/include -I../../common -I. -include
kovan/kovan.hpp -include stdio.h -include math.h hello_pixy.cpp
chirpreceiver.o pixy.o pixyinterpreter.o usblink.o timer.o
chirp.o /usr/lib/libstdc++.so.6.0.17 -lusb-1.0 -lboost_thread
-lboost_system -lkovan -lm -lpthread
/usr/lib/libboost_chrono.so.1.51.0 -o
/kovan/snarc/pixy_main/pixy_main

sync
sync

echo "Program built."
```

4 Pixy Compatibility with the Link's Architecture

The Pixy libraries use the Boost library [4] as a dependency. Unfortunately, the Link does not include Boost by default. The Link does have a package manager, which one might think would be helpful here (that's how we hacked the Link to use the Xtion depth sensor back in 2013 [5]).

However, in this case, there are serious problems.

The big problem is that the package repositories that the Link uses do not actually have the Boost libraries we need. Or rather, they *did* have those libraries, at different times throughout history, but for some reason they are not in the current repository contents. Luckily, the older versions of the repository contents are still available if you trick the Link into asking for them. Not so luckily, doing this requires replacing a system file. Less luckily still, there was no time period when *all* of the needed libraries were present at once, so we need to mix and match time periods. Still decreasing in luckiness, the Pixy libraries need different sets of these time periods to be present depending on whether you're compiling the Pixy library or compiling your robot program. And least lucky of all, trying to do this chimeric mixture of package versions is guaranteed to damage some other parts of the Link's filesystem, meaning you will need to manually reverse the damage yourself before the next time you reboot, otherwise you may have a bricked Link after a reboot.

5 Building the Library

To start out, make sure your Link has a full battery and is connected to a charger, connect the USB flash drive that contains the Pixy libraries to the Link, make sure that your Link is connected to WiFi, and check its IP address. Open up an SSH prompt to the Link, and run the following commands:

```
cd /etc/opkg

echo "src/gz base http://feeds.angstrom-
distribution.org/feeds/v2013.06/ipk/eglibc/armv5te/base">base-
feed.conf

sync
sync
```

What we've just done is trick the Link into retrieving packages from the June 2013 version of the package repositories.

Now run the following over SSH:

```
opkg update
opkg --force-overwrite install boost-dev
sync
sync
```

This will install the half of the Boost libraries that are needed for building the Pixy library. However, in the process of running this operation, the package manager has changed the standard C++ library version in such a way that the standard C++ library is bricked. We don't know what would happen if it were left like this, but it would certainly not be good, and the Link might become unbootable. To rectify this, do the following over SSH:

```
ln -s -f /usr/lib/libstdc++.so.6.0.17 /usr/lib/libstdc++.so
sync
sync
```

This will fix the system's ability to find the standard C++ library. Note that there might be other, less-used components that are also bricked; we haven't run into any, but if you have problems with libraries not being findable, you can use the same type of `ln` command to fix it.

Now log out of SSH and then reboot the Link. If it fails to boot for some reason, something has gone horribly wrong, and you should reflash from an official firmware and start over. (Or ask on the Botball Community [8] for help.)

Reconnect to SSH, and run the following commands:

```
cd /kovan/media/sda1/pixy-master/src/host/hello_pixy
./build_snarc.sh
sync
sync
```

The build script may take approximately 5 minutes to run. You may notice that the `build_snarc.sh` file that we've added isn't the official way to build; the official way to build is to use the "`cmake .`", "`make`", and "`sudo make install`" commands. However, the official method requires using the CMake framework [6], and while the Link does support CMake, CMake gets highly confused by the nonstandard Boost library configuration that we're using, and throws an error. While it may be possible to work around this with some command line arguments, we found that it is much easier to write a custom build script.

We also noticed something very odd regarding the Boost packages installed in this step. When we were initially figuring these steps out in summer 2014, the `boost-dev` package was simply not present in the latest repos, which is why we used the June 2013 version. Sometime between then and summer 2015 when we submitted this paper, that package was re-added to the latest repos. But... installing the version that was re-added to the latest repos causes the C++ compiler to stall for about 30 minutes and then crash while building the Pixy libraries. The version from June 2013 still works fine. We have no idea what is broken there, but since there's a workaround, we haven't concerned ourselves with it.

When these commands have finished, the Pixy libraries are installed.

6 Installing the Remaining Boost Libraries

Now we have to set up the rest of the Boost libraries, which are used by Pixy programs. In SSH, run the following:

```
cd /etc/opkg

echo "src/gz base http://feeds.angstrom-
distribution.org/feeds/core/ipk/eglibc/armv5te/base">base-
feed.conf

sync
sync
```

This restores the package manager's ability to download current packages, rather than the ones from June 2013. Now run the following commands over SSH.

```
opkg update
opkg install libboost-chrono1.51.0
sync
sync
```

This will install the other half of the Boost libraries.

Reboot your Link again. If it boots, then everything went fine and you're ready to compile a Pixy program. If it fails to boot for some reason, something has gone horribly wrong, and you should reflash from an official firmware and start over. (Or ask on the Botball Community [8] for help.)

7 Building a Pixy Program

Finally, all that remains is to build the Pixy program.

Place the following two files on the root of your USB flash drive:

The first file is named `BuildPixyMain.c`, and has these contents:

```
int main()
{
     printf("Launching build script\n");

     chdir("/kovan/media/sda1/pixy-master/src/host/hello_pixy/");
     system("./build_snarc_main.sh");
     chdir("/kovan/media/sda1/");

     printf("Build script complete\n");
}
```

The second file is named `PixyBootloader.c`, and has these contents:

```
#include <unistd.h>


int main()
{
     printf("Launching Pixy main...\n");
     execl("/kovan/snarc/pixy_main/pixy_main",
"/kovan/snarc/pixy_main/pixy_main", (char*) 0);
}
```

Now compile and execute `BuildPixyMain.c` from the standard Link GUI. You'll notice that this C program simply runs a build script that we had previously placed on the flash drive. The build script `#define`s some constants that normally the CMake framework (which we bypassed) would have provided to the Pixy code, includes and links with all of the relevant libraries (including libusb, libkovan, Boost, and the Pixy libraries), and places a binary program in the Link's filesystem.

If any errors occurred, they will be listed. It will probably show a warning about a Boost version conflict; this is because of the chimeric Boost installation we've performed, and can be safely

ignored. If you don't see “Building program” and “Program built”, then the build scripts failed to even start, meaning something is wrong.

Now plug the Pixy into the Link's USB port, and compile and execute `PixyBootloader.c` from the standard Link GUI. It should display the Pixy's library and firmware version (for us the library version was corrupted, but it didn't cause any problems). Try waving a brightly colored object in front of it – if it's a color that the Pixy is configured to recognize, it will display information about the blob in real time.

You may have noticed an oddity: the first .c program we ran uses the `system` command to run an external program, while the second one uses `execl`. Why is this? `system` allows the original program to continue when the called program finishes, but the called program runs with its own process ID (pid). `execl` forces the original program to exit, with the called program taking over the pid of the original program. Why does the pid matter? It matters because the emergency-stop button in the Link GUI kills the pid of the C program that was called by the Link GUI. This means that if you use `system` to run your Botball program, the e-stop button won't have any effect. Even worse, if you run your Botball program a second time, you'll have two copies of your Botball game running at once, which won't end well (especially since libkovan isn't thread-safe). Hence, we use `execl` for all use cases that involve a Botball game program.

8 Writing Your Own Program

To write your own program, just insert your own code into `pixy-master/src/host/hello_pixy/hello_pixy.cpp` on the flash drive, insert the flash drive into the Link, and rerun the `BuildPixyMain.c` and `PixyBootloader.c` programs. A huge advantage of this method is that you don't need to use SSH or WiFi once the library is installed; everything is done with a flash drive and the Link GUI. Unfortunately, the setup we've created only allows one Pixy program to be loaded on the flash drive at once.

The provided code already shows some sample code for accessing the color blob data. For more detailed documentation, the libpixyusb documentation [9] is a great resource. To train color models, it is recommended to hook up the Pixy to a PC and use the “pixymon” tool provided by Charmed Labs. Documentation on using pixymon to create color models, as well as other information about the Pixy, is at the Pixy wiki [7].

Since libkovan is linked into your program, you're free to write whatever Botball code you like that utilizes the Pixy data. Note that `printf` will be subject to a large buffer, so information that you print will not display immediately. To flush the buffer, you can run:

```
fflush(stdout);
```

It should also be noted that you are writing in C++. Since C++ is mostly a superset of C, this will be familiar to C programmers. However, if you wish to use the libkovan C library (which is what the Botball workshops teach and most teams use), you should have this at the top of your C++ program:

```
#include <kovan/kovan.h>
```

9 Why We're Using USB

Some of you may have wondered why we're using USB to connect to the Pixy, when it already supports digital, analog, and serial connections (which the Link also supports). The main reason was that doing it this way is easier from a hardware level (power is already safely provided, no custom cables need to be soldered), and we're more comfortable with software hacking than hardware hacking. Also, the digital/analog interfaces don't deliver enough information for our use cases, and serial would have eaten up the Create port. (Unless there's another serial port on the Link that we don't know about, which is entirely possible.)

10 Conclusion

We hope you've enjoyed reading about our latest hacking efforts on making the Link more awesome using the Pixy. We'd love to hear what you do with the Link and the Pixy. Please tell us on the Botball Community [8]!

Happy Tracking!

11 References

[1] Charmed Labs. Pixy Kickstarter Video. https://www.youtube.com/watch?v=J8sl3nMlYxM

[2] Charmed Labs. Pixy. https://github.com/charmedlabs/pixy

[3] Charmed Labs. Pixy at e87bb729f472f202e440681a5871a58bf9f685f7. https://github.com/charmedlabs/pixy/tree/e87bb729f472f202e440681a5871a58bf9f685f7

[4] Boost contributors. Boost C++ Libraries. http://www.boost.org/

[5] Jeremy Rand. Depth Imaging with the Asus Xtion Pro Live Sensor and the KIPR Link (Parts 1 and 2). Proceedings of the 2013 Global Conference on Educational Robotics.

[6] Kitware. CMake. http://www.cmake.org/

[7] CMUcam5 Pixy contributors. Wiki – CMUcam5 Pixy. http://cmucam.org/projects/cmucam5/wiki

[8] Botball Youth Advisory Council. Botball Community. http://community.botball.org/

[9] Charmed Labs. pixy.h File Reference. https://charmedlabs.github.io/pixy/pixy_8h.html