Project Overview

The goal of Project Ellesmere is to streamline identity verification in New York airports by installing physical terminals equipped with the ability to read and verify mDL data extracted from an EWA device. The data transfer is executed via an ISO 1803-5 compliant NFC/Wifi-Aware handoff sequence. The terminals will be connected to a CAT machine via usb.

Driver Function

The function of the CAT-side usb driver that I've written is to enable effective USB half-duplex data transfer between the dongle and the CAT machine (which is the host). I implemented the Android Accessory Protocol using the low-level API of usb4java, a java wrapped C library, for host-side operations and android.hardware.usb classes for device-side operations. Unfortunately, I was unable to verify whether or not the driver facilitated a successful data exchange between my PC and the Hikey board as my testing was cut short by a hapless corruption of crucial Hikey system files which disabled adb over usb capabilities.

I have written a few drivers that differ in style and minor implementation details. A procedurally written driver is located under the master branch with filepath: host\procedural. There are two object-oriented drivers under the masterbranch with filepath: host\oop\async and host\oop\sync, respectively. The former transfers data in an asynchronous manner which allows for a degree of simultaneity without multithreading while the latter transfers data in a synchronous manner and is the closest to completion.

Dongle-side App

The dongle-side application is located under the master branch under the device folder. It is triggered once the hikey is in accessory mode and enters an infinite loop in which it constantly tries to read in data from the host (using FileInputStream) and, once it succeeds in doing so, sends a USBResponse proto object (using FileOutputStream). This process is repeated for an indefinite number of iterations.

Data Serialization

In terms of data serialization, I read a fair bit about Google’s protocol buffers. Early on, I wrote a fairly naïve but functional program that transforms java objects directly into protobuf generated classes, saving the time of manually writing a .proto file. The program only works for simple cases but can handle nested objects(objects with objects (with objects… and so on) as data fields) complete with proper indentation. This program is located under the master branch at: serialization\protobuf\testing\objproto and is named ProtoConverter.java. The accompanying files are output, and templates.

A Maven build automating the compilation of .proto files is located under the master branch at: serialization\protobuf\testing\mvngooglex\autoprotoc. It uses an open-source plugin.

The asynchronous transfer driver uses protobufs as a serialization mechanism for the CAT machine as does the dongle-side app. The protobuf objects are built, converted to byte arrays and sent out on streams(dongle-side) or using LibUsb.bulkTransfer() (CAT-side).