DemoFlow: Low-Cost Flow Cytometry for Increased Access to Medical Diagnostics

by
Kyler Stole

A THESIS

submitted to

Oregon State University

Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Associate)

Honors Baccalaureate of Arts in International Studies
(Honors Associate)

Presented March 13, 2018
Commencement June 2018

AN ABSTRACT OF THE THESIS OF

Kyler Stole for the degree of Honors Baccalaureate of Science in Computer Science and Honors Baccalaureate of Arts in International Studies presented on March 13, 2018.
Title: DemoFlow: Low-Cost Flow Cytometry for Increased Access to Medical Diagnostics

Abstract approved: ______________________________

D. Kevin McGrath

Flow cytometry is a highly-extensible technology for performing rapid multi-parametric particle analysis. The use of flow cytometry has proliferated with the advent of smaller, intuitive, and more affordable instruments, along with a perpetual increase in research and medical applications. However, the technology is still largely inaccessible due to cost and portability. The *DemoFlow* project is a prototype low-cost flow cytometer that provides considerable functionality at an affordable price. It is simpler than a typical industry flow cytometer, but is capable of performing useful scientific assays. DemoFlow is a portable system that interfaces with an accompanying Android application to provide simple, easy-to-use data collection and analysis. Such a system has the potential to impact how diseases are handled in resource-limited settings.

Key Words: flow cytometry, Android, embedded systems, graphing, medical

Corresponding e-mail address: kyler.stole@lifetime.oregonstate.edu

DemoFlow: Low-Cost Flow Cytometry for Increased Access to Medical Diagnostics

by
Kyler Stole

A THESIS

submitted to

Oregon State University

Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Associate)

Honors Baccalaureate of Arts in International Studies
(Honors Associate)

Presented March 13, 2018
Commencement June 2018

Honors Baccalaureate of Science in Computer Science and Honors Baccalaureate of Arts in International Studies project of Kyler Stole presented on March 13, 2018.

APPROVED:

D. Kevin McGrath, Mentor, representing Electrical Engineering and Computer Science

Mike Bailey, Committee Member, representing Electrical Engineering and Computer Science

Wesley Smith, Committee Member, representing Thermo Fisher Scientific, Inc.

Rebekah Lancelin, Representative for International Studies Degree Program

Toni Doolen, Dean, Oregon State University Honors College

Kyler Stole, Author

Contents

1 Introduction

THE field of medicine is vast and the emergence of highly-extensible flow cytometry technology has impacted it greatly. The term *flow cytometry* comes from the prefix *cyto-*, meaning cell, and the suffix *-metry*, meaning measurement [1]. Flow cytometers are often used on blood samples to perform cell counts, cell sorting, or to measure cell health. However, flow cytometers can be used in applications beyond standard hematology as generic particle analyzers. They can perform both qualitative and quantitative analysis of samples and are useful in a wide range of applications such as medical research, where they are used at the forefront of cancer research, and for providing medical diagnoses. Flow cytometers are especially valuable for the velocity at which they can collect simultaneous multiparametric data on huge populations. They are also able to detect cells of interest in extremely dilute samples where other technologies only work on higher concentrations [2].

The increasing availability of flow cytometry instruments and reagents at more portable sizes and lower price points has opened up new applications for the technology. Breweries can use flow cytometry to investigate yeast populations [3], scientists can use the technology when studying aquatic microbiology [4], and flow cytometry has replaced previous results acquisition procedures in a variety of medical scenarios [5]. However, the technology is still not generally accessible to most people. Cost and portability are the top restraining factors.

This project, titled *DemoFlow*, has the goal of creating a low-cost flow cytometer that is relatively portable, simple to use, and costs about a tenth the cost of a typical industry flow cytometer. Such an instrument would democratize flow cytometry in a way that would expand applications for its use. While there are other factors to consider in the instrument that decide whether it is applicable in certain cases, this project aims to show through a prototype that the technology could be made available at a reduced cost. It demonstrates a low-cost option for democratizing flow cytometry; thus the name, DemoFlow.

2 Background

This project's sponsor, Thermo Fisher Scientific, makes a high-end flow cytometer called the *Attune NxT*. The Attune NxT can be configured with four lasers (of different wavelengths) and 16 detectors. In comparison, the DemoFlow project provides fewer functions, slower processing, and lower precision at a reduced cost. The DemoFlow flow cytometer operates on a single sample at once with a single laser excitation source and three separate detectors. Such an instrument would be affordable for universities to educate students using the simplified design, allow brewers and other hobbyists to measure exact data from their projects, and make the technology available for medical diagnoses in impoverished areas.

DemoFlow started as a senior capstone project between my computer science (CS) team and a mechanical engineering (ME) team, supported by Wesley Smith from Thermo

Fisher Scientific. The ME team developed the system's structure and optics with direct assistance from Wesley, who continued to perfect the design after the ME team finished its capstone work. The CS team was responsible for electronics within the device and an external software application to display data visualizations. Following the conclusion of capstone, I continued to develop the DemoFlow project and expanded it to investigate how the technology can impact disease management in resource-limited settings.

2.1 Internals of a flow cytometer

A flow cytometer is a device that measures particle properties by interrogating a moving stream of particles from a fluid sample with an excitation source. Figure 1 shows the components of a typical flow cytometer. The excitation source is some type of laser that emits a known wavelength. The sample is pumped through the flow cell, where it passes through a narrow capillary at the excitation point. Particles are forced to the center of the pathway, often with the help of surrounding sheathing fluid, and in the case of the Attune NxT, with additional help from acoustic focussing, which vibrates the flow cell at high frequency to keep particles in alignment.

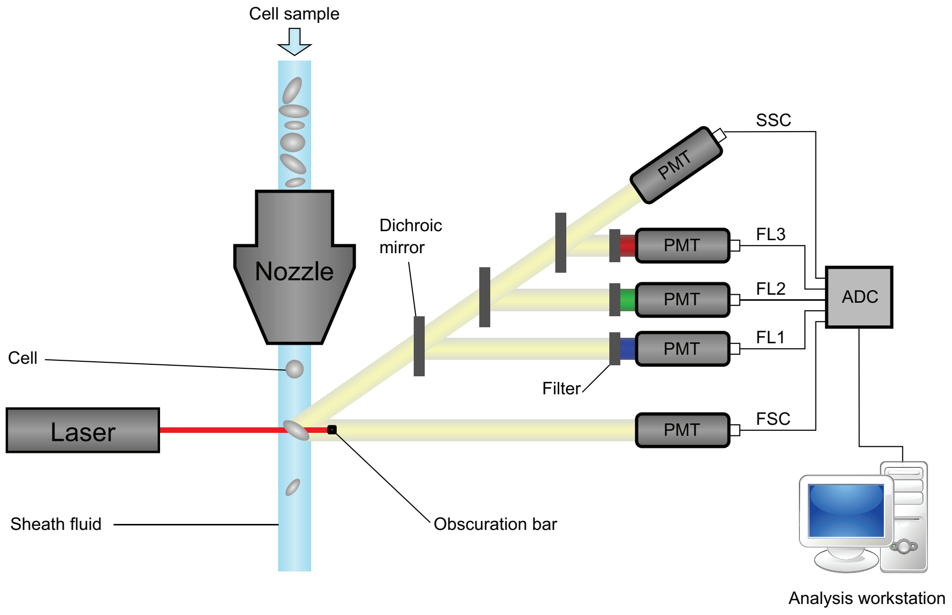


Figure 1: Diagram showing the components of a typical flow cytometer

When a particle passes through the excitation point, it scatters the laser beam, which is picked up by a detector directly in front of the laser (forward scatter detector - FSC) and at a right angle to the laser beam (side scatter detector - SSC). FSC gives an indication of the size of the particle passing through the beam since larger particles will generate larger pulses on the forward scatter detector. SSC gives an indication of particle granularity since more complex particles deflect more light to the side [2] [5].

While there are typically only two scatter detectors, there can be a greater number of fluorescence detectors. Flow cytometer operators typically mix fluorescent biomarkers with the sample. The biomarkers, called fluorochromes, are activated by a known wavelength of energy before falling to a lower energy state, releasing a different but known wavelength of energy. By filtering the light emitted from the particle, fluorescent detectors can detect these biomarkers. Detectors can be basic photodiodes but are typically photomultiplier tubes (PMTs), which amplify the weak signals they receive.

In addition to data collection, some flow cytometers are equipped with the capability to physically separate identified cells. Such an instrument is known as a fluorescence-activated cell sorter (FACS) [2]. This works by selecting cells through pre-set data gates and then diverting them using electrical or mechanical means such as charged metal plates.

2.2 Flow cytometry hardware

There are a wide variety of competing flow cytometers in the industry. Some examples are the Attune NxT, from our project sponsor; the BD Accuri, from Becton Dickson; and the CytoFLEX platform, from Beckman Coulter [6]. Another example, which is more similar to the DemoFlow device, is the MilliporeSigma Muse Cell Analyzer. It has an integrated screen in a compact unit, simple instruction flow for preset assays, and costs around $14,000 [7].

While each flow cytometer series may have its own specialties, there are some general specs that change between lower- and higher-end models. More expensive flow cytometers will have more fluorescence detectors and potentially more lasers. Additional lasers and detectors quickly add to the price and complexity of the flow cytometer, but make it more versatile by allowing it to detect a wider variety of reagents. They can also better distinguish between different cell populations in some cases, which is important because data on multiple fluorochromes may overlap.

Beyond the lasers and detectors, each flow cytometer has some sort of microcontroller unit (MCU) with (proprietary) firmware, an optics system for channeling the light emitted from particles at the interrogation point, and a fluidics system with some sort of pump (such as a syringe pump or peristaltic pump). Electrical systems also include the interface between the MCU and detectors, the interface between the MCU and the pump, and the interface between the MCU and the flow cytometry application (Bluetooth in the case of DemoFlow).

2.3 Flow cytometry software

The utility of a flow cytometer is highly dependent on the flow cytometry software that accompanies the instrument. Most flow cytometers come with a software application that runs on a standalone computer although some flow cytometers integrate a screen with the software into the device. Flow cytometry software is used to collect data, create charts of the data, and analyze data by gating regions of plotted points. Data obtained from flow cytometry experiments are stored in an industry standard file format: the Flow Cytometry Standard (FCS) format.

Since software is such a key component of flow cytometry, there are a few features that are common to all flow cytometry programs. First, for data collection, the operator needs to be able to specify parameters for the experiment. Basic parameters are the volumetric sample size, sample dilution factor, and flow rate (how fast the sample moves through the flow cell. These few parameters usually provide enough information to start data collection. Data is collected as a growing set of events. Each event represents a particle passing through the excitation point and is made up of values for each available detector, also known as parameters (FSC, SSC, FL1, etc. are all parameters). Some systems will report multiple parameters for a single detector, reflecting the height (H), width (W), and/or area (A) of the pulse registered at the detector. Thus, for example, FSC-H and FSC-A data would represent height (the maximum value of the pulse) and area (the side of the pulse) for a single FSC detector.

The thing that sets flow cytometry programs apart from each other is their ability to provide decent charts and data gating. There are only two core types of charts that are used for plotting flow cytometry data: bivariate scatter charts, which display one parameter on the x-axis and another on the y-axis, and histograms, which show a frequency plot for a single parameter. Sometimes bivariate scatter can be shown as density plots, which uses color to indicate areas containing densely-packed points. An additional complexity to charting the data is that these charts often need to make use of logarithmically-scaled axis values to present the data in an effective way. Otherwise, the data may be undesirably skewed and stacked on one side of the chart.

Gating regions are used to select a population of data, which can then be applied to another chart to filter the plotted data. Drawing gates on a histogram is rather simple as the user just needs to indicate a range of inclusion. There are several ways to implement gating on scatter plots. The simple approaches involve drawing rectangular or elliptical data gates since those are basic shapes. A more useful gate is a custom polygon, with straight edges connecting user-defined vertices. Taking that to the extreme, some software even supports drawing freeform gates with curved lines, although this adds little on top of polygonal gates. Finally, some flow cytometry applications also provide quadrant gating, which may be adjusted arbitrarily to separate the chart into four quadrants, essentially creating four gates at once. This is useful for applications where data are naturally split into four distinct populations by the combination of two binary characteristics (typically one half separates dead cells from live cells and the perpendicular splits some other characteristic).

2.4 Glossary

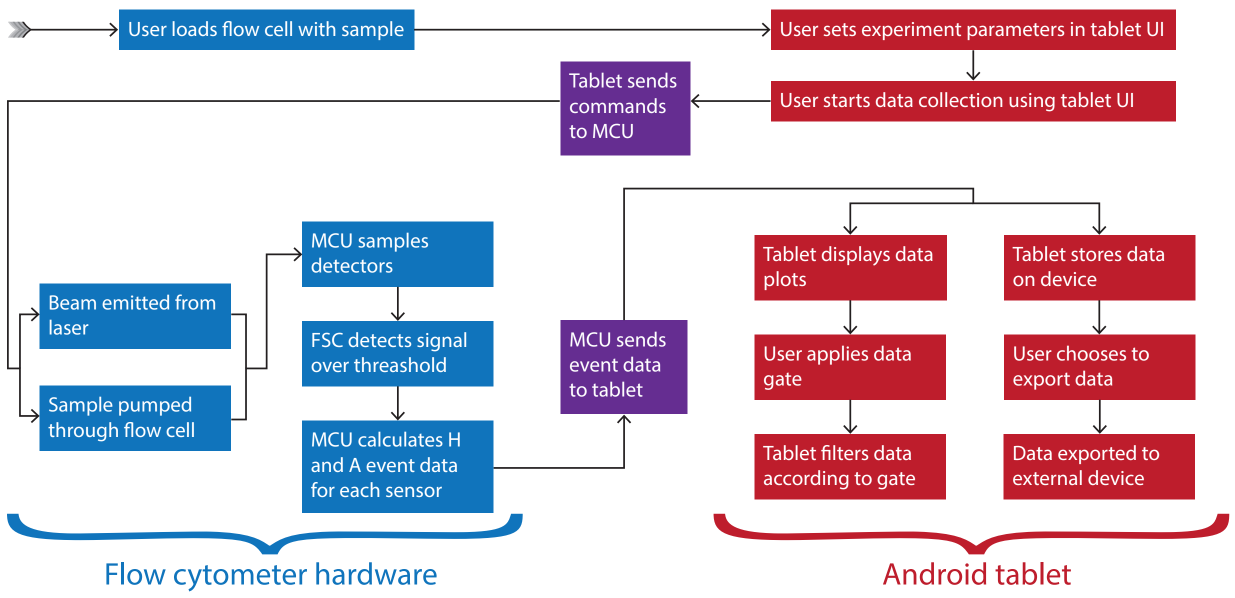
API	**application programming interface** The set of definitions and protocols defining the format for communicating with a particular technology.

—	**assay** An investigative procedure to devise the presence or measure the amount of a target substance.
—	**data gating** The process of drawing gating regions around populations (i.e. selecting a group of points). Data gates are applied to charts to filter a particular subset of the data (an important concept in flow cytometry).
—	**flow cell** A chamber designed to move particles in as close to single-file as possible through the interrogation point [8].
—	**flow cytometry** A technology for particle analysis that involves measuring light scatter and fluorescence as particles pass through a laser beam.
FCS	**Flow Cytometry Standard** The ISAC standard for flow cytometry data sets that describes files containing experimental data. The current standard, introduced in 2010, is FCS 3.1 [9], [10].
ISAC	**International Society for Advancement of Cytometry** The main organization for advancing and standardizing cytometry practices.
—	**Java** The main programming language used for Android application development.
MCU	**microcontroller unit** A small computer on a single chip with a microprocessor, memory, I/O pins, and other modules.
UI	**User Interface** The visual part of a computer application or operating system through which a user interacts with a program.

3 DemoFlow

The DemoFlow project combines several distinct components to create a fully functional flow cytometer. The mechanical engineering team worked on an enclosure containing a flow cell assembly and optics system. The laser and detectors are integrated into the optics system and the syringe pump connects to the flow cell assembly. DemoFlow uses an Android application to manage data acquisition, display data visualizations, and provide system controls. The electronics are separate from the ME team's work, as is the Android tablet and its DemoFlow application. The Android application can be used independently to analyze data stored in FCS files but the other components rely on the entire interconnected system.

The functional diagram in Figure 2 demonstrates the functionality of the entire system. This diagram shows a high-level overview of the flow that occurs during typical use of the system. The functional overview is split between actions that occur in the flow



User loads flow cell with sample
User sets experiment parameters in tablet UI
User starts data collection using tablet UI
Tablet sends commands to MCU
MCU samples detectors
Beam emitted from laser
Sample pumped through flow cell
FSC detects signal over threashold
MCU calculates H and A event data for each sensor
MCU sends event data to tablet
Tablet displays data plots
User applies data gate
Tablet filters data according to gate
Tablet stores data on device
User chooses to export data
Data exported to external device
Flow cytometer hardware
Android tablet

Figure 2: Functional diagram of the DemoFlow system showing user interaction and flow for the entire system

cytometer's hardware (in blue) and actions that occur in the Android tablet application (in red). Purple shows the communication between the flow cytometer and the Android tablet. For a more complete breakdown of the processes that occur in the flow cytometer hardware and the Android application, see Sections 6 and 4 respectively.

3.1 Original requirements

The original requirements for the system can be found in Appendix C. Requirements in this list were given priorities and there were several core requirements. The ability for the software to read and export files in the FCS file format was vital because it allows the application to open experiments that were run on other instruments and allows experimenters to export their data for analysis in other software. The DemoFlow application supports the FCS file format through a custom FCS parsing package. Several requirements relate to the need for charting and data gating. The DemoFlow application supports span gates for histograms and polygonal gates for bivariate scatter charts with gate manipulation controlled via multi-touch gestures.

4 ANDROID APPLICATION

The software that accompanies the DemoFlow flow cytometer is an Android application that runs on any reasonably-powerful Android tablet. The application is used both for starting experiments and collecting data as well as providing data visualizations and export of experiment data in the FCS file format. Additionally, users can load FCS files that were generated by other flow cytometers and make use of DemoFlow's charting and gating capabilities. The application allows users to plot data in histograms and bivariate scatter charts, draw span gates and polygonal gates respectively, and apply gates to charts.

The DemoFlow Android project contains two packages; `fcsparser` is a full-featured Java FCS parser and `app` contains all of the Android application's activities, file manipulation, and plotting features. The `fcsparser` package is described in Section 5.

The `app` package has subpackages for `files`, `plots`, and `gating`, which cover the classes that are used primarily for those purposes. The `files` subpackage includes the activity and supporting classes to display a navigable view of files and folders stored on the device. The `plots` subpackage contains the Plots activity, which is the defining screen of the application, as well as some helper classes and structures for storing data that defines charts. The `gating` subpackage contains classes that help the user draw data gates on top of charts using multi-touch gestures.

4.1 Environment

All Android development is done using the latest version of Android Studio with Java Development Kit JDK 1.7. Android Studio will prompt to install any packages that are missing when the code is compiled. The application can be loaded onto an Android emulator, such as Android Studio's built-in Android Emulator or Genymotion, or to a physical device from within Android Studio. Initial Android testing was performed in the Android Emulator on multiple 10″ devices with Google Apps support. Physical device testing was performed on a 9.7″ Samsung Galaxy Tab S2 tablet running API level 23 (Android version 6.0.1) and later API level 24 (Android version 7.0).

4.2 User interface

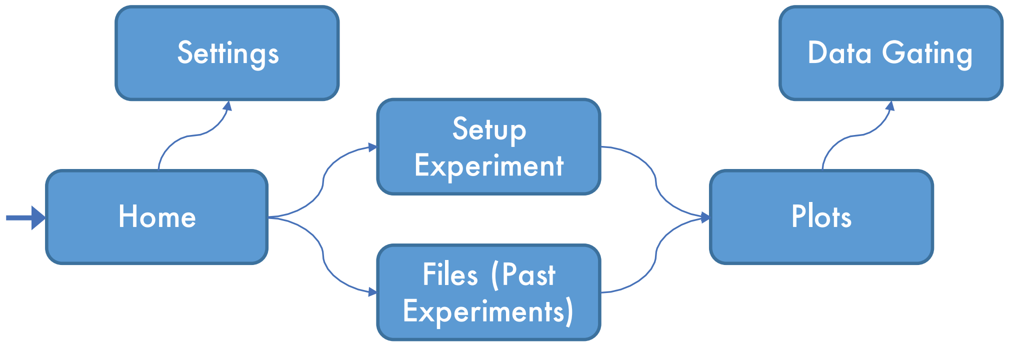


Figure 3: Diagram showing the structure (app flow) of DemoFlow screens

The user interface of the DemoFlow app leverages many core Android elements, such as dialogs, but also makes heavy use of a charting library and custom design elements. The structure of the screens (Figure 3) is meant to simplify the process of collecting data or working with preexisting FCS files.

The **Home screen** (Figure 4) is designed to be friendly and simple. For that reason, it has two large buttons for the two main paths through the app: collecting new data and opening files. It also has a small Settings button in the top corner (known as the ActionBar area in Android), which shows a screen to edit app-wide settings.

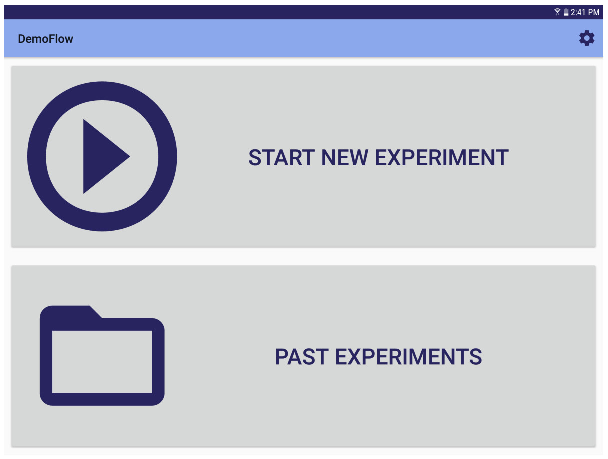
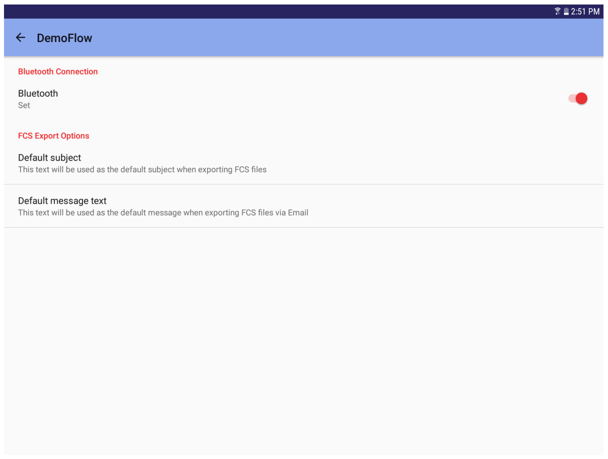


Figure 4: Home screen with buttons to set up a new experiment, access past experiment files, and navigate to app settings

The **Settings screen** (Figure 5) uses Android's PreferenceFragment class, which is the standard way of presenting persistent app-level settings. Values are stored in the Shared-Preferences object for the app, which is initialized to default values when the app is first run. Figure 5 shows the current settings available in the app settings. Additional settings will be added in the future for default FCS parameters and preferences for how the app should behave.



Bluetooth Connection

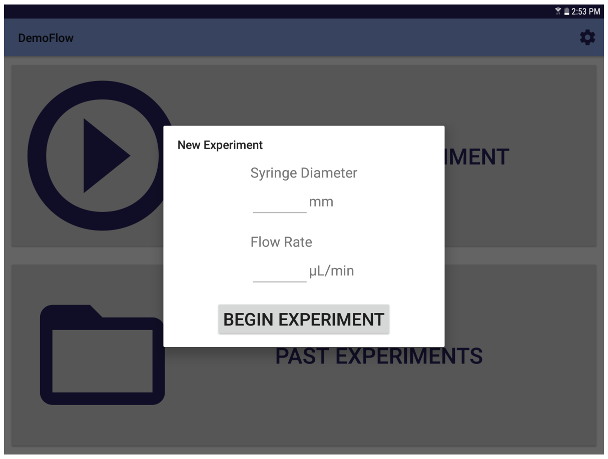
Bluetooth
Set

FCS Export Options

Default subject
This text will be used as the default subject when exporting FCS files

Default message text
This text will be used as the default message when exporting FCS files via Email

Figure 5: App Settings screen using an Android PreferenceFragment



New Experiment

Syringe Diameter
______ mm

Flow Rate
______ µL/min

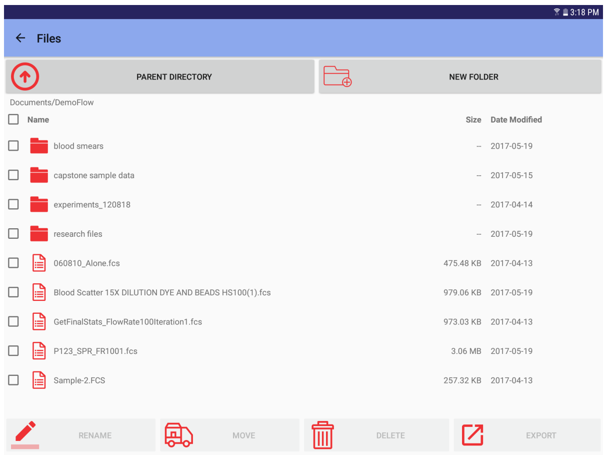
BEGIN EXPERIMENT

Figure 6: Setup experiment activity displayed as a dialog with options to configure syringe diameter and flow rate parameters

The **Experiment Setup activity** (Figure 6) has been modified to present itself as a simple dialog box, which takes as input the two necessary experiment setup parameters: syringe diameter and flow rate. The Begin Experiment button launches the Plots screen and sends the flow cytometer the necessary commands to begin collecting data.

The **Files screen** (Figure 7) shows FCS files and directories with options to add new directories, navigate into directories, and navigate up the directory hierarchy. It also provides the ability to select files and directories and then rename, delete, move, or export them (for FCS files only). Finally, tapping an FCS file loads the Plots screen and parses the file's contents into memory.

The **Plots screen** (Figure 8) displays a configurable grid of histograms and scatter charts, which resize as charts are added and deleted. Charts can be added and configured using a dialog that allows users to choose the chart type, parameters for each axis, whether the x-axis should use logarithmic scaling, and which data gate should be applied, if any. The left portion of the Plots screen displays statistics about current data collection.



Documents/DemoFlow

Name	Size	Date Modified
blood smears	--	2017-05-19
capstone sample data	--	2017-05-15
experiments_120818	--	2017-04-14
research files	--	2017-05-19
060810_Alone.fcs	475.48 KB	2017-04-13
Blood Scatter 15X DILUTION DYE AND BEADS HS100(1).fcs	979.06 KB	2017-05-19
GetFinalStats_FlowRate100Iteration1.fcs	973.03 KB	2017-04-13
P123_SPR_FR1001.fcs	3.06 MB	2017-05-19
Sample-2.FCS	257.32 KB	2017-04-13

Figure 7: Files screen showing multiple directories and FCS files arranged in alphabetical order (with directories first)

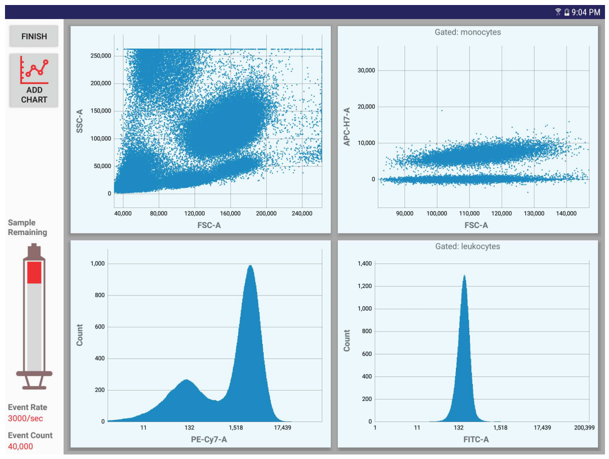


Figure 8: Plots screen showing bivariate scatter charts and histograms created from data stored in an FCS file

The **Gating screen** shows a single chart onto which data gates are drawn and a sidebar with statistics and the list of gates for the corresponding chart. The gates list provides functionality for adding, deleting, renaming, and editing gates. Gating is described in detail in Section 4.4.

4.3 Plotting

The DemoFlow app uses an Android graphing library, MPAndroidChart, to plot bivariate scatter charts and histograms (bar charts). Both types of charts can be added and configured from a dialog (which uses the Android DialogFragment class). Users can reconfigure and delete existing charts by long pressing to bring up the dialog again.

Past experiment data are read into the Plots activity from FCS files using the FCS parsing package. This makes parameters stored in the FCS file available to be plotted as the chart axes. This plotting functionality has been tested with multiple FCS files found in online repositories and supplied by Thermo Fisher.

4.3.1 Logarithmic scaling

The ability to plot data on a logarithmically scaled axis is necessary for viewing most flow cytometry histograms because the data are naturally skewed. Using a logarithmic axis normalizes the data distribution, making it usable for analysis. Figure 10 shows a histogram that has been plotted on a logarithmic x-axis. This chart demonstrates how a span gate can be drawn to select a particular segment of data that has been separated distinctly due to the logarithmic scaling. If this histogram were plotted on a linear x-axis, the two distinct peaks would be smashed together against the left y-axis.

Since MPAndroidChart does not natively support logarithmic scaling of its axes, the logarithmic transformation must be calculated manually. This is currently only available for histograms, and works by applying a logarithmic transformation to the values as they are placed in bins. Due to the nature of the logarithmic function, values less than 1 require special treatment and this feature currently drops those values. Full support for values less than 1, including negative values will be added in a future version. In addition to transforming values as they are binned, special calculations are needed when displaying axis labels on the altered x-axis and when working with span gates drawn onto logarithmically scaled histograms.

Code for managing logarithmic scaling in histograms can be found in Appendix A.2. Listing 4 shows the initial binning process for histograms. This relies on a custom `AxisValueFormatter`. Line 14 shows where the bin index for each event value is calculated, after which the count for that bin is incremented in line 17 (except values off the chart, which are dropped in line 15).

Listing 6 shows the important parts of the LinearLogarithmicAxisValueFormatter, which formats labels that are added to the x-axis of histograms. The values needed to perform logarithmic value transformations are calculated in lines 10-12 when the formatter is initialized. These values are then used in line 21 and line 30 when converting between logarithmically scaled bin values and values from the FCS file.

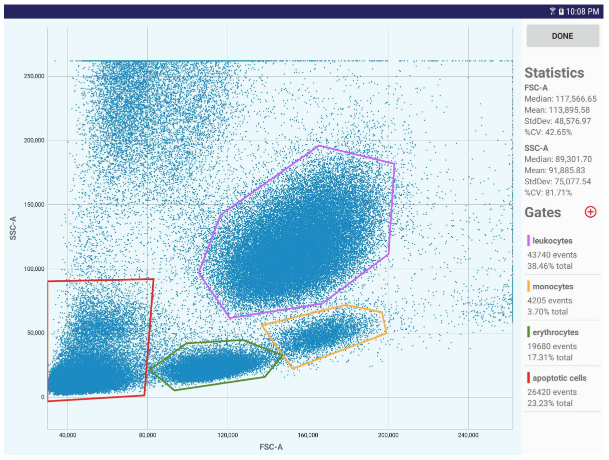
4.3.2 Averaged histograms

Since a histogram is essentially a bar chart showing the binned frequencies of a continuous random variable, the chart can look a bit rough and sporadic when plotted directly. Since the point of plotting flow cytometry data in a histogram is to isolate distinct peaks, it can be helpful for analysis to smooth the plotted data [11]. DemoFlow does this by computing a central average for each bar of the chart (averaging the original point with data equally spaced to either side of the point) and plotting the computed average. Listing 5 in Appendix A.2 shows how this is implemented. Special care has to be taken when computing averages for values at the ends of the chart where the full range of past and future data is not available.

4.4 Data gating

The data gating capabilities of the DemoFlow application changed significantly over multiple iterations. Gating for bivariate scatter charts changed most noticeably, starting with a simplified bounding approach, then a drawable rectangle, and finally custom polygons. Gate drawing works with a custom gating view superimposed on the content region of the chart. This view is hooked up to a touch event listener that detects individual touches, interprets them as gestures, and uses those gestures to manipulate polygonal and span gates.

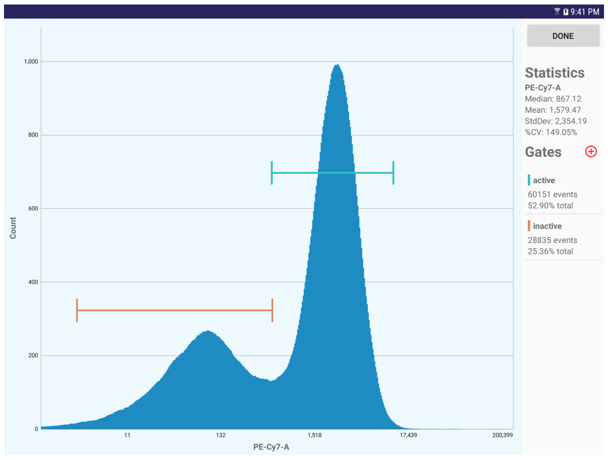
Figures 9 and 10 show polygonal gates on a bivariate scatter chart and span gates on a histogram, respectively. The sidebar on the right shows statistics for the overall dataset and a list of gates for the corresponding chart with statistics about each visible gate. While the interface for these is very similar, there are major differences between the handling of bivariate scatter charts and histograms. Each uses a different type of `DataGateView` for the layer onto which the gates are drawn and there are separate polygonal and span gate touch listeners to interpret gestures for drawing those gates. The sidebar also has a small section of help text at the bottom, which provides relevant instructions when editing a polygonal or span gate.



Statistics
FSC-A
Median: 117,566.65
Mean: 113,895.58
StdDev: 48,576.97
%CV: 42.65%
SSC-A
Median: 89,301.70
Mean: 91,885.83
StdDev: 75,077.54
%CV: 81.71%

Gates
leukocytes
43740 events
38.46% total
monocytes
4205 events
3.70% total
erythrocytes
19680 events
17.31% total
apoptotic cells
26420 events
23.23% total

Figure 9: Polygonal gates drawn on a bivariate scatter chart

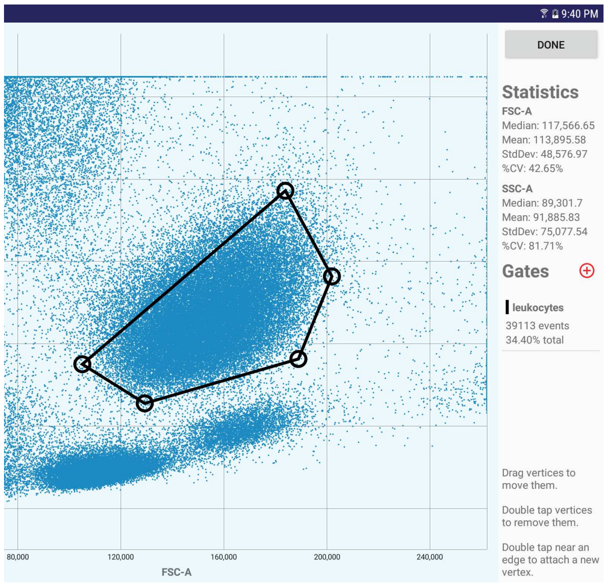


Statistics
PE-Cy7-A
Median: 867.12
Mean: 1,579.47
StdDev: 2,354.19
%CV: 149.05%

Gates
active
60151 events
52.90% total
inactive
28835 events
25.36% total

Figure 10: Span gate drawn on a histogram with logarithmic scaling

Figure 11 shows a polygonal gate that is being edited with the appropriate help text displayed in the sidebar. The events captured by a gating region are calculated as the gate is edited so that the statistics for the current gate can be updated in the sidebar. This filtering process is described in Section 4.4.3. Gates that have been created can be applied to charts on the Plots screen with the configure chart dialog as seen in Figure 12. The gate selection dropdown, which is empty when there are no gates available, lists the gates that can be applied to the chart.



Statistics

FSC-A
Median: 117,566.65
Mean: 113,895.58
StdDev: 48,576.97
%CV: 42.65%

SSC-A
Median: 89,301.7
Mean: 91,885.83
StdDev: 75,077.54
%CV: 81.71%

Gates

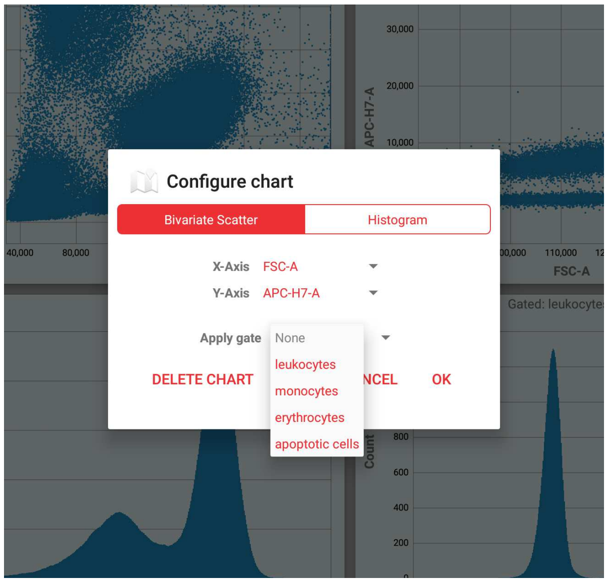
leukocytes
39113 events
34.40% total

Drag vertices to move them.

Double tap vertices to remove them.

Double tap near an edge to attach a new vertex.

Figure 11: Editing a polygonal gates drawn on a bivariate scatter chart



Configure chart

Bivariate Scatter | Histogram

X-Axis FSC-A

Y-Axis APC-H7-A

Apply gate None

leukocytes
monocytes
erythrocytes
apoptotic cells

DELETE CHART NCEL OK

Figure 12: Applying a data gate through the configure chart dialog

4.4.1 Interpreting gate drawing gestures

One of the benefits of flow cytometry software on a tablet is the multi-touch interface. Gates are drawn by the user directly on top of the chart with a `Polygonal` or `Span GateViewOnTouchListener` interpreting the gestures. The touch listener for span gates is relatively simple. It essentially sets the left and right bounds of the gate according to the x-coordinates of two input pointers (fingers) on the screen. To make the parts of the gate adjustable by a single finger (including the center bar), the listener calculates the correct mode of transformation using the function in Listing 7 from Appendix A.3.

The touch listener for polygonal gates deals with a bit more complex input and uses an Android `GestureDetector` to interpret double tap and scroll (translation) gestures. Coming up with an intuitive mechanism for creating and editing polygonal gates was difficult because there are several potential actions that need to be taken into consideration. Unlike span gates, which are always composed of two boundaries, vertices in polygonal gates must be added, removed, and moved. Combining these actions into the same interface can lead to complications. For example, if a vertex were removed with a long press, then it would be possible to accidentally remove a vertex when trying to move it slightly if the gesture detector registered the long press before it recognized the scrolling action. Another source of complication is the need for an enlarged touch target surrounding each vertex, which may intersect the touch targets of other vertices. Additionally, since a polygonal gate is useless if it has fewer than three vertices, a different mode of operation might make sense when adding the first few vertices. DemoFlow deals with this by allowing the user to plot the first three vertices by single tapping on the chart. After the third vertex is placed, drawing automatically changes to the normal mode of operation and the help text in the sidebar is updated.

Any vertex can be moved by simply dragging it. Double tapping on a vertex removes that vertex. Double tapping near an edge attaches a new vertex to that edge. To avoid confusion, vertices are only added to edges if the double tap occurs relatively close and perpendicular to the edge.

To make the interface consistent, the same algorithm is used when any type of touch begins (i.e. a single tap, double tap, or drag). The listener first searches for the vertex closest to the touch point. If the distance to the closest vertex is within a threshold, then that vertex is modified according to the gesture. Otherwise, assuming a double tap gesture, the listener searches for the closest edge. Edge distances are calculated by a modified algorithm for calculating the perpendicular distance between a point and a line segment that returns an extremely large value if the point is not perpendicular to the line segment. Again, if the distance to the closest edge is within a threshold, a vertex is added to that edge. If no edge is close enough, then nothing happens.

As a final adjustment to the drawing rules, vertices are not allowed to fall off the chart. Nor are the bounds of span gates. If a touch is detected beyond the bounds of the chart, that axis value is set to the closest point that still lies on the chart.

4.4.2 Aligning DataGateView with the chart

One of the major struggles in developing effective gating was matching the view onto which gates are drawn (DataGateView) with the content region of the chart (i.e. the rectangular area bordered by the x and y axes). This required calculating appropriate layout margins for the DataGateView so that it lines up exactly with the chart. However, this has to be done when the chart is already drawn or else it will not have calculated the content region. This is accomplished by setting an `OnLayoutChangeListener` that responds to updates in the chart position.

Listing 1: Calculating layout margins for positioning DataGateView.

```
1  int[] frameAnchor = new int[2];
2  gateFrame.getLocationOnScreen(frameAnchor);
3  int[] chartAnchor = new int[2];
4  chartView.getChart().getLocationOnScreen(chartAnchor);
5  Rect chartBounds = new Rect();
6  chartView.getChart().getContentRect().round(chartBounds);
7
8  int marginLeft = chartAnchor[0] - frameAnchor[0] + chartBounds.left;
9  int marginTop = chartAnchor[1] - frameAnchor[1] + chartBounds.top;
10 int marginRight = frameAnchor[0] + gateFrame.getWidth() - ( chartAnchor[0] +
       chartBounds.right );
11 int marginBottom = frameAnchor[1] + gateFrame.getHeight() - ( chartAnchor[1] +
       chartBounds.bottom );
```

Listing 1 shows how layout margins are calculated for the gating view. In lines 1-4, the screen location of the chart view and its parent frame are retrieved. Then, in line 6, a method provided by MPAndroidChart is used to locate the content region of the chart, relative to the entire chart. Finally, lines 8-11 use these positions to calculate the necessary layout margins, relative to the parent frame. The calculated margins are then used to set the `LayoutParams` of the gating view.

4.4.3 Filtering events

When a gate is drawn, the gate calculates which events it contains and uses that list for statistics. Filtering events contained within a gating region is a two step process. The first step is to take the gate that has been drawn on the screen and store it as a series of vertices in the chart's coordinate system. This involves converting the range of screen position values to the range of axis values, which can be done with the function in Listing 2. This function makes it easy to convert values from one range to another given the minimum and maximum values of each range. It even works if the ranges include negative values. This function is also used when loading a stored gate onto the screen. For histograms, in addition to converting between value ranges, the LinearLogarithmicAxisValueFormatter must be used to match values with bins.

Listing 2: Convert a value in one range to the corresponding value in a different range.

```
public static double convertValueRange(double value, double oldMin, double oldMax,
    double newMin, double newMax) {
    // oldRange = (oldMax - oldMin)
    // newRange = (newMax - newMin)
    // newValue = newMin + (oldValue - oldMin) * newRange / oldRange
    return newMin + (value - oldMin) * (newMax - newMin) / (oldMax - oldMin);
}
```

The second step is to find which points lay within the boundaries of the gate. That is a simple matter for span gates, which are univariate and just check that each event lies to the right of the left bound and to the left of the right bound. For polygonal gates, filtering events is not a trivial matter, and it was for this reason that rectangular gates were first used. The solution is to use one of several algorithms for detecting the inclusion of a point in a simple polygon. Two notable algorithms are the crossing number algorithm, which counts the number of times a ray starting at the point crosses an edge of the polygon, and the winding number algorithm, which counts the number of turns made around the point when following the path all the way around the polygon. The crossing number algorithm has typically been more popular for computer science applications because it is fairly simplistic whereas the winding number algorithm uses trigonometric functions to add up all of the angles made between the point and each edge of the polygon [12]. The winding number, however, also works in non-simple closed polygons and for a point enclosed by overlapping sections, gives the number of layers. The DemoFlow application uses a highly-optimized implementation of the winding number algorithm, adapted from code provided by Dan Sunday, which does not compute exact angles and runs with the same time complexity as the crossing number algorithm [12].

4.4.4 Accelerating chart drawing

MPAndroidChart provides charts that are dynamic; they can be scaled and translated via multi-touch gestures. The downside of this is that they are not static when sitting under a DataGateView. When drawing occurs on the DataGateView layer, the chart has to redraw its content. This introduces considerable lag when drawing a data gate on top of thousands of points, especially considering movement within the chart is already disabled when data gates are active. The solution is to draw the chart view to a static buffer, which is then ready whenever the view needs to be rendered to the screen. Android provides a view method that makes this as simple as calling `setLayerType` on the view.

```
chart.setLayerType(View.LAYER_TYPE_SOFTWARE, null);
```

This stores the view in a software buffer and removes all noticeable lag when drawing gates, even with large data sets. It is also possible to use a hardware buffer but the initial set up produces a delay and a software buffer is sufficient here. Restoring chart functionality can be done by setting the layer type to `View.LAYER_TYPE_NONE`.

4.5 FCS file export

The ability to export experiment data in the FCS file format was a core requirement for the system because it provides an interface to work with DemoFlow data in other flow cytometry software packages and vice versa. This was made simpler by already storing DemoFlow data in FCS files. The export mechanism uses Android's built-in `ShareCompat.Builder` object to support multiple export options. This creates an Android Implicit Intent, which is a way of declaring a general action to perform without a specific component that should fulfill it. In this way, multiple files can be attached to the Intent and any application the user has installed that can handle binary files is able to receive them. Figure 13 shows the app chooser that allows the user to pick where the files will be sent. This includes the user's email programs and cloud storage applications. The code also includes a small fix for sending a single file to Google Drive, because the Google Drive file uploader has a bug causing it to replace the file name with the subject that would be used if the file was being sent by email.

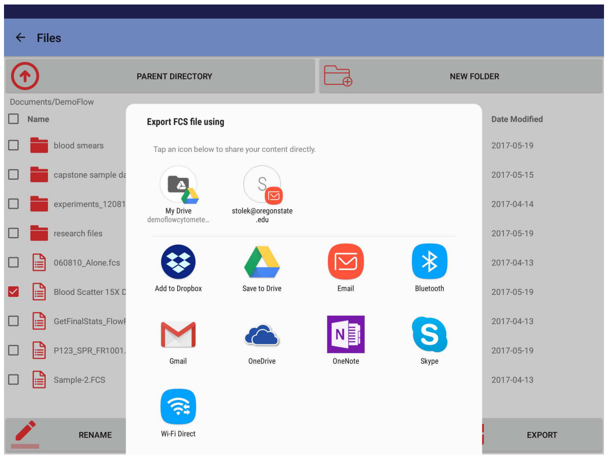


Figure 13: Application chooser showing options for exporting selected FCS files

5 FCS PARSING PACKAGE

The ability to load and export experiment data in the Flow Cytometry Standard (FCS) file format was a core application requirement. FCS is the industry standard for storing flow cytometry experiment data. It is produced by the International Society for Advancement of Cytometry (ISAC), which publishes a document describing the format in concise detail. Using that normative reference document and sample FCS files, I created an FCS parsing package in Java that implements most features of the standard. See [10] for detailed information on the FCS format.

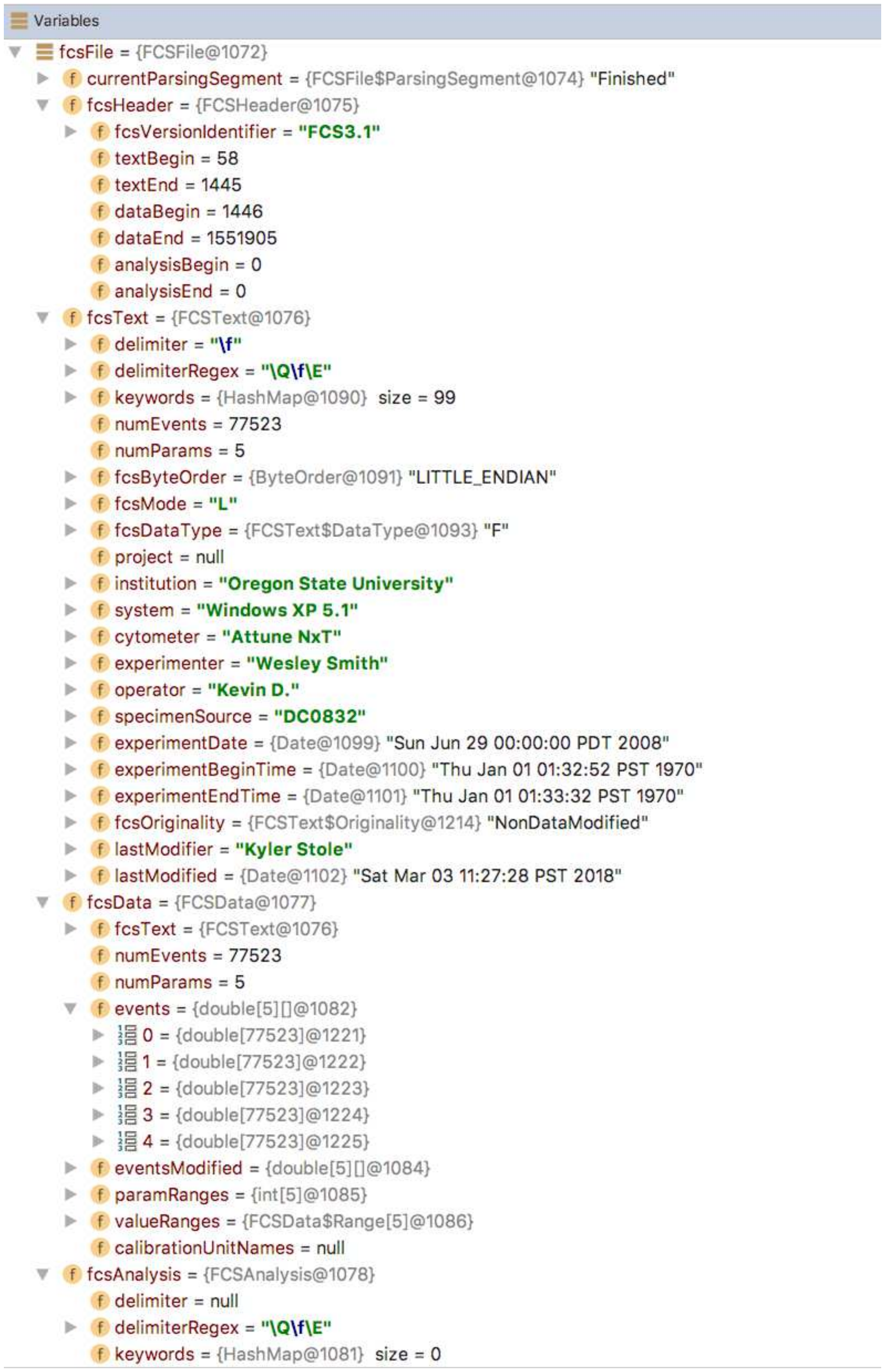
The FCS parsing package was originally developed in a standalone environment using the IntelliJ IDEA IDE and later integrated into the DemoFlow Android project. The original package was built for Java 1.8, and uses some language features that were added since Java 1.7, which is the supported Java version for Android development. Code that used unsupported features had to be modified for inclusion in the DemoFlow Android project. However, the original `FCSParser` package has been retained to leverage those features when Android adds support for Java 1.8. Since the FCS parsing code is contained in the `fcsparser` package, it is kept logically separate from the rest of the Android application code, which uses the package as if it were an external library.

The FCS parsing package supports nearly all FCS capabilities. Most importantly, it can:

- read all keywords
- read/write integer and both single- and double-precision floating point data
- read integer data with arbitrary bit sizes (i.e. not divisible by a byte)
- apply linear and logarithmic amplification
- apply fluorescence compensation
- apply calibration parameters
- support both big-endian and little-endian byte orders

The only features that are missing are relatively minor ones and most are not required for a minimum viable product. The package does not currently compute a CRC value when writing FCS files, which is ideally placed in the last 16 bits of valid documents.

The FCS parsing package has a class structure that mimics the storage structure of an FCS file. The `FCSFile` class represents an entire FCS file and must be the starting point for reading or writing an FCS file. The other four classes in the package represent distinct segments of the FCS file. The `FCSHeader` class represents the HEADER segment of the FCS file and does not expose any methods. The `FCSText` class represents the TEXT segment of the FCS file, which is parsed into a key-value store. This class also provides some helper methods and dedicated accessors for expected keys. The `FCSData` class manages the actual experiment data and is therefore the core of functionality in the package. The DATA segment stores binary data, which may be double or single-precision floating point, or integer data with potentially variable bit length. The ANALYSIS segment is rarely used, but the `FCSAnalysis` class exists to store key-value pairs from this



```
Variables
▼ fcsFile = {FCSFile@1072}
    ▶ currentParsingSegment = {FCSFile$ParsingSegment@1074} "Finished"
    ▼ fcsHeader = {FCSHeader@1075}
        ▶ fcsVersionIdentifier = "FCS3.1"
          textBegin = 58
          textEnd = 1445
          dataBegin = 1446
          dataEnd = 1551905
          analysisBegin = 0
          analysisEnd = 0
    ▼ fcsText = {FCSText@1076}
        ▶ delimiter = "\f"
        ▶ delimiterRegex = "\Q\f\E"
        ▶ keywords = {HashMap@1090}  size = 99
          numEvents = 77523
          numParams = 5
        ▶ fcsByteOrder = {ByteOrder@1091} "LITTLE_ENDIAN"
        ▶ fcsMode = "L"
        ▶ fcsDataType = {FCSText$DataType@1093} "F"
          project = null
        ▶ institution = "Oregon State University"
        ▶ system = "Windows XP 5.1"
        ▶ cytometer = "Attune NxT"
        ▶ experimenter = "Wesley Smith"
        ▶ operator = "Kevin D."
        ▶ specimenSource = "DC0832"
        ▶ experimentDate = {Date@1099} "Sun Jun 29 00:00:00 PDT 2008"
        ▶ experimentBeginTime = {Date@1100} "Thu Jan 01 01:32:52 PST 1970"
        ▶ experimentEndTime = {Date@1101} "Thu Jan 01 01:33:32 PST 1970"
        ▶ fcsOriginality = {FCSText$Originality@1214} "NonDataModified"
        ▶ lastModifier = "Kyler Stole"
        ▶ lastModified = {Date@1102} "Sat Mar 03 11:27:28 PST 2018"
    ▼ fcsData = {FCSData@1077}
        ▶ fcsText = {FCSText@1076}
          numEvents = 77523
          numParams = 5
        ▼ events = {double[5][]@1082}
            ▶ 0 = {double[77523]@1221}
            ▶ 1 = {double[77523]@1222}
            ▶ 2 = {double[77523]@1223}
            ▶ 3 = {double[77523]@1224}
            ▶ 4 = {double[77523]@1225}
        ▶ eventsModified = {double[5][]@1084}
        ▶ paramRanges = {int[5]@1085}
        ▶ valueRanges = {FCSData$Range[5]@1086}
          calibrationUnitNames = null
    ▼ fcsAnalysis = {FCSAnalysis@1078}
          delimiter = null
        ▶ delimiterRegex = "\Q\f\E"
          keywords = {HashMap@1081}  size = 0
```

Figure 14: Variables panel in the IntelliJ IDEA IDE showing an internal representation of an FCS file parsed by the FCS parsing package

segment. Figure 14 demonstrates the structure of a sample FCS file that has been read into internal memory.

Event data are retrieved from the instance of `FCSData` contained in an `FCSFile` object. From that instance, the `getNumEvents()` and `getNumParams()` methods are used to retrieve the total number of events that were recorded in the experiment and the number of parameters (i.e. channels) that are available. Then, `eventsForParamIndex(index)` is used to retrieve an array of the events for a particular index. This provides all of the event data, which is the key component of an FCS file.

5.1 FCSFile

When loading an FCS file, the package begins with the `FCSFile` class. Calling the `readFromFile(file)` method of the `FCSFile` class will load the provided `File`'s contents and parse them. It parses the segments in order, updating a state variable as it goes along. If a parsing error occurs, the `currentParsingSegment` variable will indicate which segment encountered an issue. The HEADER segment must be parsed first because it stores the lengths of the other segments. The TEXT segment must be parsed before the DATA segment because it defines the bit structure of the DATA segment.

Storing an FCS file also uses this class. This assumes that the experiment information, event data, and potentially analysis values have been stored appropriately in the appropriate members of an `FCSFile` object. The instance method `writeToFile(file)` is then used to write the segments to the provided file. Correct padding is calculated automatically and a binary representation of each segment is retrieved using each segment's `toByteArray()` method.

5.2 FCSHeader

The HEADER segment stores the begin and end locations of the other segments. It is the simplest of all the segments. When parsing a file, the `FCSHeader` class reads a fixed number of bytes that form an ASCII-encoded string with the starting and ending positions of the other segments, given as an offset from the start of the file.

5.3 FCSText

The TEXT segment is ASCII-encoded and contains key-value pairs that store information about the experiment and the data format. The first character is the ASCII delimiter character, which is used in both the TEXT and ANALYSIS segments. Since the delimiter can be any character, including an unprintable character, the `FCSText` class treats it specially by generating a regular expression with the character quoted as a literal using `Pattern.quote(string)`. This is then used to split the rest of the TEXT segment data into an array of strings, which are then split into keys and values and stored in a

`Map`. Special care also has to be taken with the keyword values, which are encoded in UTF-8.

When the map of keywords has been built, some additional interpretation is performed with some of the values. Time and date strings are parsed for the experiment date and time and the last modification date of the file. Most importantly, the byte order of event data is interpreted from the `$BYTEORD` keyword and the data type and mode are interpreted from the `$DATATYPE` and `$MODE` keywords.

5.4 FCSData

The DATA segment contains the raw experiment data in a continuous bit stream, which may formatted as unsigned integer, floating-point, double-precision floating-point, or ASCII data. ASCII is deprecated in FCS 3.1 and this FCS package does not support it. Parsing floating-point data is simple due to its fixed bit length; a single-precision IEEE floating-point number occupies $32\,\mathrm{bits}$ while a double-precision number takes twice the size at $64\,\mathrm{bits}$. The only caveat is the need to follow the byte order specified in the TEXT segment. Listing 9 and Listing 10 in Appendix A.4 show how Java helps to enforce the endianness of the data while reading from the DATA segment.

Reading integer data is more complicated because bit lengths can vary and may not be byte-divisible. A series of keywords from the TEXT segment give the bit lengths allocated for storage of each parameter and the actual range each parameter can take. This means that each integer value in the data may not align with the typical byte-by-byte reading of data and bit masks may be necessary to read data correctly. However, this enables bit compression of the data for smaller file sizes. For example, if an FCS file were produced by a flow cytometer with an ADC resolution of $12\,\mathrm{bits}$, each value could be stored in $1.5\,\mathrm{bytes}$ rather than a round $2\,\mathrm{bytes}$ and the values could be packed together, saving $0.5\,\mathrm{bytes}$ for each value (25% overall). Listing 8 in Appendix A.4 shows how integer data are read along with some of the helper functions for working with bit compressed data.

When the events have been loaded, there are a series of transformations that may be necessary depending on the keywords from the TEXT segment. First, the minimum and maximum values need to be calculated for each parameter. Integer data starts at 0 and goes to a maximum provided by the range value from the TEXT segment. For floating-point data, the minimum and maximum values are found by searching the data set. Next, linear or logarithmic amplification may be applied to integer data. Amplification allows for digital gain without affecting the original data set, so DemoFlow's FCS package stores the modified events in a new array. The type and amount of amplification is calculated from a keyword value in the TEXT segment. The same amplification must be applied to each value for each parameter of every event as well as for the minimum and maximum values.

The last two data transformations may be important in some cases but are not as integral to the basic use of FCS. Pre-calculated compensation may be applied to the data based on a spillover matrix stored in the TEXT segment. DemoFlow's FCS package can apply this compensation (storing it in the array for modified events) but does not support manipulating a spillover matrix for the DemoFlow flow cytometer. The last transformation calculates calibration parameters, such as MESF. Similar to compensation, this feature is read only and not fully supported.

5.5 FCSAnalysis

The ANALYSIS segment contains a set of key-value pairs like the TEXT segment. Unlike the TEXT segment, it is rarely used and is often zero length. The `FCSAnalysis` class uses the delimiter passed down from the `FCSText` instance and parses the segment in the same way.

6 Embedded systems

The embedded systems in the flow cytometer are responsible for connecting the laser, syringe pump, and detectors; processing events; and interfacing with the Android tablet. At the core of the embedded systems is a microcontroller unit (MCU). Following a technology review, the STM32F429I Discovery kit from STMicroelectronics was selected for development, which uses an STM32F429ZI ARM microcontroller. This makes a good development board because it comes with a lot of extra hardware components built in, such as a firmware programmer, a QVGA TFT LCD screen, and a large array of I/O pins, which are not strictly necessary but allow flexibility when prototyping. In a more developed prototype, a smaller, slightly-cheaper, less feature-rich board could be used instead. In production, a custom chip would perform better and cost less.

The other major hardware consideration was the communication medium between the MCU and the tablet. One of the benefits of using an Android tablet is its inherent portability, and that meant it made sense to connect it wirelessly to the flow cytometer. Most tablets support both Wi-Fi and Bluetooth wireless technologies. While Wi-Fi was considered for its dependability and high theoretical data transfer rates, Bluetooth was chosen to reduce overhead. Bluetooth is simpler than Wi-Fi and there are a lot of Bluetooth adapters available for embedded systems.

The other major hardware components, the laser and syringe pump, were selected by the project sponsor and ME team. Both of the selected components operate over an RS-232 line, so it was necessary to get a converter to interface with the MCU board. The full list of components is as follows:

- STM32F429I-DISCO (MCU board)
- HC-06 Bluetooth module
- MAX3232 RS232 to TTL converter
- NE-500 OEM Syringe Pump
- Coherent StingRay Laser

Figure 15 shows all pertinent hardware components. SSC, FL1, and FL2 are the photodiodes set up to measure scatter and fluorescence emission. All detectors connect to the MCU on individual channels through the board's analog-to-digital converters (ADCs). The board connects to the syringe pump and laser controller over RS-232 lines. Finally, the board is also connected to the Bluetooth module, which sends data to a tablet running the DemoFlow Android application.

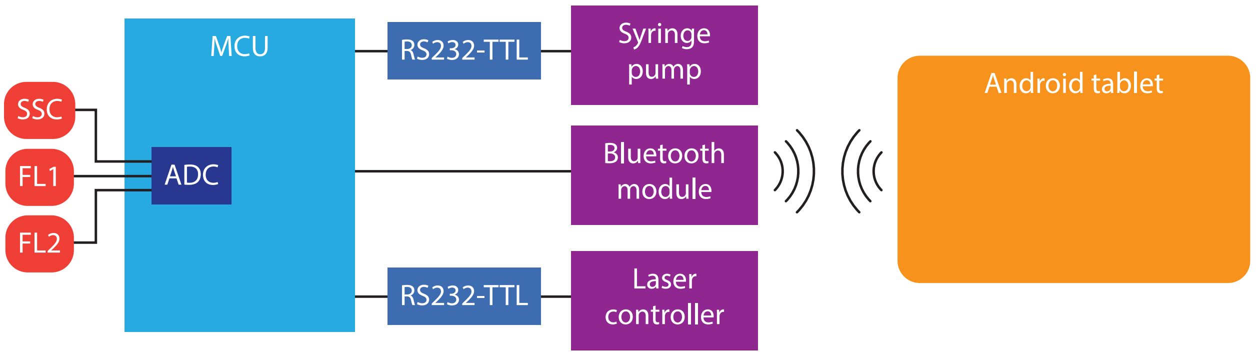


Figure 15: Block diagram showing the DemoFlow electronics hardware setup

6.1 Developing for the STM32

A considerable amount of the time was dedicated to establishing a decent environment for STM32 development. This was extremely difficult due to the density, dispersion, and poor organization of development-related documentation. ST provides several resources that relate to the Discovery board and the STM32F429ZI chip but ARM holds a lot of the documentation on programming ARM devices. During the evolution of this family of devices, no clear guides were set up to help illuminate new users to the options for programming them, which include these libraries:

- Middleware and Utilities
- Hardware Abstraction Layer (HAL) APIs
- Low-Layer (LL) drivers
- Standard Peripheral Libraries (StdPeriph)
- Cortex Microcontroller Software Interface Standard (CMSIS)
- direct register manipulation (no libraries)

I started setting up an STM32 development environment by experimenting with multiple development toolchains. I initially attempted to set up a native development environment on my local macOS laptop, but then switched to a Windows virtual machine to make use of the Windows-only STM32 development IDEs. First was the full-featured Keil MDI-ARM suite of tools. I also installed trial versions of Atollic TrueSTUDIO, TASKING, and IAR Embedded Workbench, but abandoned the Windows environment before fully testing them.

Transitioning back to native macOS, I put together a development environment using open-source tools. For a compiler and debugger, the environment uses the GNU ARM Embedded Toolchain (`arm-none-eabi-gcc/-gdb`), which is provided by ARM, available as a native installation on macOS, and provides full support for ARM compilation. To interface with the board over JTAG, the environment uses OpenOCD (the Open On-Chip Debugger). Finally, the environment also includes an open-source command line version of ST's ST-LINK tools, which are Windows-only tools for connecting to, programming, and pulling information from ST MCUs.

Following some sample code, I successfully used the open-source toolchain to run some code that made use of the STM32 Standard Peripheral Libraries. Building on the sample code, I was able to convert an analog value from the board's internal temperature sensor using one of the analog-to-digital converters (ADC) with continuous conversions and display updated values in OpenOCD using semihosting. This used two key facets of the board, as ADCs are needed for DemoFlow and semihosting allows debugging and logging using I/O functions on the host machine.

More research made clear that switching to the STM32 HAL would be beneficial because it is better supported (STMicroelectronics has stopped developing the Standard Peripheral Libraries) and there is a program, STM32CubeMX, that helps generate initialization code. STM32CubeMX is designed to generate code set up for particular toolchains, and most are for Windows. The only Mac tool that it supports is System Workbench for STM32 (SW4STM32), which is a modified version of the Eclipse IDE. To avoid the difficulties of working with SW4STM32, I found an adequate Makefile for using GNU Make to build projects using the HAL APIs. Experimenting with the STM32CubeMX software to see what it generates and comparing that with the StdPeriph programs helped me become familiar with its use.

This open-source environment works with the following commands:

- `make` - compile the project for the selected board (specified in the Makefile)
- `make openocd` - launch OpenOCD and connect (board must be attached via USB)
- `make debug` - launch GDB and load program (OpenOCD must be connected to the board)
- Running `continue` in GDB begins program execution

6.2 ADCs

One of the major considerations for the MCU was its ability to convert analog signals to digital values. The STM32F429I Discovery has three analog-to-digital converters (ADCs) that can sample multiple channels in a variety of modes. They can each achieve a resolution of up to 12 bits, which meets our requirement for ADC precision (Appendix C). As a particle flows very quickly through the interrogation point in the flow cell, traversing the laser beam, each detector needs to takes multiple samples to generate an event.

Figure 16 shows how a single particle traversing the laser beam is sampled multiple times to generate an event pulse.

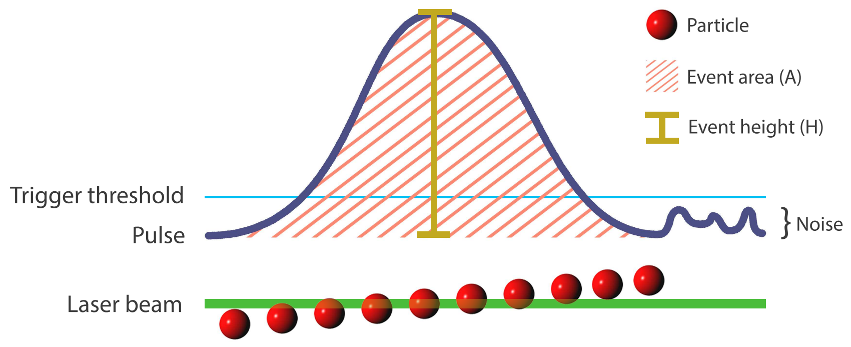


Figure 16: Single event generated from multiple samples of a single particle

To measure useful event data, the MCU's ADCs need to support a high sampling rate. With the goal of measuring particles as small as $10\,\mu\text{m}$, we can calculate the necessary ADC sampling rate given some assumptions:

- The particle should be sampled 25 - 30 times as it transits the laser beam.
- The laser beam has a diameter of $15\,\mu\text{m}$ at the interrogation point.
- The particle has a sampling distance of $30\,\mu\text{m}$.
  ($beam\ width + \frac{3}{4} particle\ size$ on each side of the laser)
- The particle will have a linear velocity of $1\,\text{m/s}$.

A particle moving at $1\,\text{m/s}$ will travel $30\,\mu\text{m}$ in $30\,\mu\text{s}$. To sample a detector 30 times as a particle transits the laser beam in $30\,\mu\text{s}$, the detector would need a sampling rate of $1\,\text{MHz}$. As a final calculation, a new particle will arrive about every $300\,\mu\text{s}$. That gives an incidence rate of about $3000\,\text{particles/s}$.

The original goal of sampling 4 separate channels at $1\,\text{MHz}$ each was theoretically achievable with a configuration that could use all of the board's 3 ADCs in triple-interleaved mode to sample 4 channels rapidly. This also required setting up the board's direct memory access (DMA) controller to move the resulting conversions into a buffer. I approached this goal incrementally by creating the following projects.

1) regular conversion
2) regular conversion + DMA
3) regular conversion + scan mode (4 channels) + DMA
4) triple-interleaved mode + DMA
5) triple-interleaved mode + 2 channels + DMA
6) triple-interleaved mode + scan mode (3 channels) + DMA
7) triple-interleaved mode + scan mode (4 channels) + DMA

Testing the code involved connecting 4 potentiometers to the 4 ADC channels as shown in Figure 17. Figure 18 shows the resulting conversions displayed in OpenOCD when running all 4 channels.

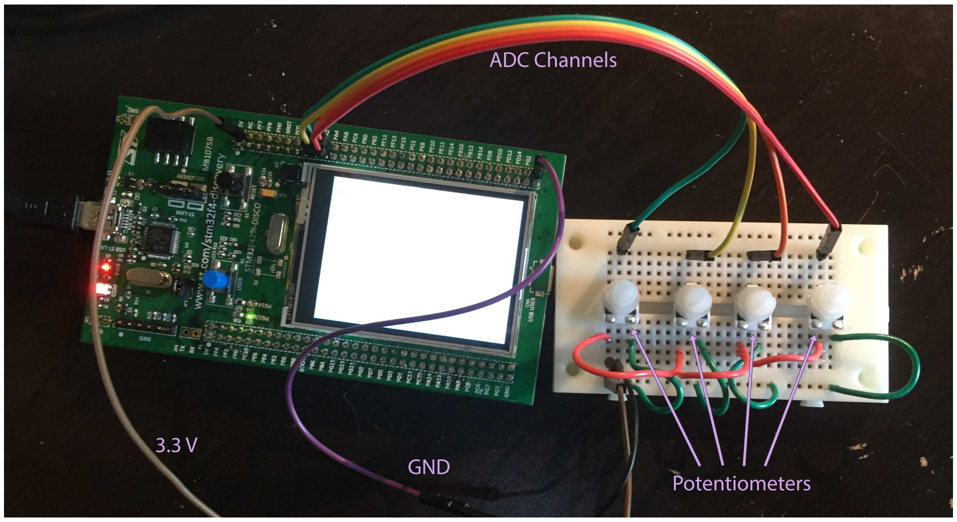
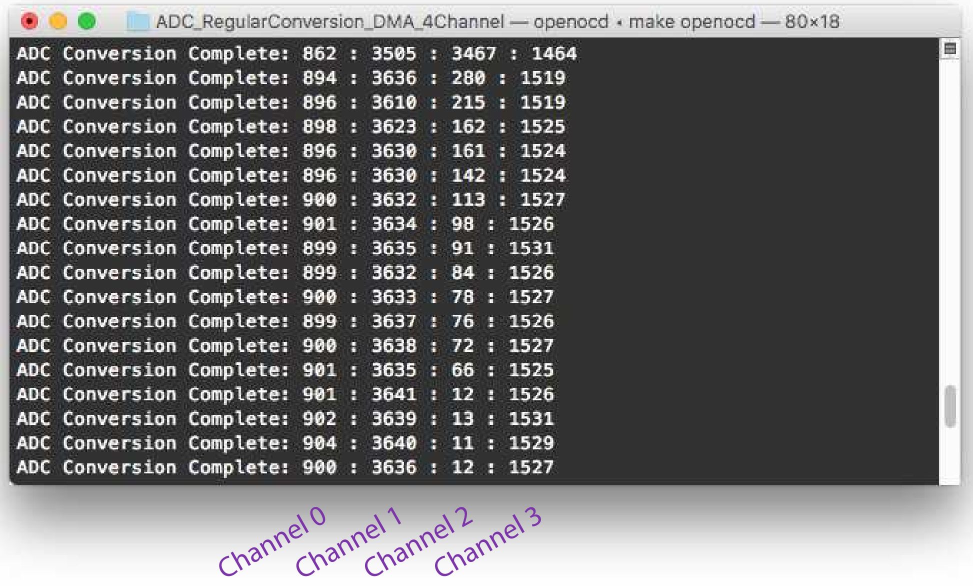


Figure 17: Testing setup with 4 ADC channels connected to 4 potentiometers


ADC_RegularConversion_DMA_4Channel — openocd ◂ make openocd — 80×18
Channel 0
Channel 1
Channel 2
Channel 3

```
ADC Conversion Complete: 862 : 3505 : 3467 : 1464
ADC Conversion Complete: 894 : 3636 : 280 : 1519
ADC Conversion Complete: 896 : 3610 : 215 : 1519
ADC Conversion Complete: 898 : 3623 : 162 : 1525
ADC Conversion Complete: 896 : 3630 : 161 : 1524
ADC Conversion Complete: 896 : 3630 : 142 : 1524
ADC Conversion Complete: 900 : 3632 : 113 : 1527
ADC Conversion Complete: 901 : 3634 : 98 : 1526
ADC Conversion Complete: 899 : 3635 : 91 : 1531
ADC Conversion Complete: 899 : 3632 : 84 : 1526
ADC Conversion Complete: 900 : 3633 : 78 : 1527
ADC Conversion Complete: 899 : 3637 : 76 : 1526
ADC Conversion Complete: 900 : 3638 : 72 : 1527
ADC Conversion Complete: 901 : 3635 : 66 : 1525
ADC Conversion Complete: 901 : 3641 : 12 : 1526
ADC Conversion Complete: 902 : 3639 : 13 : 1531
ADC Conversion Complete: 904 : 3640 : 11 : 1529
ADC Conversion Complete: 900 : 3636 : 12 : 1527
```

Figure 18: ADC conversions of 4 potentiometers displayed in OpenOCD

6.3 Event processing

The algorithm to sample sensors and convert sample values into height and area data provides the core of data acquisition within the flow cytometer. Algorithm 1 shows how events are triggered and sensor sampling data is converted into height (H) and area (A) data. This algorithm works by sampling all detectors every $1\,\mu s$ and storing the samples in queues. One detector (`sensor1` in the algorithm) acts as an event trigger for all detectors. Although all detectors are continuously sampled, the H and A data for each detector is calculated and sent to the Android tablet only when the trigger detector registers an event.

Algorithm 1 Detector sampling data to H & A event data

```
Require: $threshold$
Require: $weight$
Require: Each $sensor$ has a circular queue $samples$
  for $i = 0$ to $1000$ do
    Sample $sensor1$
  end for
  $zero \leftarrow Avg(sensor1.samples[0:1000])$
  Turn on laser
  loop
    Wait for $timer$ to expire
    Set $timer$ for $1\,\mu s$
    for each $sensor$ in $sensors$ do
      Sample $sensor$
      Add $sample$ to $sensor.samples$
    end for
    if $sensor1.samples[current] > (threshold - zero)$ then
      $sampleCount \leftarrow sampleCount + 1$
    else if $sampleCount < 3$ or $sampleCount > 100$ then
      $sampleCount \leftarrow 0$
      $zero \leftarrow zero + sensor1.samples[current]/weight$
    else
      for each $sensor$ in $sensors$ do
        $H \leftarrow Max(sensor.samples[sampleCount : current])$
        $A \leftarrow Sum(sensor.samples[sampleCount : current])$
        Send $(H, A)$
      end for
    end if
  end loop
```

The actual implementation of this event processing algorithm is shown in Listing 3 of Appendix A.1. Due to design difficulties for the ME team, the device is limited to 3 detectors. This was a change from the original requirements that allows the 3 detectors to be sampled synchronously using the 3 ADCs available on the MCU board, thus, circumventing synchronization issues. This firmware was tested at Thermo Fisher's Eugene campus with a signal provided by a waveform generator. Testing confirmed that events were being triggered properly (events must consist of at least a few samples but not too many). However, experiments indicated that the MCU was only reaching $0.5\,\mathrm{MHz}$

sampling of each channel. Our theory is that the data sheet arrives at its 2.4 MSPS number by considering ADC readings over multiple channels with sampling occurring in a pipelined fashion. Although this does not match the sampling requirement that was originally described, the laser beam height is a bit taller than the number originally reported by the ME team, so this lower sampling rate should be sufficient for recording useful data.

6.4 Bluetooth

Bluetooth is the chosen connection mechanism for communication between the MCU and the tablet. The HC-06 and HC-05 Bluetooth modules connect to the board through UART. When new data is available, a callback function is triggered:

```
UART_HandleTypeDef huart1; // previously set up

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
  // Receive newly-available Bluetooth data
  HAL_UART_Receive_IT(&huart1, (uint8_t*) &receiveBuffer, 1);
  // Send data through Bluetooth
  HAL_UART_Transmit(&huart1, (uint8_t*) &sendBuffer, 6, 10000);
}
```

To demonstrate the Bluetooth connection between the MCU and the Android tablet, I set up a simple command interface. A program on the MCU accepts ASCII data over Bluetooth, which triggers LEDs on the board and a response sent back over Bluetooth. On the tablet, Android Bluetooth APIs help send the ASCII data. This demonstrates the basic Bluetooth needs, but this connection has not yet been integrated into the DemoFlow Android project or the event processing algorithm on the MCU.

6.5 System controls

Following the completion of the ME development, we now have specifics on the hardware controls that must be provided by the MCU. DemoFlow is set up with an NE-500 syringe pump from New Era Pump Systems and a Coherent StingRay Laser, both of which take commands on an RS-232 line. The MCU will communicate with these components over RS-232 from its UARTs using an RS232-to-TTL converter. Commands from the MCU to the syringe pump and laser are triggered by commands from the tablet sent over Bluetooth. This part of the Android application is not yet implemented. Communication between the MCU and the two external systems has not yet been established successfully. Control of the laser and syringe pump from the Android tablet is currently in progress, and in the meantime, it can be provided by a laptop.

7 Effect of low-cost flow cytometry

The wide array of applications for flow cytometry is undeniable. New applications for the technology are added frequently as the price drops and it overtakes older technologies to become the de facto instrument in more fields. For a long time, flow cytometers have been large instruments, relegated to high-end research laboratories, but the development of lower-cost benchtop devices and cost-effective assays has transformed them into standard clinical equipment [1], [5].

For a large population, however, the technology remains inaccessible due to its cost and complexity. The potential for flow cytometry to save lives from diseases in resource-limited settings justifies the development of lower-cost instruments and more accessible assays. In the following two sections, this paper will explore how flow cytometry can be used to combat two notable diseases of poverty: HIV and malaria. The use of flow cytometry to monitor HIV-positive people and to diagnose malaria could be crucial to bringing better quality of life to epidemic areas.

7.1 Lowering the cost of HIV/AIDS detection and monitoring

The human immunodeficiency virus (HIV) is a lentivirus (a genus of retrovirus) [13] that targets the human immune system, making infected people susceptible to opportunistic infections or cancers. According to the World Health Organization (WHO), 1 million people died from HIV-related causes in 2016 [14]. There is no effective cure for HIV; once someone is HIV-positive, they will remain that way for the rest of their life [15]. It can, however, be treated with antiretroviral therapy (ART), which can dramatically prolong the lives of those infected [16]. In addition to starting ART, awareness of the disease through diagnosis allows infected people to take preventative measures against further transmission of the virus. The WHO estimates that only 70% of people with HIV know they have it, and an additional 7.5 million people would need access to HIV testing services to reach the target of 90% awareness [14].

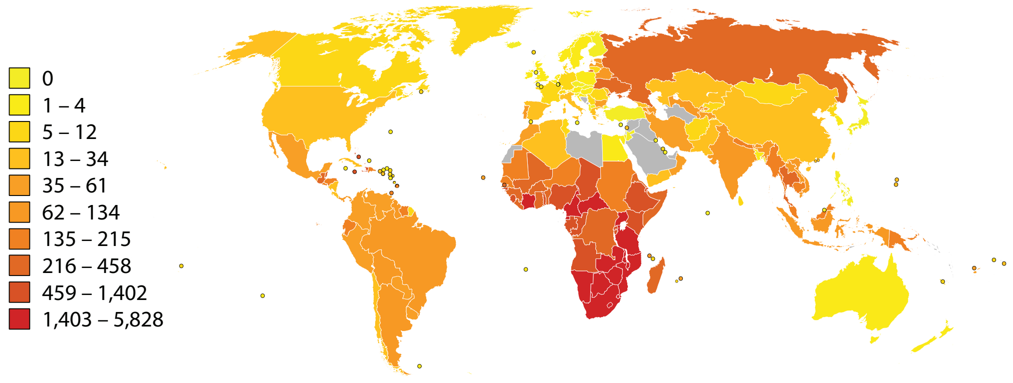


Figure 19: Deaths due to HIV/AIDS per million persons in 2012

Figure 19 shows estimated deaths due to HIV/AIDS. The virus is thought to have originated in chimpanzees as simian immunodeficiency viruses (SIVs) and then spread out in the vicinity of the Congo [17]. The infection is now most prevalent in countries of southern Africa, where the virus has continued to proliferate and access to clinical diagnostics and treatment is more limited. Given the importance of HIV testing and monitoring for deciding when to start treatment and raising awareness of the disease, access to low-cost diagnostics is crucial [18].

The main targets of HIV are the body's CD4+ T-cells, also known as T helper cells or CD4+ lymphocytes [16], [19], [20]. This is a type of white blood cell responsible for alerting the body's immune system to the presence of potentially intrusive viruses and bacteria. HIV infects one of these cells by presenting a GP120 protein that binds onto the cell's CD4 receptor along with a CCR5 and/or CXCR4 co-receptor that is also present on the cell [19], [21], [22]. As the immune system detects some HIV-infected cells, it destroys them, causing chain reactions that kill uninfected CD4 cells [19].

While there are rapid HIV tests, which can detect infection after only a few weeks and take about 30 minutes, proper treatment requires persistent monitoring through CD4 counts [16], [18]. A CD4 count measures the levels of CD4 cells in the bloodstream, serving as an indicator of the health of the immune system. The CD4 count of an uninfected person ranges between 500 and $1200\,\mathrm{cells/\mu L}$. In the first stage of infection, the acute HIV infection stage, the CD4 count of the infected person drops dramatically and they are at a greater risk for transmitting the disease due to a high viral load in the body [19], [22]. This is followed by a prolonged stage of clinical latency during which the CD4 count slowly rises as the body comes to term with the infection [21]. It is important for infected people to be diagnosed and begin treatment in the early stages because they are most at risk for transmitting the disease and beginning ART helps extend the clinical latency stage. If the CD4 count falls below $200\,\mathrm{cells/\mu L}$, the infected person has developed a stage 3 HIV infection, acquired immune deficiency syndrome (AIDS), at which point they are at risk for more and unusual infections [19], [22]. In addition to monitoring CD4 counts to assess treatment efficacy, infected people need access to CD4 counts to observe the overall trend in their CD4 count since it can fluctuate between times of day and with other factors. When starting ART, recurring CD4 counts should occur every three to six months, although testing frequency may be reduced over time [18].

Compared to other potential methods for CD4 counts, flow cytometers are ideal as they provide high accuracy, high throughput, and can measure CD4 percentage in addition to absolute counts [23]. One popular method is to look for CD45+CD3+CD4+ cells, as these three markers identify relevant CD4+ lymphocytes (where CD45 and CD3 markers are used to exclude other CD4+ cells such as monocytes) [16]. This could potentially be simplified to look just for CD3+CD4+ markers, which would lower the cost of testing. [20] provides an example from Thailand of how lowering the cost of CD4 enumeration could ease the burden on a country's health care system. The potential for lower cost assays for performing CD4 counts such as the Blantyre count single platform method may help alleviate some of the difficulty in using flow cytometry in poor regions [16], [18], [23].

One of the complications with using flow cytometry in HIV monitoring is that it is a continual process. It is not as simple as providing a single, accurate diagnosis and then moving on. Since HIV patients need to monitor their immune systems over time, they need continued access to flow cytometry technology. Additionally, since results vary and are somewhat relative, the patient should ideally be monitoring their HIV infection with the same equipment every time. In some cases, this may be an additional strain on the goal of increased access to flow cytometry because it is simply one more constraint to take into account when providing access to flow cytometry.

7.2 Replacing traditional microscopy techniques for malaria diagnostics

Malaria is an infectious disease that causes symptoms including fever, chills, tiredness, vomiting, headaches, and other flu-like symptoms. If left untreated, the disease can worsen, causing severe complications and potentially death. However, death is quite preventable with proper early diagnosis and prompt treatment. The CDC estimates that in 2016, there were 216 million malaria cases worldwide with 445,000 deaths [24], although estimation of malaria infection is rather imprecise. Figure 20 shows estimated deaths due to malaria, which are concentrated in the regions of Africa and South/Southeast Asia. Although the majority of malaria cases in the United States, Europe, and other regions outside of the tropical and subtropical regions come from travelers returning from those countries where malaria transmission does occur, these cases are generally treated with relative ease.

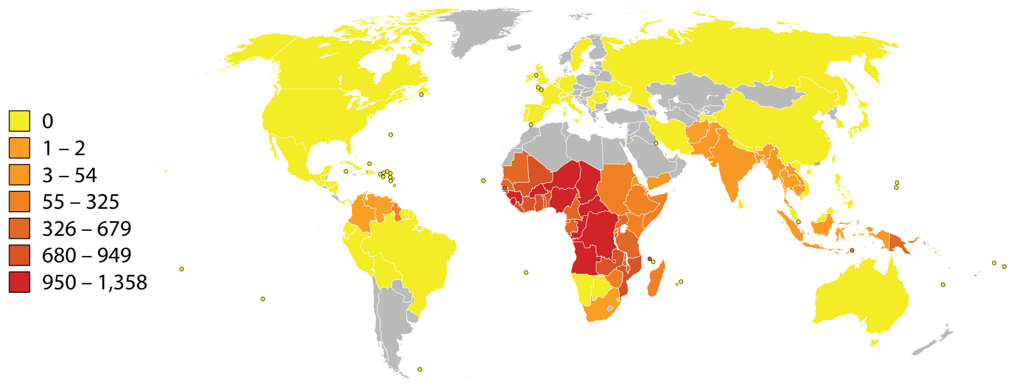


Figure 20: Deaths due to malaria per million persons in 2012

Malaria is a mosquito-borne disease caused by parasitic protozoans of the genus Plasmodium: *P. falciparum*, *P. vivax*, *P. ovale*, *P. malariae*, and *P. knowlesi* [24]. Modern medicine is able to treat malaria without much difficulty, but diagnosis still relies on outdated diagnostic technologies. One reason for the uncertain estimation of malaria deaths per year is the high amount of presumptive diagnoses made without parasitemia (demonstrable presence of parasites in a patient's blood). Diagnoses based on symptoms and the cyclic nature of symptoms in malaria patients can be misleading and lead to improper treatment [25]. There have been several diagnostic methods developed up to this point. The first was a stain developed over several decades and finalized by Gustav Giemsa in 1904 [26]. Others generally involve microscopy with trained observers detecting subtle morphological features of parasitized red blood cells in low concentration [26].

Flow cytometry can be used to measure cellular constituents such as DNA, RNA, and the malaria pigment hemozoin (Hz). Compared to microscopy techniques, this can provide more reliable identification of plasmodia species and insight into their developmental stages [26]. It is also more capable of detecting low-density populations, as is the case with malaria. While flow cytometry is a versatile approach to malaria detection, in the future it may be aided by a specialized microfluidic cell deformability sensor to provide ”label-free, rapid and cost-effective parasitemia quantification and stage determination” in remote regions [27]. For now, however, flow cytometry promises better results in malaria diagnostics.

Although I am not working on novel methods for diagnosing malaria nor lowering the cost of assays for malaria diagnosis, low-cost flow cytometry represents an initial step towards more available and reliable identification of malaria. This is necessary for proper treatment. Flow cytometry techniques would also do a better job of tracking the stage and progression of the disease during continued treatment. With malaria, parasitemia quantification is necessary for effective treatment. Accurate parasite identification and measurements of parasite density show the seriousness of the disease and reveal the efficacy of the treatment path.

8 Conclusions and future work

From its inception, DemoFlow was an ambitious project due to the many interconnected parts necessary to bring it together. Requirements evolved as the ME team encountered hurdles and the lack of a dedicated electrical team made this project especially daunting. In spite of this, DemoFlow provides a prototype that sets the stage for simplified, low-cost flow cytometry. Although the final cost is highly contingent on the ME team’s work (the laser, syringe pump and tablet cost under $2000 and the other electrical components cost about $100 collectively), the entire project costs somewhere in the vicinity of $5000. Its design demonstrates how a relatively simple group of components can be put together to form a useful and understandable flow cytometer at reasonable cost. The accompanying Android application, especially, provides functionality in a standalone package, which is novel for its portability and touch interface. The largest part that remains incomplete is the data recording component, which relies on a connection between the app and the embedded hardware. Although most of the pieces for data collection and recording are in place — the event processing algorithm on the MCU, the Bluetooth connection with the tablet, the events data structure in the Android app — they have not all been tied together.

Several programs for the embedded hardware now need to be merged to create the core firmware that will run within the flow cytometer. Event processing forms the core of this firmware, but it needs to be connected with Bluetooth for sending data to the tablet and it needs to interface with the syringe pump and laser controller based on commands received over Bluetooth. A final feature that could still be applied to the embedded systems, which was mentioned as an optional requirement in Appendix C, is instrument status feedback. With these features in place, the embedded hardware could join the work from the ME team to form the DemoFlow flow cytometer.

While the Android application is close to fully-featured, it could still see changes. The Bluetooth component for communicating with the tablet needs to be fully integrated to complete data collection. Several small usability improvements could be made on the Files screen and the Plots screen. Internally, several stability improvements are needed in the part of the app between the Plots screen and the Gating screen. These improvements relate to how charts and gates are stored so that they may be transferred between the two screens and persist when activities are destroyed and recreated. The last overarching aspect of the Android application that is missing is related to performance. Several long-running tasks such as loading FCS files, rendering charts, and manipulating gates should be modified to run asynchronously so that they do not slow down the main UI thread.

As the above case studies (Section 7 demonstrate, flow cytometry is a powerful technology that can impact the world as it becomes more accessible. That means lowering the cost of flow cytometers and making them easier to use so that a wider audience of people may benefit from the technology. With more projects like DemoFlow, we can truly democratize the technology to benefit everyone.

References

[1] History of Flow Cytometry. Beckman Coulter Life Sciences. [Online]. Available: https://www.beckman.com/resources/discover/fundamentals/history-of-flow-cytometry

[2] B. R. Glick, T. L. Delovitch, and C. L. Patten, *Medical Biotechnology*. American Society of Microbiology, Jan. 2014. [Online]. Available: http://www.asmscience.org/content/book/10.1128/9781555818890

[3] A. R. Boyd, T. S. Gunasekera, P. V. Attfield, K. Simic, S. F. Vincent, and D. A. Veal, "A flow-cytometric method for determination of yeast viability and cell number in a brewery," *FEMS yeast research*, vol. 3, no. 1, pp. 11–16, Mar. 2003.

[4] J. Vives-Rego, P. Lebaron, and G. Nebe-von Caron, "Current and future applications of flow cytometry in aquatic microbiology," *FEMS Microbiology Reviews*, vol. 24, no. 4, pp. 429–448, Oct. 2000. [Online]. Available: https://academic.oup.com/femsre/article/24/4/429/510325

[5] M. Brown and C. Wittwer, "Flow cytometry: principles and clinical applications in hematology," *Clinical Chemistry*, vol. 46, no. 8 Pt 2, pp. 1221–1229, Aug. 2000.

[6] J. Picot, C. L. Guerin, C. Le Van Kim, and C. M. Boulanger, "Flow cytometry: retrospective, fundamentals and recent instrumentation," *Cytotechnology*, vol. 64, no. 2, pp. 109–130, Mar. 2012. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3279584/

[7] J. P. Roberts, "Cell Analysis at the Bench: Benchtop Flow Cytometers," Sep. 2013. [Online]. Available: https://www.biocompare.com/Editorial-Articles/146008-Cell-Analysis-at-the-Bench-Benchtop-Flow-Cytometers/

[8] M. G. Ormerod. (2008) Flow cytometry - a basic introduction. [Online]. Available: http://flowbook.denovosoftware.com/chapter-2-flow-cytometer

[9] J. Spidlen, W. Moore, D. Parks, K. Y. Goldberg, C. Bray, P. Bierre, P. Gorombey, B. Hyun, M. Hubbard, S. Lange, R. Lefebvre, R. C. Leif, D. Novo, L. J. Ostruszka, A. S. Treister, J. Wood, R. F. Murphy, M. Roederer, D. Sudar, R. Zigon, and R. R. Brinkman, "Data File Standard for Flow Cytometry, version FCS 3.1." *Cytometry. Part A : the journal of the International Society for Analytical Cytology*, vol. 77, no. 1, pp. 97–100, 2010.

[10] *Data File Standard for Flow Cytometry*, International Society for Advancement of Cytometry Std. FCS 3.1, 2008.

[11] D. W. Scott, "Averaged Shifted Histogram," *WIREs Comput. Stat.*, vol. 2, no. 2, pp. 160–164, Mar. 2010. [Online]. Available: https://doi.org/10.1002/wics.54

[12] D. Sunday, “Inclusion of a Point in a Polygon,” 2012. [Online]. Available: http://geomalgorithms.com/a03-_inclusion.html

[13] R. A. Weiss, “How Does HIV Cause AIDS?” *Science*, vol. 260, no. 5112, pp. 1273–1279, 1993. [Online]. Available: http://www.jstor.org.ezproxy.proxy.library.oregonstate.edu/stable/2881758

[14] “WHO | HIV/AIDS.” [Online]. Available: http://www.who.int/mediacentre/factsheets/fs360/en/

[15] “HIV/AIDS | CDC,” Feb. 2018. [Online]. Available: https://www.cdc.gov/hiv/default.html

[16] P. Balakrishnan, S. Solomon, N. Kumarasamy, and K. H. Mayer, “Low-cost monitoring of HIV infected individuals on highly active antiretroviral therapy (HAART) in developing countries,” *The Indian Journal of Medical Research*, vol. 121, no. 4, pp. 345–355, Apr. 2005.

[17] N. R. Faria, A. Rambaut, M. A. Suchard, G. Baele, T. Bedford, M. J. Ward, A. J. Tatem, J. D. Sousa, N. Arinaminpathy, J. Ppin, D. Posada, M. Peeters, O. G. Pybus, and P. Lemey, “The early spread and epidemic ignition of HIV-1 in human populations,” *Science (New York, N.Y.)*, vol. 346, no. 6205, pp. 56–61, Oct. 2014. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4254776/

[18] P. Balakrishnan, H. S. Iqbal, S. Shanmugham, J. Mohanakrishnan, S. S. Solomon, K. H. Mayer, and S. Solomon, “Low-cost assays for monitoring HIV infected individuals in resource-limited settings,” *The Indian Journal of Medical Research*, vol. 134, no. 6, pp. 823–834, Dec. 2011. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3284092/

[19] S. L. Gorbach, J. G. Bartlett, and N. R. Blacklow, *Infectious Diseases*. Philadelphia, PA: LWW (PE), 2001. [Online]. Available: http://ebookcentral.proquest.com/lib/osu/detail.action?docID=3418770

[20] K. Pattanapanyasat, S. Lerdwana, H. Shain, E. Noulsri, C. Thepthai, V. Prasertsilpa, A. Eksaengsri, and K. Kraisintu, “Low-cost CD4 enumeration in HIV-infected patients in Thailand,” *Asian Pacific Journal of Allergy and Immunology*, vol. 21, no. 2, pp. 105–113, Jun. 2003.

[21] W. Kirch, “HIV-Infection and AIDS,” in *Encyclopedia of Public Health: Volume 1: A - H Volume 2: I - Z*. Springer Science & Business Media, Jun. 2008, pp. 676–680.

[22] Annenberg Foundation, Oregon Public Broadcasting, and Annenberg/CPB, “HIV and AIDS,” in *Rediscovering Biology: Molecular to Global Perspectives*. S. Burlington, VT: Annenberg Foundation, 2003, oCLC: 57306379.

[23] C. A. MacLennan, M. K. P. Liu, S. A. White, J. J. G. v. Oosterhout, F. Simukonda, J. Bwanali, M. J. Moore, E. E. Zijlstra, M. T. Drayson, and M. E. Molyneux, “Diagnostic accuracy and clinical utility of a simplified low cost method of counting CD4 cells with flow cytometry in Malawi: diagnostic accuracy study,” *BMJ*, vol. 335, no. 7612, pp. 190–190, Jul. 2007. [Online]. Available: http://www.bmj.com/content/335/7612/190

[24] “CDC - Malaria,” Jan. 2018. [Online]. Available: https://www.cdc.gov/malaria/

[25] W. Kirch, “Malaria,” in *Encyclopedia of Public Health: Volume 1: A - H Volume 2: I - Z*. Springer Science & Business Media, Jun. 2008, pp. 872–867.

[26] H. M. Shapiro, S. H. Apte, G. M. Chojnowski, T. Hnscheid, M. Rebelo, and B. T. Grimberg, “Cytometry in Malaria–A Practical Replacement for Microscopy?” in *Current Protocols in Cytometry*. John Wiley & Sons, Inc., Jul. 2013, dOI: 10.1002/0471142956.cy1120s65. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/0471142956.cy1120s65/abstract

[27] X. Yang, Z. Chen, J. Miao, L. Cui, and W. Guan, “High-throughput and label-free parasitemia quantification and stage differentiation for malaria-infected red blood cells,” *Biosensors & Bioelectronics*, vol. 98, pp. 408–414, Dec. 2017.

APPENDIX A
ESSENTIAL CODE LISTINGS

A.1 Embedded systems

Listing 3: Event processing algorithm. Run when a new set of conversions is available.

```c
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* AdcHandle) {
  if (waitCounter < 1000000) { // Delay while board is initialized
    waitCounter++;
  } else if (counter < 100000) { // Take 100000 samples to establish zero
    counter++;
    total += convValues[0];
  } else if (counter == 100000) { // Calculate zero
    zero = total / counter++;
  } else if (convValues[0] > (threshold - zero)) {
    sampleCount++;
    uint16_t detector1Value = convValues[0] - zero;
    uint16_t detector2Value = convValues[1] - zero;
    uint16_t detector3Value = convValues[2] - zero;
    detector1Sum += detector1Value;
    detector2Sum += detector2Value;
    detector3Sum += detector3Value;
    detector1Max = MAX(detector1Max, detector1Value);
    detector2Max = MAX(detector2Max, detector2Value);
    detector3Max = MAX(detector3Max, detector3Value);
  } else {
    if (sampleCount > 3 && sampleCount < 100) { // events must have between (3, 100)
    samples
      // send max (H) and sum (A) for each detector
      printf("Sending event..\n");
      printf("# of samples: %u\n", sampleCount);
      printf("Detector1: [H: %u] [A: %lu]\n", detector1Max, detector1Sum);
      printf("Detector2: [H: %u] [A: %lu]\n", detector2Max, detector2Sum);
      printf("Detector3: [H: %u] [A: %lu]\n", detector3Max, detector3Sum);
    }
    sampleCount = 0;
    detector1Sum = detector2Sum = detector3Sum = 0;
    detector1Max = detector2Max = detector3Max = 0;
    zero = zero + convValues[0] / weight;
  }
}
```

A.2 Histogram calculations

Listing 4: Histogram binning supporting linear and logarithmic scaling.

```java
final static int NUM_BINS = 512;

final double min = (xParam.min < 1 && logScale) ? 1.0 : xParam.min;
final double max = xParam.max;

valueFormatter =
    new LinearLogarithmicAxisValueFormatter(logScale, NUM_BINS - 1, min, max);

int[] binCounts = new int[NUM_BINS]; // value counts
for (int eventIdx = 0; eventIdx < xParam.events.length; eventIdx++) {
  if (gate != null && !gate.eventKeepList.contains(eventIdx)) continue;

  double val = xParam.events[eventIdx];
  int binIdx = (int) valueFormatter.getChartValueForFCSValue(val);
  if (binIdx < 0) continue; // drop values that are off chart

  binCounts[binIdx]++;
}
```

Listing 5: Plotting averaged histograms.

```
List<BarEntry> barEntries = new ArrayList<>(NUM_BINS);

final int AVERAGING_TAIL = NUM_BINS / 70;
double runningAvgSum = 0.0;
int numValsInAvg = 0;
for (int binIdx = 0; binIdx < AVERAGING_TAIL; binIdx++) {
  runningAvgSum += binCounts[binIdx];
  numValsInAvg++;
}

for (int binIdx = 0; binIdx < NUM_BINS; binIdx++) {
  if (binIdx + AVERAGING_TAIL < NUM_BINS) {
    runningAvgSum += binCounts[binIdx + AVERAGING_TAIL];
    numValsInAvg++;
  }

  if (binIdx - 1 - AVERAGING_TAIL >= 0) {
    runningAvgSum -= binCounts[binIdx - 1 - AVERAGING_TAIL];
    numValsInAvg--;
  }

  double yValSmoothed = runningAvgSum / numValsInAvg;

  barEntries.add(new BarEntry(binIdx, yValSmoothed));
}
```

Listing 6: Custom AxisValueFormatter applied to histograms that formats the x-axis labels according to linear or logarithmic transformations.

```
public class LinearLogarithmicAxisValueFormatter implements IAxisValueFormatter {
  ...

  LinearLogarithmicAxisValueFormatter(boolean logScale, int maxBinIdx,
                                      double min, double max) {
    this.logScale = logScale;
    this.min = min;

    // for logarithmic scales
    final double factor = Math.pow(min / max, 1.0 / maxBinIdx);
    log10Factor = Math.log10(factor);
    log10Min = Math.log10(min);

    // for linear scales
    FCSRangeToBinRange = maxBinIdx / (max - min);
    BinRangeToFCSRange = 1.0 / FCSRangeToBinRange;
  }

  public double getFCSValueForChartValue(double chartValue) {
    if (logScale) {
      return Math.pow(10, log10Min - log10Factor * chartValue);
```

```java
    } else {
      return chartValue * BinRangeToFCSRange + min;
    }
  }

  public double getChartValueForFCSValue(double fcsValue) {
    if (logScale) {
      if (fcsValue < 1) return -1;
      else return (log10Min - Math.log10(fcsValue)) / log10Factor;
    } else {
      return (fcsValue - min) * FCSRangeToBinRange;
    }
  }
}
```

A.3 Gating

Listing 7: Calculating the mode of transformation in SpanGateViewOnTouchListener.

```java
private void calculateMode(float x, float y) {
  float distanceToCenterBar = Math.abs(y - polygon.get(1).y);
  float distanceToLeftBar = Math.abs(x - polygon.get(0).x);
  float distanceToRightBar = Math.abs(x - polygon.get(0).y);

  if (polygon.get(0).x < x && x < polygon.get(0).y && distanceToCenterBar < 100) {
    mode = Mode.CENTER_BAR;
  } else if (distanceToLeftBar < 100 || distanceToRightBar < 100) {
    if (distanceToLeftBar < distanceToRightBar) mode = Mode.LEFT_BAR;
    else mode = Mode.RIGHT_BAR;
  } else {
    mode = Mode.NONE;
  }
}
```

A.4 FCS parsing

Listing 8: Reading integer data from the DATA segment of an FCS file.

```java
// pre-set fields: int eventSizeInBits, int[] paramSizesInBits, int[] paramRanges
private void readIntegerDataFromData(byte[] dataSegmentData) throws ParseException {
  int bytesToRead = (numEvents * eventSizeInBits) / Byte.SIZE;

  if (dataSegmentData.length < bytesToRead)
    throw new ParseException("DATA segment is too small", 0);

  if (paramsAreByteDivisible(paramSizesInBits)) {
    int byteOffset = 0;
    ByteBuffer intBB = ByteBuffer.allocate(Integer.BYTES);
    intBB.order(fcsText.fcsByteOrder);
    for (int eventNum = 0; eventNum < numEvents; eventNum++) {
      for (int paramNum = 0; paramNum < numParams; paramNum++) {
        int paramSizeInBytes = paramSizesInBits[paramNum] / Byte.SIZE;

        intBB.clear();
        if (fcsText.fcsByteOrder == ByteOrder.BIG_ENDIAN)
          intBB.position(Integer.BYTES - paramSizeInBytes);
        intBB.put(dataSegmentData, byteOffset, paramSizeInBytes);
        intBB.flip().limit(Integer.BYTES);

        events[paramNum][eventNum] =
            intBB.getInt() & genMask(closestBasePowerForRange(paramRanges[paramNum]));
        byteOffset += paramSizeInBytes;
      }
    }
  } else {
    int bitOffset = 0;
    for (int eventNum = 0; eventNum < numEvents; eventNum++) {
      for (int paramNum = 0; paramNum < numParams; paramNum++) {
        int paramSizeInBits = paramSizesInBits[paramNum];

        int value = getBitSeqAsInt(dataSegmentData, bitOffset, paramSizeInBits);

        events[paramNum][eventNum] =
            value & genMask(closestBasePowerForRange(paramRanges[paramNum]));
        bitOffset += paramSizeInBits;
      }
    }
  }
}

/* Helper functions for reading integer data */

private static int getBitSeqAsInt(byte[] bytes, int bitOffset, int len) {
  int byteOffset = bitOffset / Byte.SIZE; // Number of full bytes from start of data
  int bitIndex = bitOffset % Byte.SIZE; // Index of first data bit in byte

  int numBytesInvolved = (bitIndex + len) / Byte.SIZE + 1;
  int rightPadding = (numBytesInvolved * Byte.SIZE) - (bitIndex + len);

  int value = 0;
  for (int byteNum = 0; byteNum < numBytesInvolved; byteNum++)
    value |= (bytes[byteOffset + byteNum]
        << (Byte.SIZE * (numBytesInvolved - 1 - byteNum)));
  value >>= rightPadding;

  return value & genMask(len);
}

private static int closestBasePowerForRange(int range) {
  return Integer.SIZE - Integer.numberOfLeadingZeros(range - 1);
}

private static int genMask(int size) {
  int mask = 0;
  while (size != 0)
    mask |= (1 << --size);
  return mask;
}
```

Listing 9: Reading single-precision floating-point data from the DATA segment of an FCS file.

```
private void readFloatDataFromData(byte[] dataSegmentData) throws ParseException {
  int bytesToRead = numEvents * numParams * Float.BYTES;

  if (dataSegmentData.length < bytesToRead)
    throw new ParseException("DATA segment is too small for data", 0);

  ByteBuffer floatData = ByteBuffer.wrap(dataSegmentData);
  floatData.order(fcsText.fcsByteOrder);
  for (int eventNum = 0; eventNum < numEvents; eventNum++)
    for (int paramNum = 0; paramNum < numParams; paramNum++)
      events[paramNum][eventNum] = (double) floatData.getFloat();
}
```

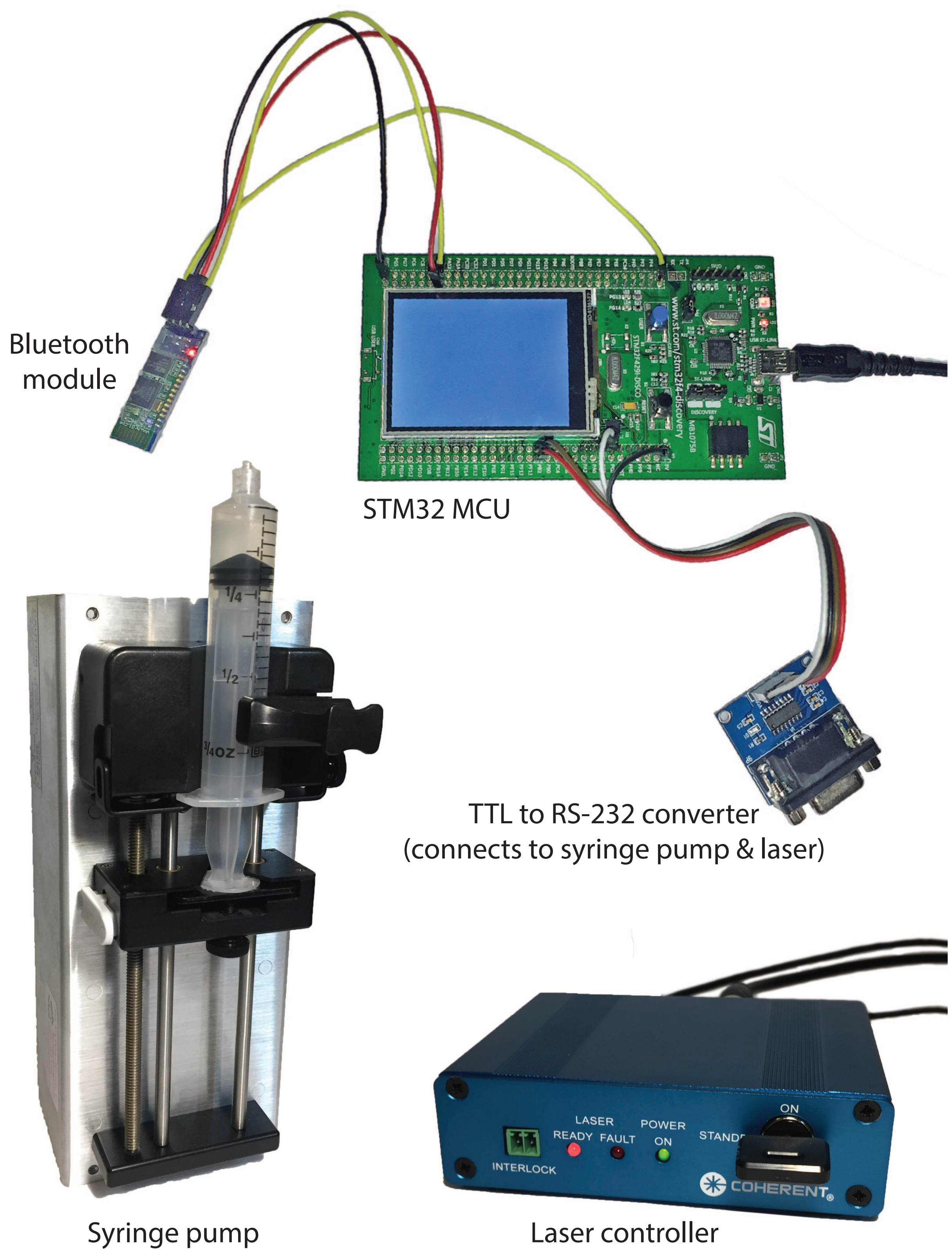
Listing 10: Reading double-precision floating-point data from the DATA segment of an FCS file.

```
private void readDoubleDataFromData(byte[] dataSegmentData) throws ParseException {
  int bytesToRead = numEvents * numParams * Double.BYTES;

  if (dataSegmentData.length < bytesToRead)
    throw new ParseException("DATA segment is too small for data", 0);

  int byteOffset = 0;
  for (int eventNum = 0; eventNum < numEvents; eventNum++) {
    for (int paramNum = 0; paramNum < numParams; paramNum++) {
      byte[] bytes = Arrays.copyOfRange(dataSegmentData, byteOffset,
      byteOffset + Double.BYTES);
      events[paramNum][eventNum] = ByteBuffer.wrap(bytes).order(fcsText.fcsByteOrder)
                                               .getDouble();
      byteOffset += Double.BYTES;
    }
  }
}
```

Appendix B
Images



APPENDIX C
ORIGINAL REQUIREMENTS

This section contains all of the functional, performance, and quality requirements of the system. Each requirement is assigned a priority of either critical, non-critical, or optional. Critical requirements are those that are necessary to ensure a minimum viable product. Non-critical requirements are planned along with the critical requirements, and are required to meet the goal of the project but are not required to create a minimum viable product. Optional requirements are not part of the core requirements for this project, but rather are provided as stretch goals that may be considered during product development.

C.1 Functional requirements

C.1.1 UI landscape orientation [Priority: CRITICAL]

The UI shall be designed for a tablet in landscape orientation.

C.1.2 UI portrait orientation [Priority: OPTIONAL]

The UI shall support both landscape and portrait orientations. Landscape will remain the standard orientation but support for portrait orientation will allow the UI layout to change based on orientation.

C.1.3 Continuous scaling of visuals [Priority: CRITICAL]

Data plots shall support both zooming in and zooming out. The zoom out feature should be restricted to go no farther than displaying the maximum data range. Ideally, this will be implemented using a pinch-to-zoom gesture on the Android tablet's multi-touch interface. However, this could also be accomplished by other means, such as $+/-$ zoom controls.

C.1.4 Updating data [Priority: NON-CRITICAL]

Data will update at least every 2 seconds. This means that the plots displaying the data must also refresh as new data comes in. Ideally, a continuous update would be possible, such that the plots are always kept current with new incoming events. This may be restricted by the event rate and the data processing capabilities of the tablet, so it is not required that the plots are always current.

C.1.5 Continuous data streaming [Priority: OPTIONAL]

This is an optional improvement to the above, requiring data to update in realtime.

C.1.6 Data gating (region select) [Priority: CRITICAL]

Users shall be able to gate data on an initial plot using the Android tablet. This may be implemented by using a preset shape and adjusting its size and position. The data gate (selected region of data) shall be applied to all other data plots.

C.1.7 Custom gate regions [Priority: OPTIONAL]

Users should be able to draw a custom gate region on the graph using the multi-touch display. This could be based on a finger drawing on the display or from tools that aid in creating custom gate regions.

C.1.8 Store experiment data on device [Priority: CRITICAL]

Users shall be able to store the results of their experiment on the device as a single experiment. Logically, this data would be stored in the FCS file format, but it could be stored in some other format if that was deemed superior.

C.1.9 Nameable data files [Priority: NON-CRITICAL]

Users shall be able to name and later rename the files that store experiment data. This functionality should be included within the Android application and not require the user to exit the application in order to rename their files.

C.1.10 Delete experiment data on device [Priority: NON-CRITICAL]

Users shall be able to delete past experiments that are stored on the device. This functionality should be included in the Android application and not require the user to exit the application in order to delete their files.

C.1.11 Export data as FCS3.1 [Priority: NON-CRITICAL]

Users shall be able to transfer stored experiment data from the tablet to other devices using the FCS 3.1 standard [10]. This does not require that experiments be stored in the FCS format. Any experimental data that is not stored in the FCS format should be converted before export. It is expected that the user will be able to export the FCS files from the device through Email.

C.1.12 Wall-powered [Priority: CRITICAL]

The unit shall be powered from a conventional U.S. power strip. This is a joint requirement between the CS and ME teams. The ME team is responsible for the physical power connection. The CS team is responsible for power regulation and other factors that go into using the wall connector.

C.1.13 International power support [Priority: OPTIONAL]

The unit shall function properly in countries beside the U.S. This means that it supports $100 - 240V$ at $50/60Hz$. This is an important requirement for providing a product that could function worldwide but will not affect the initial prototype.

C.1.14 Instrument status feedback [Priority: OPTIONAL]

The unit shall display some status feedback besides what is displayed on the tablet. This could be an audio signal, status lights, some other indicator, or a combination. Feedback could include experiment start and stop, progress of the experiment, or error notices. This requirement may also include the ME team, which could be responsible for building the status indicators into the device.

C.1.15 Plot FSC, SSC, FL1, FL2 [Priority: CRITICAL]

The tablet UI shall display plots from all four detectors: FSC, SSC, FL1, and FL2. These plots should be displayed concurrently, although this may not be possible on smaller screen sizes, in certain orientations, or when other information is displayed. The ability to show one plot on the screen may help to view data on smaller screens but is not a requirement.

C.1.16 Flow rate visible [Priority: NON-CRITICAL]

The flow rate shall be visible in the tablet UI. It should be clearly labeled.

C.1.17 Adjustable flow rate [Priority: NON-CRITICAL]

Users shall be able to adjust the flow rate. This means increasing or decreasing the speed of the actuator connected to the flow cell. Currently, this function is planned to be accessible from the Android tablet UI. It could instead be implemented as a dial or stepper located on the device. If it is implemented as a physical control on the device, it would include the ME team for building the control into the device.

C.1.18 Show event count [Priority: CRITICAL]

The event count shall be visible in the tablet UI. It should be clearly labeled or easily understandable.

C.1.19 Show event rate [Priority: CRITICAL]

The event rate shall be visible in the tablet UI. It should be clearly labeled or easily understandable.

C.1.20 H and A event data [Priority: CRITICAL]

The unit shall send height (H) and area (A) data from each detector to the tablet for each detected event. The height will be the maximum signal variation that was detected for an event. The area will be the sum of all heights detected for a single event. A single height and area data point must be calculated for each detector.

C.1.21 Start an experiment [Priority: CRITICAL]

Users shall be able to start an experiment from within the Android application.

C.2 Performance requirements

The requirements in this section provide a specification of user interaction with software and measurement placed on the system performance.

C.2.1 Sampling rate [Priority: CRITICAL]

A $5\,\mu\mathrm{m}$ particle shall be sampled at least 25 times as it transits the laser beam. Assuming a linear velocity of $1\,\mathrm{m/s}$ and a laser beam diameter of $1\,\mu\mathrm{m}$ at the interrogation point, the transit time will be $25\,\mu\mathrm{s}$. The MCU will need to sample sensors at $1\,\mathrm{MHz}$ minimum. If the hardware is capable of sampling at a faster rate, more data could be taken per particle, which is preferable for accurate data.

C.3 Design constraints

This section includes the design constraints on the software caused by the hardware.

C.3.1 Storage space on tablet [Priority: CRITICAL]

The application cannot create files that do not fit on the device. To meet this requirement, files created by a single experiment run must be small enough to store at least one experiment on the tablet.

C.3.2 Data throughput [Priority: CRITICAL]

The system must meet performance requirements of $1\,\mathrm{MHz}$ sampling from four sensors while successfully processing and transferring event data continuously to the tablet.

C.4 Changes to original requirements

- The FSC detector was removed due to difficulties in the mechanical design and therefore there will be no FSC data when performing experiments.
- System controls was added after the ME team decided we should cover it.
- Syringe diameter adjustment was added when the ME team moved to a syringe pump instead of a simple actuator. Setting the syringe diameter is necessary for correct functioning of the syringe pump.
- Plotting histograms on a logarithmic scale was added as a requirement. This was not well-understood when the original requirements were written, but is important to combat heavily-skewed data, especially for fluorescence channels.