5-1-2016

Bloon: Software and Hardware for Data Collection and Real-Time Analysis

Jacob Z. Weiss
Dartmouth College

Bloon: Software and Hardware for Data Collection and Real-Time Analysis

Jacob Weiss

Senior Honors Thesis

Advisors: Kristina Lynch (Physics) & Sean Smith (Computer Science)

Dartmouth Computer Science Technical Report TR2016-806

Abstract: Bloon is a powerful, user-friendly parsing and plotting utility optimized for use in real-time applications that I wrote for the Mac. Its generalized parser is capable of handling a wide variety of data formats. If you can define it in Bloon's parsing language, then Bloon can parse and plot it. Bloon's grapher is also extremely powerful, allowing users to zoom and scroll about millions of data points smoothly in real time. Bloon makes the difficult task of real-time data collection and visualization a breeze.

Contents

1 Software

Bloon is a powerful, user-friendly parsing and plotting utility optimized for use in real-time applications that I wrote for the Mac. Its generalized parser is capable of handling a wide variety of data formats. If you can define it in Bloon's parsing language, then Bloon can parse and plot it. Bloon's grapher is also extremely powerful, allowing users to zoom and scroll about millions of data points smoothly in real time. Bloon makes the difficult task of real-time data collection and visualization a breeze, and I am excited to see what people do with it!

As a scientist and engineer, the ability to visualize data is invaluable, and many excellent software packages exist to generate graphics. However, most of these solutions are optimized for quality. They produce beautiful graphics, but do so very slowly. This paradigm poses no problems when the data already exists, but falls flat if applied to real-time data visualization. And there are many situations when visualizing data in real time is vital. Problems can be caught as they happen, instead of being noticed after the fact. Bloon attempts to tighten this feedback loop by empowering scientists, engineers, and hobbyists alike to interact with their data as it is generated.

As can be seen in figure 1 below, Bloon is programed primarily (about 66%) in Swift. Swift was chosen over Objective C as the primary language for this project due to its clean syntax and fast execution time. The rest of the code base is written in C or Objective C. These languages were only used when the application required more direct memory manipulation and management.

Language	**files**	**blank**	**comment**	**code**
Swift	66	1892	721	7630
Objective C	7	861	267	3329
C/C++ Header	12	128	118	284
C	3	26	30	138
Total:	**88**	**2907**	**1136**	**11381**

Table 1: Files and Lines of Code

1.1 Other Solutions

Matlab: `http://www.mathworks.com/products/matlab/`

Matlab is an extremely powerful tool for scientific computing and data analysis. However, it lacks real-time capabilities built in, and although many people have tried to add these capabilities to Matlab, the results are not ideal. Matlab produces plots that are focused on quality, not speed, and therefore is unable to update its plots at the rate required to enable interactivity.

Matplotlib: `http://matplotlib.org/`

Matplotlib is the defacto plotting library for Python. It is extremely mature and well supported, however it also prioritizes graph quality over speed. Additionally, compared to a compiled language, Python is not the ideal choice for data processing.

PyQtGraph: `http://www.pyqtgraph.org/`

PyQtGraph is an alternative graphing library for Python that puts an emphasis on performance and is therefore better suited to real-time interactive plots. PyQtGraph is an excellent foundation to build an interactive plotting application upon, however, due to its Python foundation, PyQtGraph is not the ideal choice for large-scale data processing and begins to show considerable slowdown at just a few million data points.

COSMOS: `http://cosmosrb.com/`

Ball Aerospace's COSMOS is an extremely powerful tool build for real-time telemetry with remote payloads. Although it has real-time parsing and plotting capabilites, these features are not fast enough to be interactive and are not the focus of the software. COSMOS is intended to be a complete telemetry solution and although it is powerful, it is also quite difficult to set up.

MakerPlot: `http://www.makerplot.com/`

MakerPlot is a windows-only real-time plotter aimed at the maker community, specifically focused on the Arduino. The program has many capabilities, including the creation of custom dashboards containing various types of displays. The Arduino controls the plotting software by sending ASCII commands over serial to MakerPlot. All plotting is done on ASCII-formatted data. MakerPlot is an excellent choice for parsing and plotting simple data coming directly from a microcontroller. However, it is (at least out of the box) unable to parse complicated packets such as those emitted from a wireless radio. MakerPlot's feature set is better tailored to command, control, and monitor microcontrollers in an industrial setting than to record and visualizing data.

MegunoLink: `http://www.megunolink.com/`

MegunoLink (Windows) provides many of the same features as COSMOS and MakerPlot (command, control, and monitoring), but is aimed specifically at the Arduino market. It even goes so far as to provide an Arduino library for communicating with the application, and a drag-and-drop interface for creating custom control interfaces (like MakerPlot). However, this software falls short in complex bandwidth-limited situations where the overhead generated by Megunolink's message structure and ASCII formatting is unacceptable.

Realtime Plotter: `https://github.com/sebnil/RealtimePlotter`

Realtime Plotter is a simple open-source project that nicely demonstrates the type of program that most makers are currently using to plot their data. It can plot six channels of ASCII data streaming over a serial port, and it does this quite well. However, this project does not allow users to interact and explore the data, and is really only intended for very basic cases. Additionally, the restricted input format limits the usefulness of such an application in more complicated situations.

Arduino IDE: `https://www.arduino.cc/en/Main/Software`

For an Arduino user who just wants to quickly check that their sensor is working, the Arduino IDE has recently been updated to include a real-time data plotter. However, like many of the other solutions, the plotter provides no easy way of customizing the input format.

IOComp Plot Pack: `http://www.iocomp.com/`

IOComp Plot Pack is an extremely powerful real-time graphing utility. It's plotting capabilities far exceed those of Bloon. However, the basic version costs hundreds of dollars, contains no parsing functionality, and is Windows only.

KST: `https://kst-plot.kde.org/`

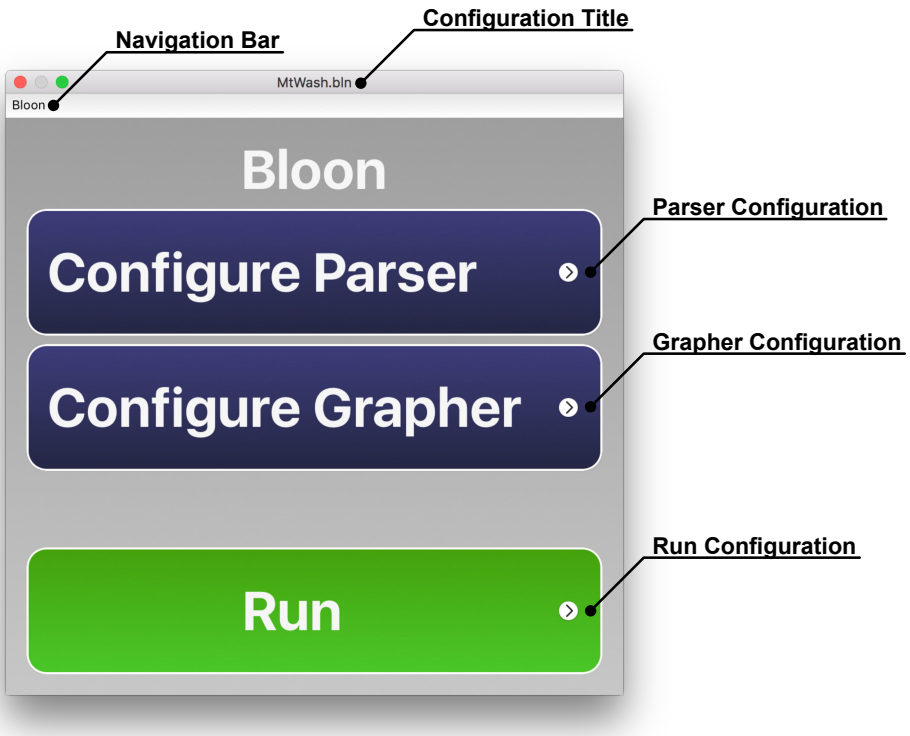
KST is self-described as "the fastest real-time large-dataset viewing and plotting tool available," and it is a powerful piece of open-source software. It is capable of plotting extremely large datasets very quickly and has some very mature data processing and analysis functions. However, it lacks the user-definable parser and smooth graph interaction that Bloon boasts.

1.2 Documentation

This section will describe each function of Bloon by walking through each screen of the graphical user interface, but first it will be useful to discuss Bloon's design on a large scale. Bloon can be thought of as being split into two parts: the Parser and the Grapher. The parser takes an input data stream (from a file, a pipe, or a serial port) and attempts to match pieces of the stream with predefined patterns. When a match is detected, the parser records the location and type of the

pattern and then moves on. The grapher looks at these records, pulls out the data, and then plots it in a graphical format. In order to facilitate repeated experiments where the same data may be collected multiple times, each half of the application (Parser and Grapher) is divided again into the configuration and the runtime. This design significantly speeds up the process of collecting and viewing data, as the configuration file only needs to be created once by a single person which can then be shared. Running the configuration is extremely simple. Bloon's graphical user interface exists to create, open, save, and run configuration files. In this document, I will describe the graphical user interface and how it is used.

1.2.1 Main

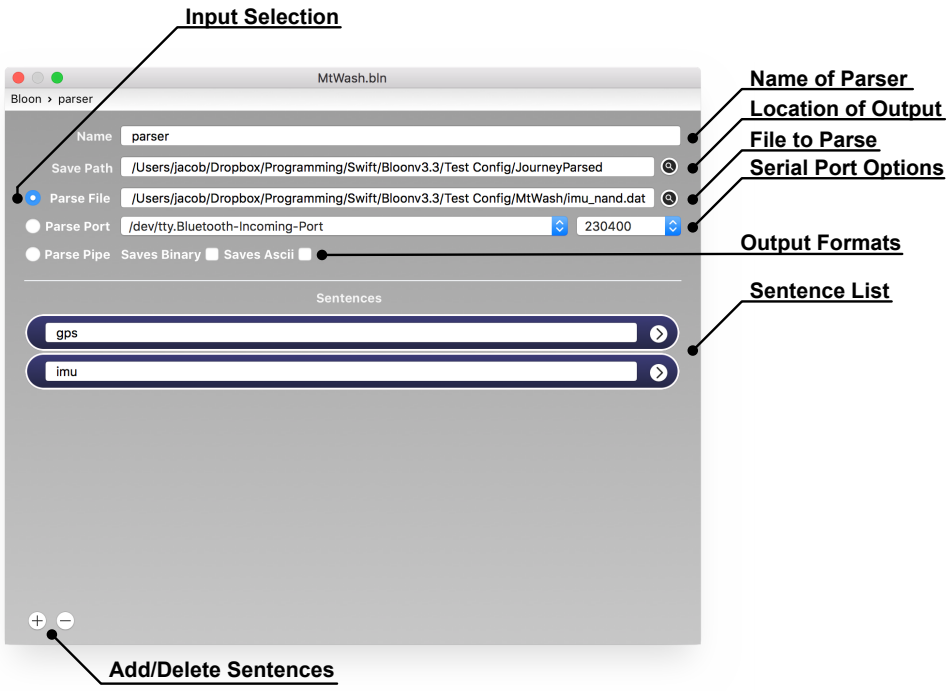


The root view of Bloon's graphical user interface displays the three major function that a user may wish to perform: configure the parser, configure the grapher, and run the configuration. The name of the configuration is displayed as the window's title. Just below this is the navigation bar. This navigation bar lets you quickly navigate up and down the tree structure of the configuration. As you move through a configuration, the navigation bar displays breadcrumbs. Clicking on a breadcrumb will bring you back to that view.

Now is also a good time to note the File menu. Here you can find shortcuts to run, stop,

open, save, and create a new configuration file. Pressing the shift key reveals a few lesser-used options, including the dangerous "Run No Output" option which will run the configuration file and suppress all output files. Any data collected in this mode will not be saved to a file.

1.2.2 Parser



In Bloon, a parser takes a stream of data as input and attempts to match it to the provided patterns (called sentences).

At the top of the parser's configuration, you can give the parser a name. This name will be used in the grapher, so it is a good idea to keep it short.

Clicking the magnifying glass on the right of the **`Save Path`** box lets you select the location in which Bloon will save all its output files. A parser, unless run in the **`No Ouput`** mode described previously, will always write the input data stream directly to a file with no modifications. The **`Output Formats`** checkboxes control whether this parser will output a few other formats as well. If **`Saves Binary`** is checked, then another file is created. This file will contain raw data, but only the data that was successfully matched to a sentence. **`Saves Ascii`** creates another file, but the data is output in a parsed Ascii format.

On the left, the three possible input sources are listed. It should be noted that the rest of the program does not distinguish between the various input sources. If **`Parse File`** is selected, then by clicking on the magnifying glass to the right, you can select a file that will be parsed.

`Parse Port` allows Bloon to easily parse data from a serial port. The dropdown menus to the right of the radio button let you select the port and baud.

`Parse Pipe` is the most powerful input source that Bloon provides, and technically could be the only input source provided. With **`Parse Pipe`** selected, at runtime Bloon will create a named unix pipe at the location of the path displayed in the **`Parse File`** box. Any data piped into that file from the unix shell will be parsed. This allows Bloon to take data directly from any source capable of writing to stdout.

Finally, in the bottom left corner of the window you can find several buttons for creating and deleting sentences. The **`+`** button creates a new sentence. Clicking a sentence after it has been created will select it, at which time it can be deleted with the **`-`** button. Bloon supports multiple selection by holding down the shift or command keys. Additionally, Bloon can duplicate the selected sentences using the standard command-c and command-v keyboard shortcuts. The concepts of adding, deleting, copying, and pasting are implemented in the same way throughout the application. Bloon also supports copying and pasting elements between two configuration files.

1.2.3 Sentence

In Bloon, a sentence is a string of tokens. The order and type of the tokens determines the format that the sentence attempts to match. At the top of the view, you can name the sentence. The **`Saves Binary`** and **`Saves Ascii`** checkboxes work similarly to those of parsers. However, the output files generated by a sentence will only contain data matched by this sentence, while parsers' output files contain all of the combined output from each of its constituent sentences.

Each token has five properties: the type, the name, the number of bytes, the byte format, and the constant value. Depending on the type of token only some of the properties are used. The *Token Type* simply states the type of token. The *Token Name* is used to label the data. A token without a name cannot be plotted. It is useful to leave the name of constant tokens blank so that they do not clutter up the plotting interface. The combination of the *Num Bytes* property and the token's type determines how the parser calculates the length of the token. Clicking on the

byte format drop down menu lets you select the endianness and sign of the token, if it is relevant. Finally, the *Constant Value* property lets you define the constant value for tokens that represent static values.

In addition to add, delete, copy, and paste, tokens can be dragged and dropped to reorder them, as unlike sentences which are processed in parallel, the order of the tokens in the list is significant.

1.2.4 Token Reference

AsciiDouble*

An ascii double token will match and parse any floating point number in an ascii format. For example: 1.0, -2.0, 42, 15.6e-7, +42.42e+3

AsciiHex*

An ascii hex token will match and parse any hex number in an ascii format. For

example: AA, ABC, 012EF, FF

AsciiInt*

An ascii int token will match and parse an integer number in an ascii format. For example: 1, +2, -3, 400, -4200

BinInt*

A binary int token will take the next N bytes of input data and parse them as if they were a N*8 bit binary integer. Bloon supports both signed and unsigned integers in little and big endian byte orders.

Char

A char token will match a single byte and interpret it as an ASCII character.

ConstBinInt*

A constant binary int token will match any binary integer (like a BinInt token) equal to the provided value.

ConstChar

A constant char token will match any ascii character (like a ConstChar token) equal to the provided value.

ConstLengthAsciiDouble*

A constant length ascii double will take the next N bytes and attempt to interpret them as a floating point ASCII double (like the AsciiDouble token). It matches if a correctly formatted number is found.

ConstLengthAsciiHex*

A constant length ascii hex token will take the next N bytes and attempt to interpret them as a hexadecimal number. It matches if a correctly formatted number is found.

ConstLengthGroup

A constant length group matches its first N bytes as a binary integer (like BinInt) taken to mean the length (in bytes) of the group. Any type of token can be put into a constant length group, however, a parseable data token placed at the end of a constant length group token will consume all the bytes from its current location to the end of the constant length group and feed them to its parser.

ConstString

Matches the constant ascii string provided.

FixedLengthGroup

A fixed length group matches the next N bytes. In all other aspects, it is the same as a constant length group.

Group

A group token has no purpose other than to help organize a complicated parser.

NullTerminatedString

A null terminated string will match a string until a NULL character is reached.

TerminatedString

A terminated string will match a string until the given termination character is found.

ParseableData

A parseable data token takes the rest of the bytes from a constant or fixed length group and uses them as input into another parser. In this way, more complicated packet structures can be parsed.

*All numbers, including integers, are stored internally as double precision floating point values.

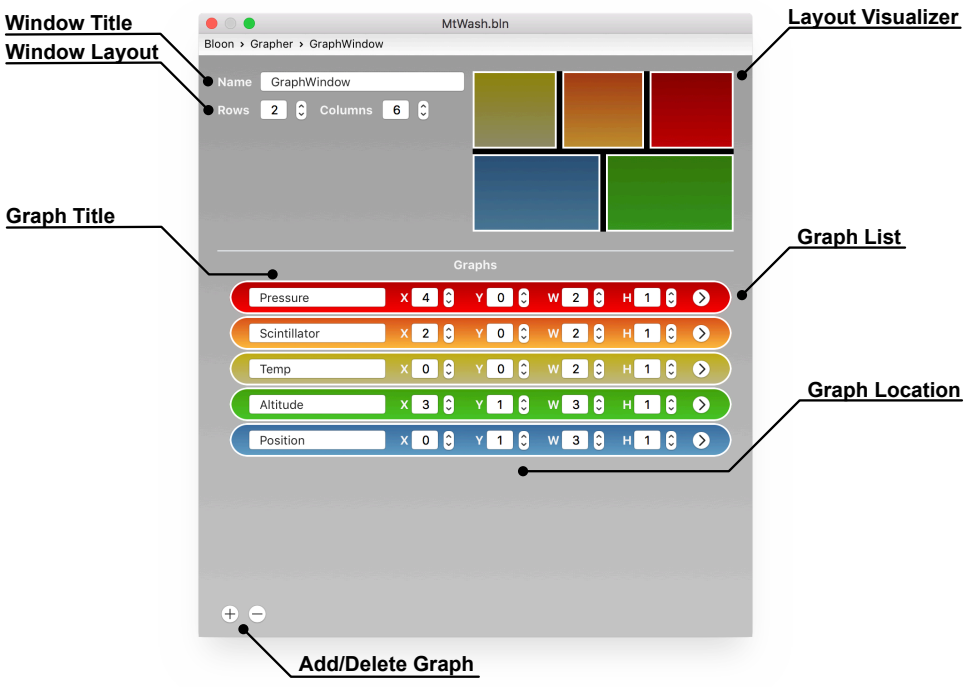
1.2.5 Window

The plotter's configuration, like that of the parser, is structured like a tree. However, the plotter's tree has a finite depth and is organized into three levels: the **`Window`**, the **`Graph`**, and the **`Plot`**. To create a new window, simply click on the **`+`** button in the bottom left of the configuration screen. Each window represents a graphical window that can contain graphs when Bloon is running. The window can be named by typing in its respective box.

1.2.6 Graph

Windows contain graphs. A graph is a 2-d scatter plot that visualizes data. To lay out the window, first decide how many rows and columns the window will have and input them into the **`Rows`** and **`Columns`** boxes. Next, create a graph by clicking the **`+`** button in the bottom left corner. In the top right corner, there is a preview that shows the layout of the graphs in the window. Each graph can be placed at any location in the window by modifying its **`x`**, **`y`**, **`width`**, and **`height`**. Note that (0,0) is in the top left.

If you want to create a non-uniform layout, like the one in the figure where there are two graphs in the second row and three in the first, you need to create a layout with six columns, the least common multiple of three and two.



1.2.7 Plot

Each graph can contain multiple plots. A plot is a single series of data that gets visualized on a graph. You can set the title of the plot in the **Title** box, and the initial x and y ranges can be set by editing their respective values in the top right of the display. Below those values are several settings that manipulate the overall look of the graph as a whole. Here you can change the color, or completely get rid of the axis, the axis labels, and the grid. You can also change the background color of the graph. However, the most important setting is on the left hand side where you select the x variable's source. There are three options: **Time**, **Index**, and **Variable**.

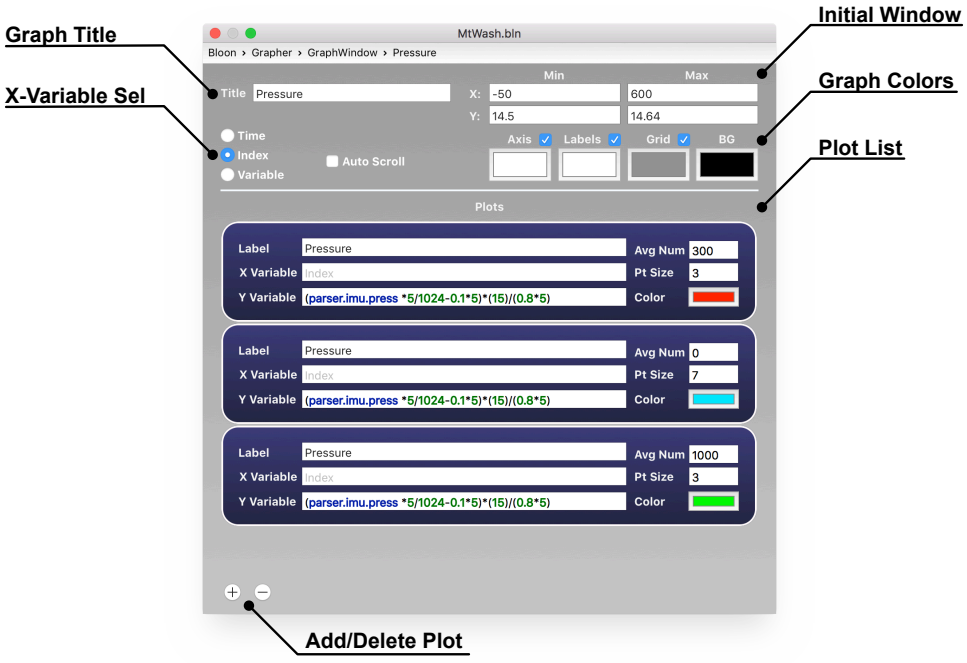
When **Time** is selected, the x value for a data point is determined by the time that it was parsed by the computer running Bloon. This mode is particularly useful if Bloon is going to be running for a long time, and it is important to have an accurate representation of the time that a data point was collected. Each unit along the x axis is equal to one second of real time according to the computer running bloon. Zero is set to the time that Bloon begins collecting data.

When **Index** is selected, the x value for a data point is set to increment after each data point is plotted. This mode is useful if you want your data to all be packed together, or if you are plotting a data file without a dedicated time variable.

Finally, when **Variable** is selected, Bloon calculates both coordinates based on the provided expressions. This allows you to create true 2-d scatterplots. Note that if multiple data values from different sentences are being used in the same plot, Bloon chooses the data points that were parsed closest to the same time. Therefore, plotting data like this is problematic unless it is being done in real time.

To create a new plot, click the **+** in the bottom left. Give the plot a name, and then type in the variable that should be plotted in the boxes provided. Variable names consist of the full path from the parser to the token, and as such can get quite long. Usefully, the names will autocomplete. Bloon can perform mathematical operations on data, an extremely useful feature for converting sensor data into real units. This functionality is provided by the excellent open-source **GCMathParser** (http://apptree.net/parser.htm) project. If Bloon detects an invalid expression, it will highlight the box in red. Mismatched parentheses and incorrect variable names commonly cause errors, so check them first if you have one. Each plot also has three other settings: **Avg Num**, **Pt Size**, and **Color**. If **Avg Num** > 1, then Bloon will calculate a running gaussian average using

the provided window size. **`Pt Size`** simply determines the size of the points in the scatter plot, and **`Color`** determines the color of said points.



1.3 Data structures

Many of the problems that arise when dealing with large volumes of real-time data quickly and efficiently require specialized data structures to solve. In the following section, I describe four of the most interesting data structures that I created and the unique issues that they each address.

1.3.1 MrSwArray

The **`MrSwArray`** (Multi-Reader-Single-Writer Array) is a dynamically expanding array data structure optimized for multithreaded use where there is a single writing thread and multiple reading threads. The data structure supports **`append`** and **`get`** operations, but once data is written, it cannot be modified. The data structure is optimized for high-volume writing and reading in a real-time environment where occasional large delays due to copying data are unacceptable. Additionally, the data structure is implemented so as not to require any synchronization between the threads. No synchronization means no blocking, which increases performance. Writing to the data structure takes amortized constant time, and reading can be done in constant time.

Typically, when a dynamically expanding array needs more space, it allocates enough memory

for the current contents plus some extra space, and then copies those contents into the new array. However, this configuration does not satisfy this application's needs for several reasons. First, as the size of the array increases, the amount of time spent on each copy operation increases as well. Although appending to an array can be done in constant amortized time, in a real-time environment where an application must remain responsive, large time penalties (even if uncommon) can hurt the user experience. Additionally, in a multi-threaded environment, it is difficult to determine how long to wait before freeing the old copy of the array without additional complicated state.

The `MrSwArray` solves these problems by storing data in chunks and maintaining a list of said chunks. If the chunks are large, then the list that keeps track of them is of insignificant size. The location of an index in the array can be calculated using modulus to determine the chunk, and integer division to determine the location inside the chunk, and is therefore (although slower than a raw array) a constant time operation. Expanding the array involves allocating space for more chunks and space to keep track of them. The old chunk-list then gets copied to the new one. However, since the chunk list is orders of magnitude smaller than the data itself, the copy operation takes a negligible amount of time and afterwards, instead of freeing the old chunk-list, it is pushed onto another list where it is kept until the destruction of the `MrSwArray`. Because the old chunk-list is not destroyed, any reader currently iterating over it will not be disrupted when the `MrSwArray` needs to be expanded. Finally, the pointer to the old chunk-list is atomically replaced with a pointer to the new one, and the length of the array is atomically updated.

1.3.2 LinkedIndexList

The `LinkedIndexList` is a data structure that can be used to iterate over a range of integers, and for each integer, decide whether to skip it and return to it later, or remove it from the list. Both creating the list and iterating over its elements take O(N) time N being the length of the list. Both skipping and removing are done in O(1) time, so will not negatively impact the runtime of the iteration. The `LinkedIndexList` does not guarantee that the indices will be visited consecutively or in order, only that when finished, every index will have been removed from the list precisely once. These features can be optimally implemented using a singly-linked list.

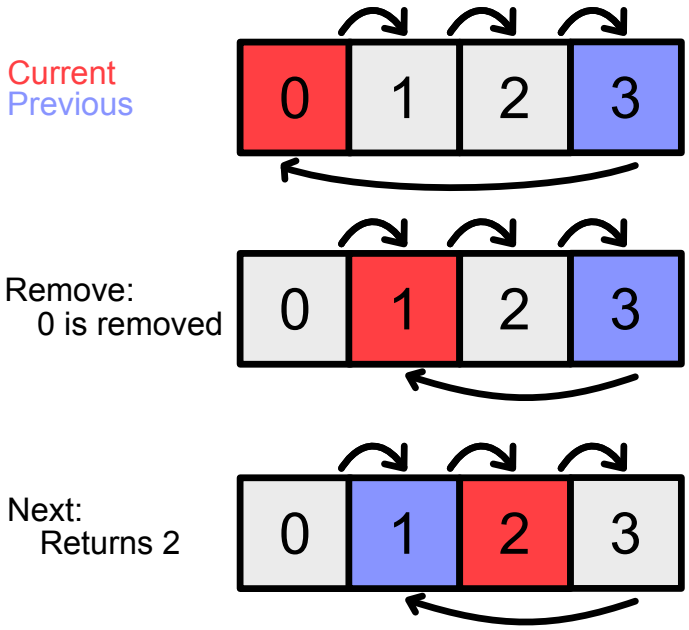


Figure 1: LinkedIndexList

Since upon creation, a **`LinkedIndexList`** knows how many indices it needs to contain, all of the space for the linked nodes can be allocated in a single large packed array. This has several benefits. First, only a single call to malloc is required to allocate the memory for any length list. And second, since all the linked elements of the list are consecutive in memory, we avoid the bad caching performance usually associated with linked structures. **`LinkedIndexLists`** have an iterator associated with them that maintains a **`current`** and **`previous`** pointer, as well as a pointer to the beginning of the allocated space so that it can be freed when the list is no longer needed. Calling **`next()`** on a **`LinkedIndexList`** moves both the **`current`** and **`previous`** pointers forward and then returns the index at the new **`current`**. Calling **`remove()`** moves the **`current`** pointer forward and then links the **`previous`** node to the new **`current`** node, removing the old **`current`** node from the loop. Notice that calling **`next()`** again will move the **`current`** pointer foward again, skipping an element. However, because a **`LinkedIndexList`** does not guarantee consecutive access, this is ok.

I implemented this data structure twice: first in Swift, and then in C. Figure 2 shows a profile comparing the two implementations. The graph shows memory usage vs time. In the test, a large LinkedIndexList was created and then iterated over. Each index was removed until the list was

empty. The first bump on the graph is the C implementation. The second one is the equivalent Swift implementation. As shown in figure 2, the C implementation is both faster, and consumes far less memory.

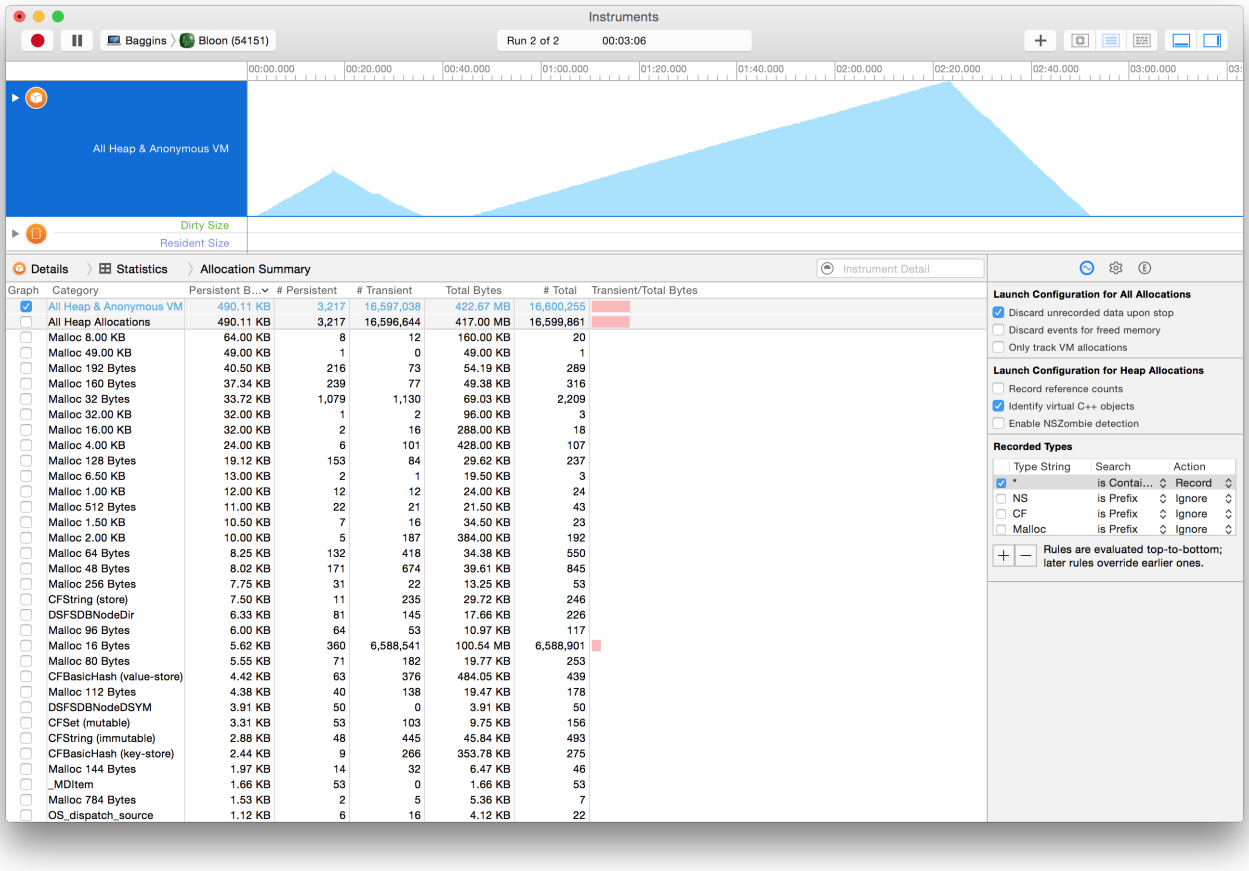


Figure 2: C vs Swift Performance Test

1.3.3 VertexArray

In order to utilize OpenGL most efficiently, each plot needs to store its vertices in a single long array. VertexArray is, at its heart, a simple dynamically expandable array. Its purpose is to, at all times and with no extra processing, provide a pointer to an array that contains a series of points that should be plotted. It also houses the averaging algorithm described in section 1.4.1. Additionally, a VertexArray can record a separate array that contains only the points that have been added after the most recent accessing of the array. This allows the grapher to, when appropriate, only draw the changes instead of having to redraw the entire screen.

Additionally, the VertexArray maintains bounds for each consecutive chunk of 8192 data points.

This information is used in two ways. First, it is used to speed up the rendering of data points. Before Bloon is able to draw vertices to the screen, it needs to transform them from their native coordinate space into the coordinate space of the graph. This process can be slow. However, because Bloon has access to the bounds of each individual chunk of vertices, it can quickly discard entire chunks of vertices if their bounds do not intersect with those of the current graph window. Bloon also uses these bounding boxes to speed up the nearest neighbor algorithm described in section 1.4.4.

1.3.4 ParsedSentence

Every sentence that the parser parses needs to be stored somewhere, along with some helpful metadata about the parsed sentence. Initially, this data was stored in a Swift object that contained a pointer to the data object, the start and end indices of the sentence in the data object, a pointer back to the Sentence object, the timestamp that the sentence was parsed at, two booleans for determining the current state of the token (is it currently parsing, and is it done parsing), and a complete dictionary that mapped the name of a token (a string) to its floating point value. Bloon requires one of these objects for each sentence parsed. Each of these structures takes up 48 bytes of space, plus the space needed to store a Swift dictionary. It quickly became obvious that this was one of the biggest consumers of memory in the application. Keeping track of all of the metadata made the implementation easier, but much of it was unnecessary.

To solve this problem, I stripped out everything non-essential from the structure and implemented it in C. I was left with a structure that contained only the index of the sentence in the data, a single byte to represent the state of the sentence (unparsed, parsing, and parsed), and a variable length array of doubles, built directly into the structure to avoid an unnecessary level of indirection. All the other parameters, it turned out, could be passed into functions when they were needed and are now stored in the Sentence object. The Sentence object also stores a dictionary that maps keys to the index at which they are stored in the ParsedSentence. This dictionary can be computed once, and used every time a piece of data needs to be retrieved. This new structure takes 16 bytes, plus 8 bytes times the number of values that need to be stored for the sentence, or about 3 times less than the previous implementation.

1.4 Algorithms

Bloon utilizes many algorithms to achieve its goals of real-time parsing and graphing. Below, I present four algorithms that I am particularly proud of. I do not claim to be the first to invent the following algorithms, but I did come up with them on my own.

1.4.1 Averaging

The ability to compute a running average is one of the simplest and most useful operations a graphing utility should be able to do. In the real world, sensors produce noisy data, and it is easier to interpret the data if it can first be smoothed. The simplest method of applying a moving average to an array of data is with a box-car averaging filter, where adding a new point can be done in constant time. An ideal filter that averages N elements should replace each element of the array with the mean of the N nearest elements to it (N/2 to the left, and N/2 to the right). In a real-time application, however, the N/2 elements to the right are not always available as some have not yet been collected. So what should the graph display? Displaying nothing, by far the easiest solution to implement, introduces a delay of N/2 data points. For large N, the delay can become unacceptably long. The next simplest solution is to, instead of averaging equally to the left and right, average only to the left. Unfortunately, this solution causes the average to appear to lag behind the real data. To complicate things further, data points may not be added in the proper order due to the relaxed nature of the LinkedIndexList. To solve these problems, I propose the following averaging algorithm for applying a convolution to an array in real time.

```
// State
Let AvgArray be an infinite array initialized to 0.
Let AdjArray be an infinite array initialized to 0.
Let N be the size of the convolution (the number of elements to be averaged).
Let Window be an array containing the convolution window. The sum of the elements of Window should equal 1.
Let x be the new value to be appended to the running average.
Let Index be the x's real index.

// Algorithm
let realMin = floor(Index - N / 2)
let max = floor(Index + N / 2.0)
let min = (realMin < 0) ? 0 : realMin

for i in min ..< max
    let coefficient = Window[i - min]
    let adj = AdjArray[i]
    AdjArray[i] += coefficient
    AvgArray[i] = (AvgArray[i] * adj + x * coefficient) / AdjArray[i]
```

To understand this algorithm, let us look at the lifetime of some index j in the **AvgArray** and **AdjArray**. Both begin initialized to 0.

Let k_0 be an index such that $j - \frac{N}{2} < k_0 < j + \frac{N}{2}$
Let x_{k0} be the value to be added
Since $\textbf{AdjArray}[j] == 0$
$\textbf{AvgArray}[j] = x_{k0}$
$\textbf{AdjArray}[j] = \textbf{Window}[j - (k_0 - \frac{N}{2})]$
Let k_1 be an index such that $j - \frac{N}{2} < k_1 < j + \frac{N}{2}$ and $k_1 \neq k_0$
Let x_{k1} be the value to be added
$\textbf{AdjArray}[j] = \textbf{Window}[j - (k_1 - \frac{N}{2})] + \textbf{Window}[j - (k_0 - \frac{N}{2})]$
$$\textbf{AvgArray}[j] = \frac{x_{k0} \cdot \textbf{Window}[j - (k_0 - \frac{N}{2})] + x_{k1} \cdot \textbf{Window}[j - (k_1 - \frac{N}{2})]}{\textbf{AdjArray}[j]}$$

At each iteration, $\textbf{AvgArray}[j]$ is multiplied by whatever its current denominator is (stored in $\textbf{AdjArray}[j]$). The new weighted value is added, and then the result is divided again to re-normalize the average. Therefore, at each iteration, the value stored in $\textbf{AvgArray}[j]$ is a valid average of the current data that is available. Once all the empty slots around $\textbf{AvgArray}[j]$ have been filled in...

$$\textbf{AvgArray}[j] = \frac{\sum_{n=j-\frac{N}{2}}^{j+\frac{N}{2}} x_n \cdot \textbf{Window}[n - (j - \frac{N}{2})]}{\sum_{n=j-\frac{N}{2}}^{j+\frac{N}{2}} \textbf{Window}[n - (j - \frac{N}{2})]}.$$

But we know that the denominator equals 1 from the assumption on line 5, so...

$$\texttt{AvgArray}[j] = \sum_{n=j-\frac{N}{2}}^{j+\frac{N}{2}} x_n \cdot \texttt{Window}[n - (j - \frac{N}{2})]$$

...which is exactly the definition of a windowed average. Additionally, we made no assumptions about the shape of the window, other than that its sum should be 1. Therefore, this algorithm can be used to take a box-car average just as easily as it could take a moving gaussian average. Bloon uses a gaussian function to create the window. Because after each addition, the average is left in a normalized state, the algorithm produces a reasonable output even when some points are missing, such as at the ends of the data. Appending a point takes linear time with respect to the number of points being averaged.

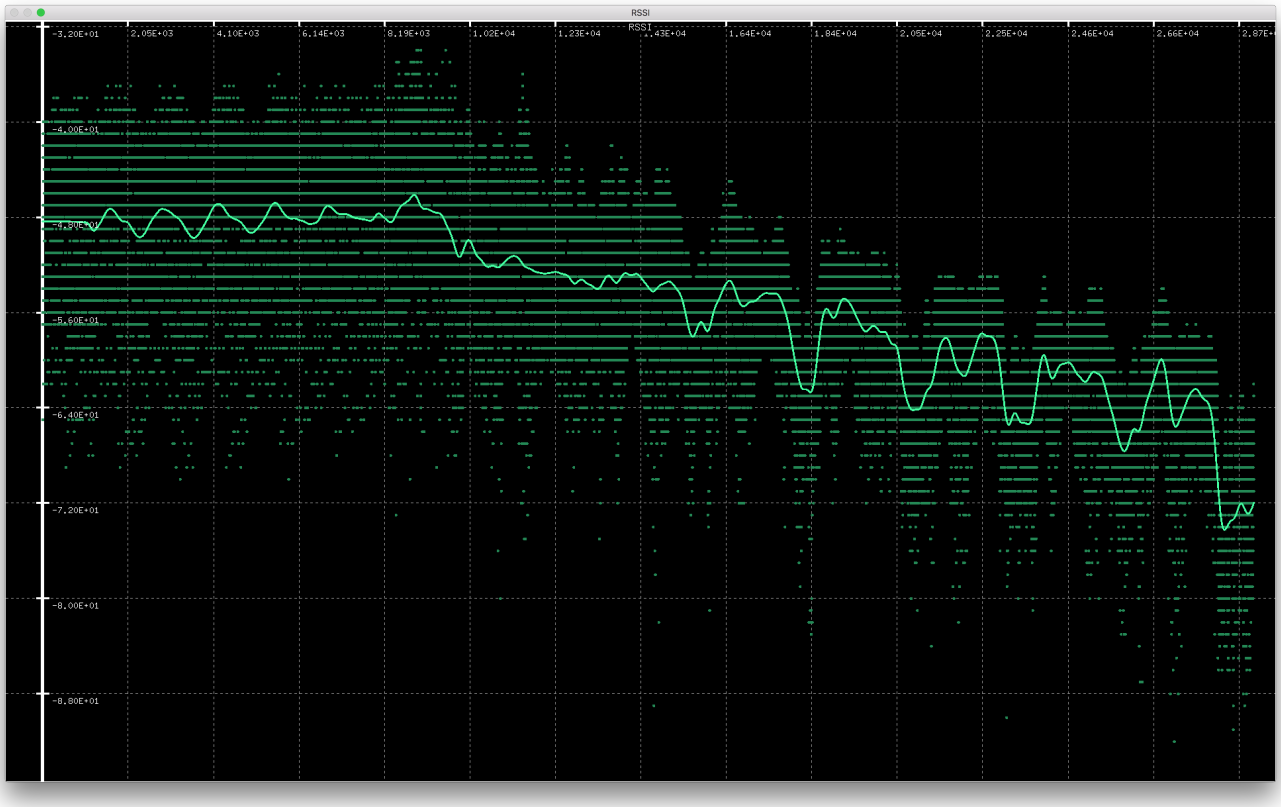


Figure 3: Bloon's averaging at work. The darker points are the un-averaged data. The brighter line represents the calculated moving gaussian average.

1.4.2 Graph Bounds Calculation

Most graphing utilities are capable of automatically guessing the proper bounds for the graph. For Bloon, this feature is mandatory, as it is likely that the data may appear somewhere far offscreen. It is extremely important that Bloon be capable of quickly locating all of the data and placing it inside the bounds of the window. This task can be accomplished trivially by keeping track of the current bounding box for each plot. However, in Bloon's intended use case where data is noisy, it would be even better if Bloon could try to calculate a bounding box that contains only the data that the user actually wants to see, and not the cloud of bad data that surrounds it.

To accomplish this, I implemented an algorithm that searches through the data twice (once for each dimension) and attempts to find the tightest range that contains some given percentage of all of the points. For example, let us say that we have a dataset of 1000 points with values from 0 to 100 and we want to find the smallest range that contains 99 percent of the points. That range may be 0 to 99, but if the data is not evenly distributed (because the data is very noisy), then that range could be much smaller. The algorithm that performs this calculation runs in order N time, and the pseudo-code is presented below.

In order to compute this property of the data quickly, the data needs to be in a sorted order. Instead of sorting the data in O(`SIZE`*log(`SIZE`)) time, Bloon calculates a histogram of the data in O(`SIZE`) time. Once the histogram has been calculated, two indices march through the data: one marking the start of the range, and one marking the end. Both indices are only ever incremented, and as such, the algorithm operates in O(`HISTOGRAM_SIZE`) time. Therefore, the entire algorithm operates in O(`SIZE` + `HISTOGRAM_SIZE`) time. `HISTOGRAM_SIZE` is constant and significantly smaller than `SIZE`, therefore, the algorithm truly runs in O(`SIZE`) time.

Figure 4 below shows a graph in Bloon where there is a large cloud of noise around the good data. Figure 5 shows the same graph after applying the automatic bounds calculation.

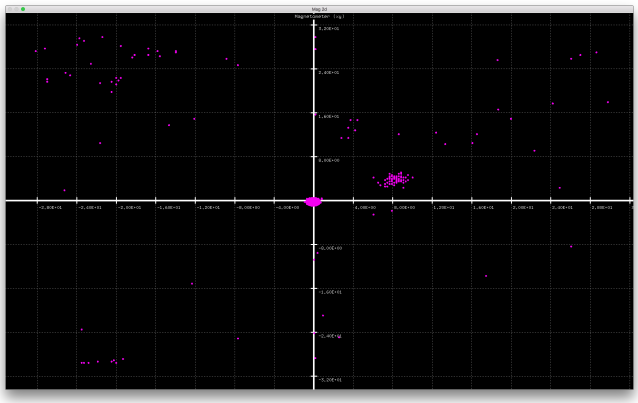


Figure 4: Before auto-bounds

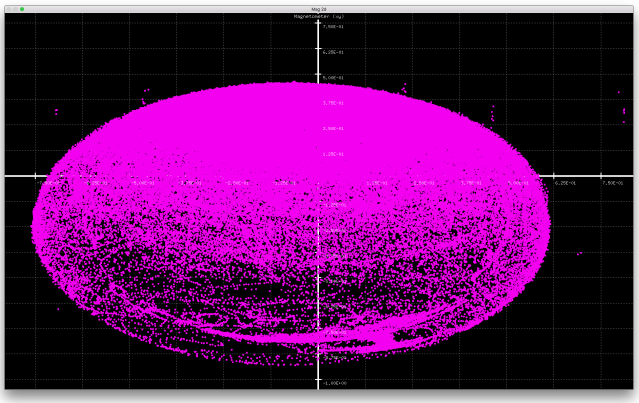


Figure 5: After auto-bounds

```
// State
Let DATA be an array of values
Let SIZE be length of DATA
Let HISTOGRAM_SIZE be the number of buckets in the histogram
Let PRECISION be the desired percentage of points to be enclosed

// Algorithm
let histogram = compute_histogram(DATA, HISTOGRAM_SIZE)
let minStart = 0
let minEnd = HISTOGRAM_SIZE
let start = 0
let end = 0
let points = 0
let isMovingEnd = true

while true
  if (isMovingEnd)
    if (end >= HISTOGRAM_SIZE)
      isMovingEnd = !isMovingEnd
      continue
    if (points / SIZE > PRECISION)
      if (end - start < minEnd - minStart)
        minStart = start
        minEnd = end
      isMovingEnd = !isMovingEnd
    points += histogram[end++]
  else
    if (points / SIZE > PRECISION)
      if (end - start < minEnd - minStart)
        minStart = start
        minEnd = end
      else
        if (end >= HISTOGRAM_SIZE)
          break
        isMovingEnd = !isMovingEnd
    points -= histogram[start++]
return (minStart, minEnd - minStart)
```

1.4.3 Graph Tick Mark Locations

Smooth and natural zooming and scrolling are extremely important features that allow the user to feel more connected to the data. In order to improve the effectiveness of zooming, Bloon dynamically places tick marks and grid lines as the user zooms. When the grid lines get too far apart, a new one is added in between. When they get too close together, every other line is removed.

In order to accomplish this, I derived the following function that relates the width of the window in pixels, the width of the displayed values in units, and the minimum pixels per tick (calculated based on the length of the tick mark labels). The function is stateless and completely deterministic. As such, tick marks will always increment by a power of two. Shown below is the function plotted using Bloon.

$$\texttt{unitsPerTick} = .5^{\lfloor log_{0.5}(\texttt{widthValue})\rfloor + \left\lfloor \lceil log_2(\texttt{widthValue})\rceil - log_2(\frac{\texttt{widthValue}\cdot\texttt{minPixelsPerTick}}{\texttt{widthPixels}})\right\rfloor}$$

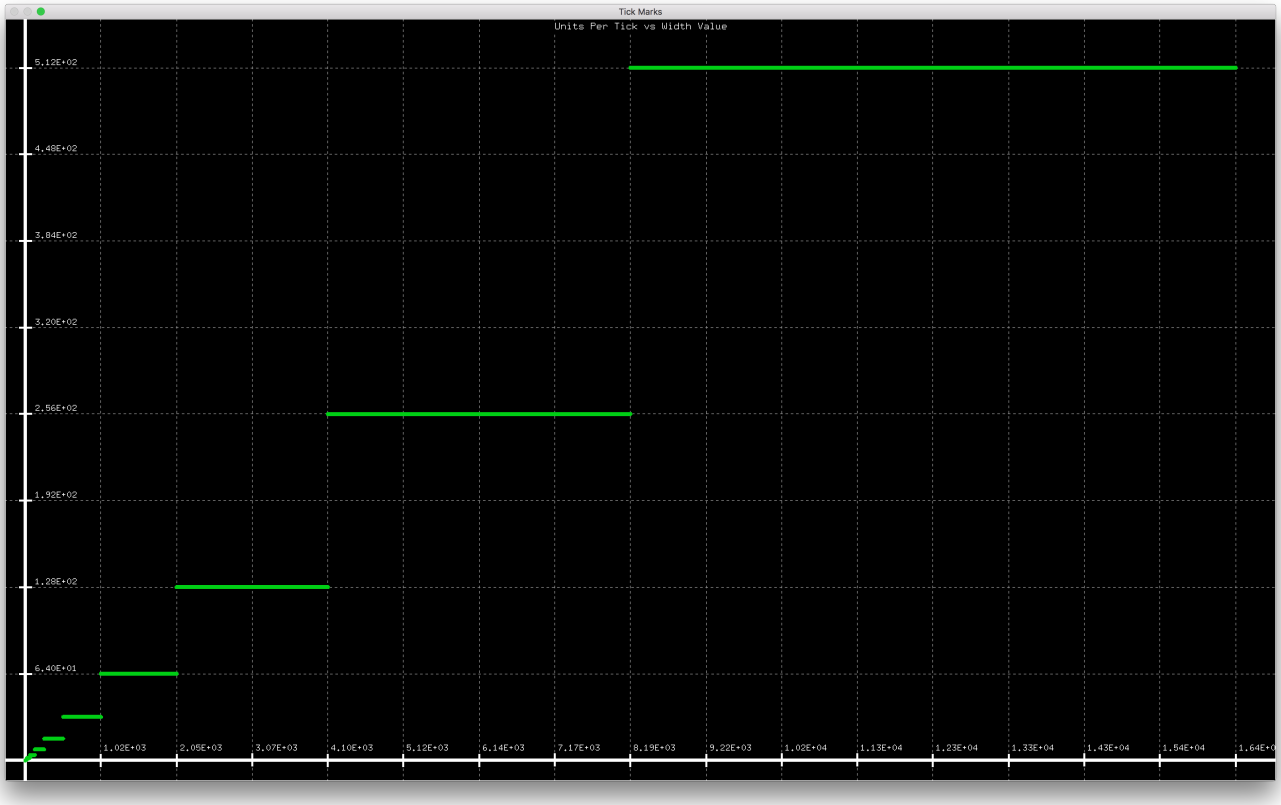


Figure 6: The function used by Bloon to place tick marks plotted using Bloon

1.4.4 Closest Point

Every good graphing utility should allow the user to check the exact value of any given point. By right-clicking on a graph, Bloon enters a mode where it highlights the closest point to the mouse cursor and displays its value. This is excellent from a user-interface perspective. However, from an algorithmic point of view, it presents quite a few challenges. Since Bloon does not store its vertices in any particular order, any algorithm to find the closest vertex to a given point must take linear time. However, when the datasets get too large, the naive implementation of such an algorithm (a simple linear search) is too slow to provide a smooth user experience. As such, I had to develop a better algorithm.

Although the vertices are not sorted, we do know (from **`VertexArray`**, section 1.3.3) a bounding box for each chunk of 8192 data points. In theory, these data points could be randomly distributed around the graph. In practice, though, this tends not to be the case. Since Bloon is optimized for real-time plotting, chunks of data points (especially when plotted against time or index) tend to be fairly close together. We can use the bounding boxes surrounding each chunk to quickly check if that chunk could possibly contain a point closer than the point that we have found already. If it might, then we have to do a linear search though that chunk of data points. If it cannot possibly contain a point closer than one we have already seen, we can skip that chunk entirely. The naive nearest neighbor algorithm took on average 0.0146 seconds to run on a certain graph. After implementing the improved algorithm, the same test executed in only 0.0009 seconds, over 16 times faster! The pseudocode for this algorithm can be seen below.

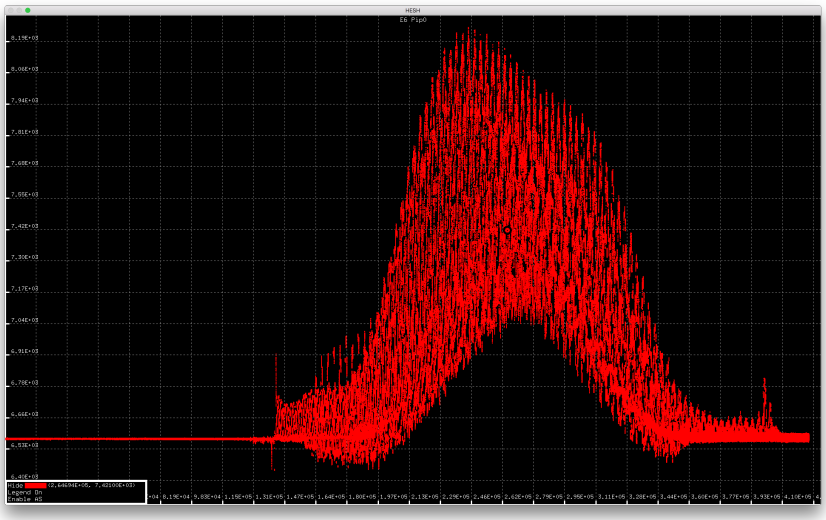


Figure 7: Bloon highlighting a data point and displaying its value in the bottom left corner.

```
// State
Let CHUNK_SIZE be 8192
Let CHUNKS be an array of chunks. Each chunk contains CHUNK_SIZE data points.
Let BOUNDS be an array containing the bounding rectangle of each chunk.
Let CHUNK_NUM be the number of chunks.
Let POINT be the point that we are trying to find the nearest neighbor of.

// Algorithm

let closestPoint = {0, 0}    // Initialize to anything
let closestDist = DOUBLE_MAX // Initialize to max double value

// First run through each chunk and compare the first point in each to the search point.
//   This is done to compensate for the fact that when data is plotted against time, the
//   chunks appear in a partially sorted order, with each chunk existing to the right
//   of the previous one. Therefore, if we are searching for a point on the far right
//   side of the graph, then since every chunk is closer than the previous one, almost
//   every chunk needs to be searched. This completely negates any benefits of this
//   algorithm. By quickly checking one point from each chunk, we increase the chances
//   that the algorithm will be able to reject entire chunks of data.
for i in 0 ..< CHUNK_NUM
  let test = CHUNKS[i][0] // Get the first point of each chunk
  let dist = distanceFromPointToPoint(POINT, test)
  if dist < closestDist
    closestPoint = test;
    closestDist = dist;

// Loop through each chunk again, this time to actually find the closest point
for i in 0 ..< CHUNK_NUM
  let bound = BOUNDS[i]
  // If the the circle defined with its center at POINT with a radius of closestDist
  //   intersects with the bounds of the given chunk, then that chunk may contain
  //   a point which is closer than closestPoint. Otherwise, it cannot contain a
  //   closer point, so the chunk is skipped.
  if circleIntersectsRect(POINT, closestDist, bound)
    for j in 0 ..< CHUNK_SIZE
      let test = CHUNKS[i][j]
      let dist = distanceFromPointToPoint(POINT, test);
      if dist < closestDist
        closestPoint = testRaw;
        closestDist = dist;

return closestPoint
```

1.5 Real World Uses

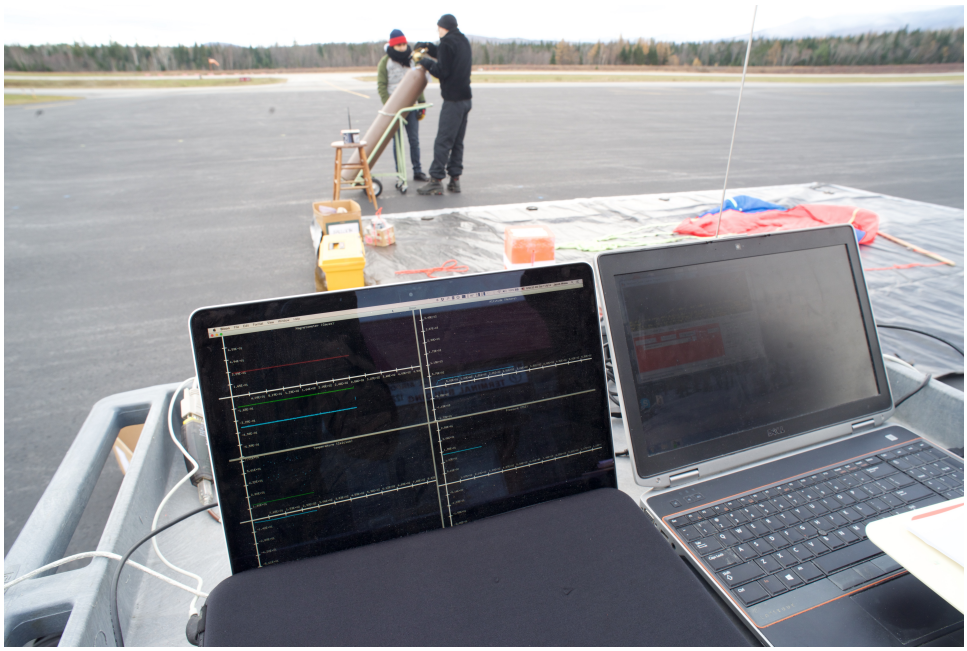


Figure 8: Bloon being used during a Greencube balloon launch

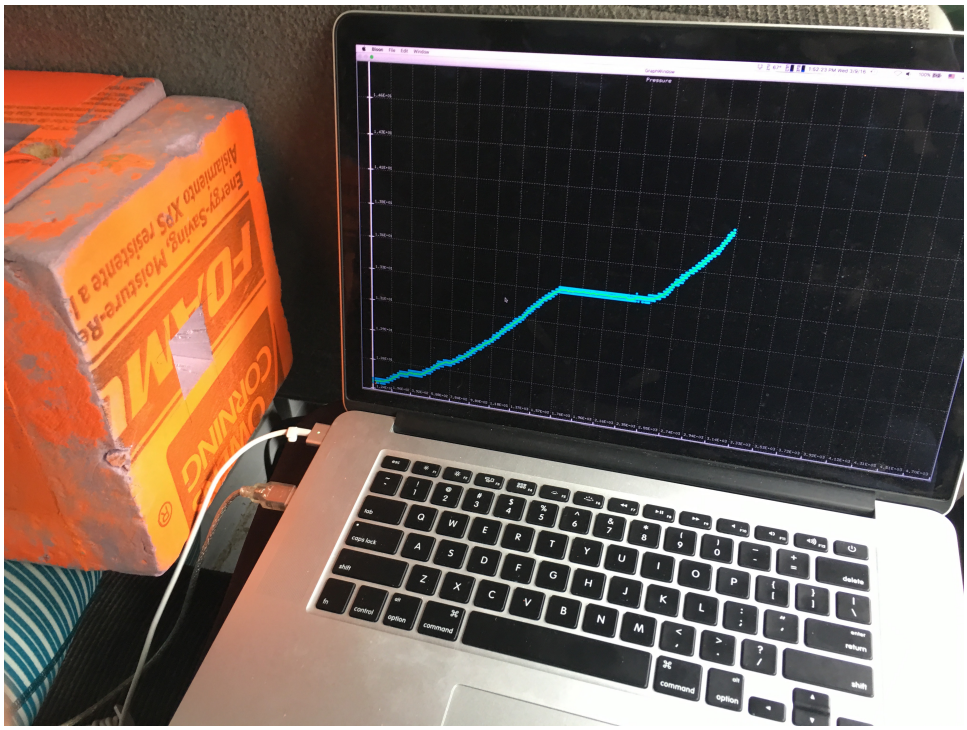


Figure 9: Bloon being used to record data while driving up Mt. Washington

2 Hardware

Due to its generality, it is possible to use Bloon with almost any hardware that can generate data. However, alongside my development of Bloon, I also developed an Arduino shield purpose-built for collecting data, the schematics and layouts of which are shown below. The shield comes equipped with on-board memory, an ADC, a DAC, a GPS, support for two different 900Mghz radios, and an ATXmega coprocessor to run all these extra features without loading the Arduino. The coprocessor on the shield runs custom firmware that communicates with the Arduino over SPI. I also implemented an Arduino API that makes it extremely easy to write code for the Arduino that controls the shield. The following sections include design schematics and drawings of the shield, as well as an API reference for the Arduino library.

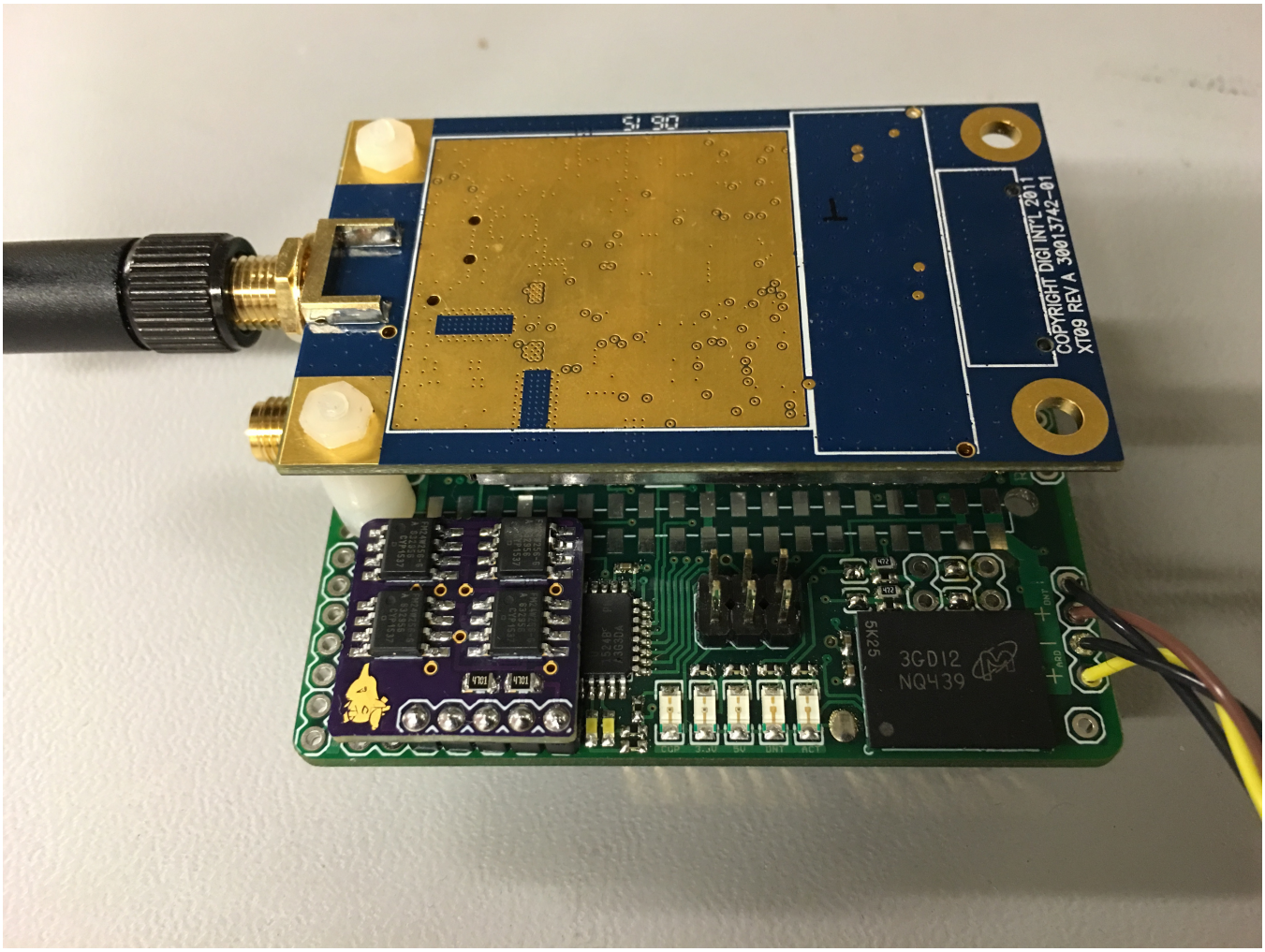


Figure 10: Fully Constructed Shield

2.1 Specifications

- Low current ±12V power supply
- High current +5V and +3.3V supplies
- MAX1147: 4ch, 14bit ADC

  2ch 0-5V, 2ch 0-3.3V
- 512 Megabyte Non-Volatile NAND Flash
- Integrated Venus 638 GPS
- 900 Mghz, 1 Watt Radio

  RFM DNT900 or Digi 9XTend
- 2ch 12-bit, -12V to +12V DAC
- ATXmega32e5 Coprocessor

  Provides easy-to-use Arduino API

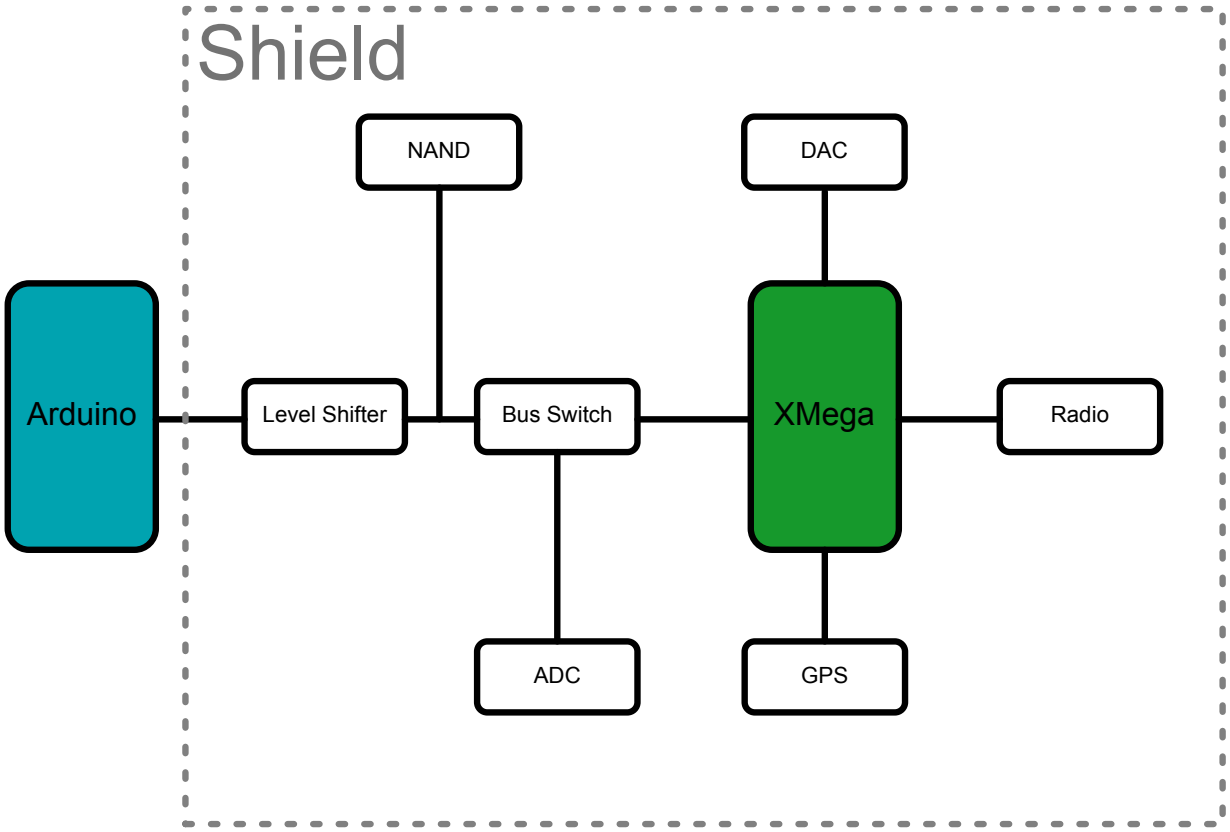


Figure 11: Block Diagram of BobShield

2.2 Real World Uses

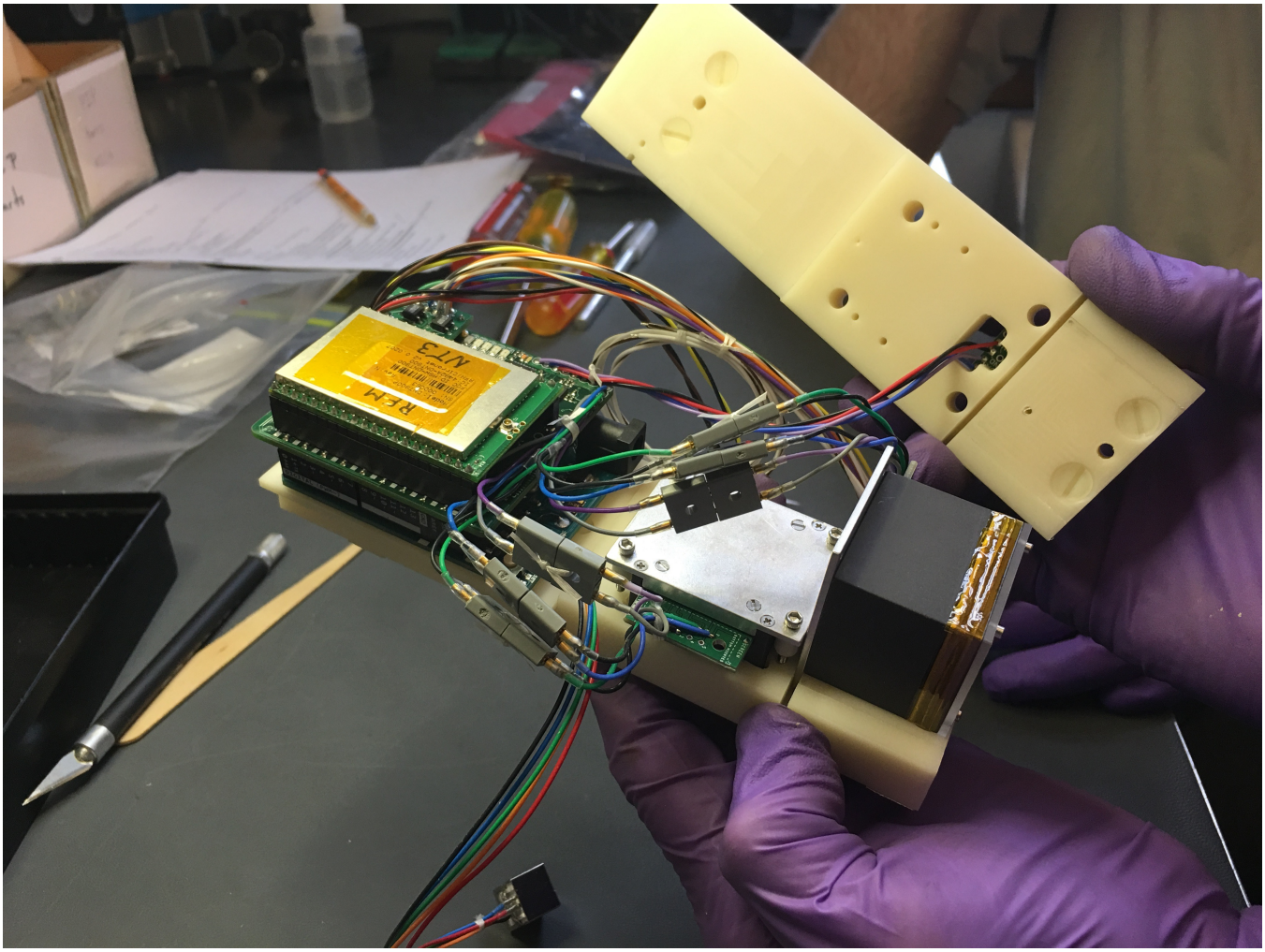


Figure 12: The shield built into a payload

2.3 Hardware Reference

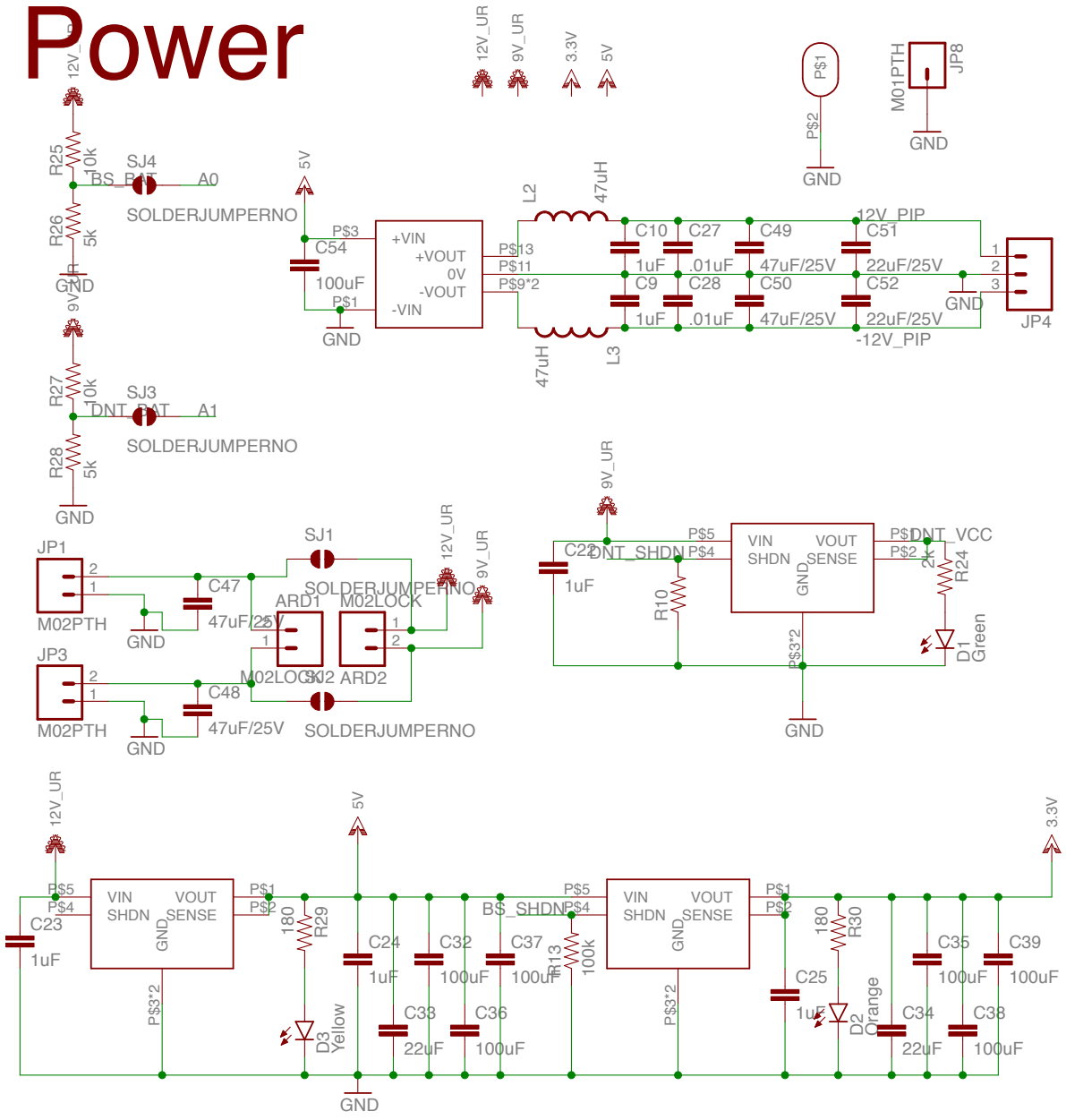


Figure 13: Power Schematic

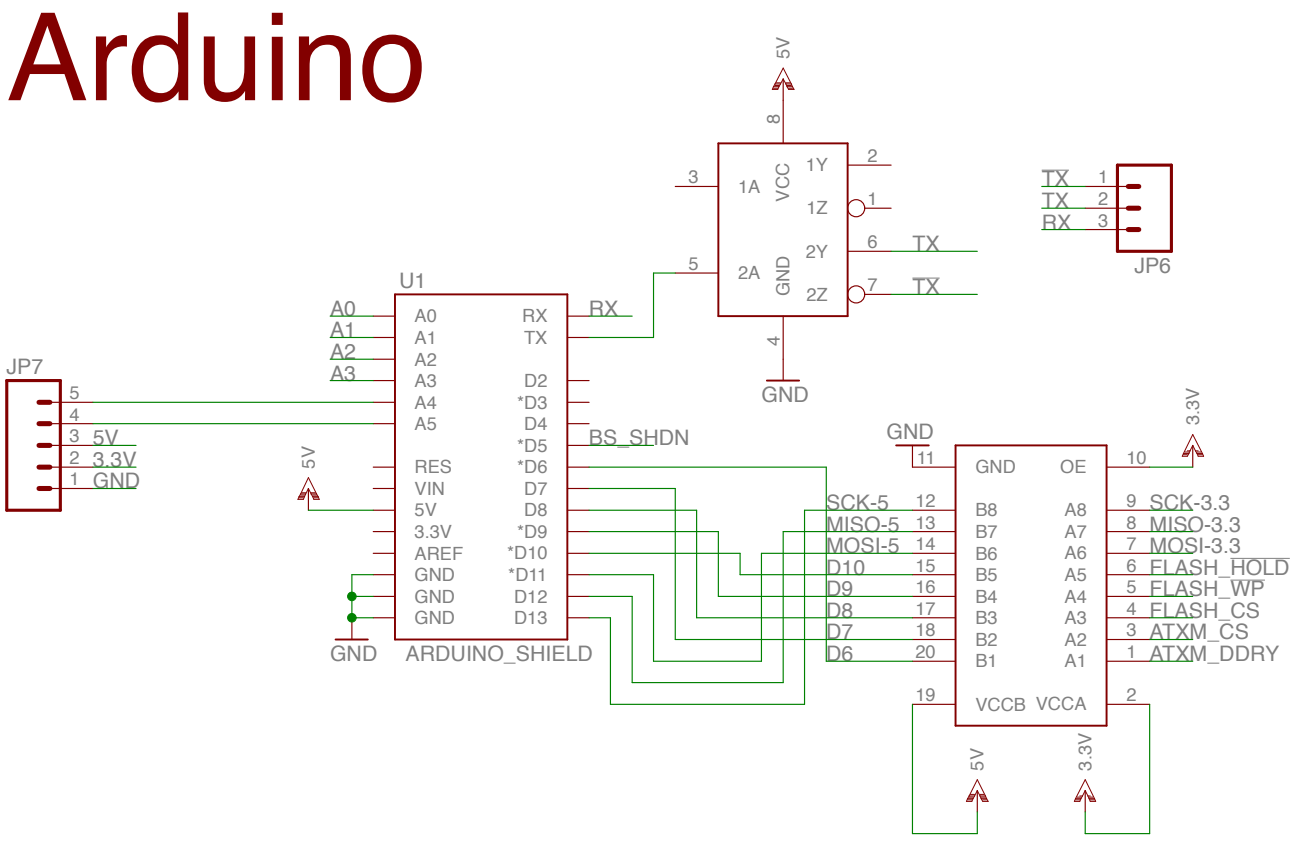


Figure 14: Arduino Schematic

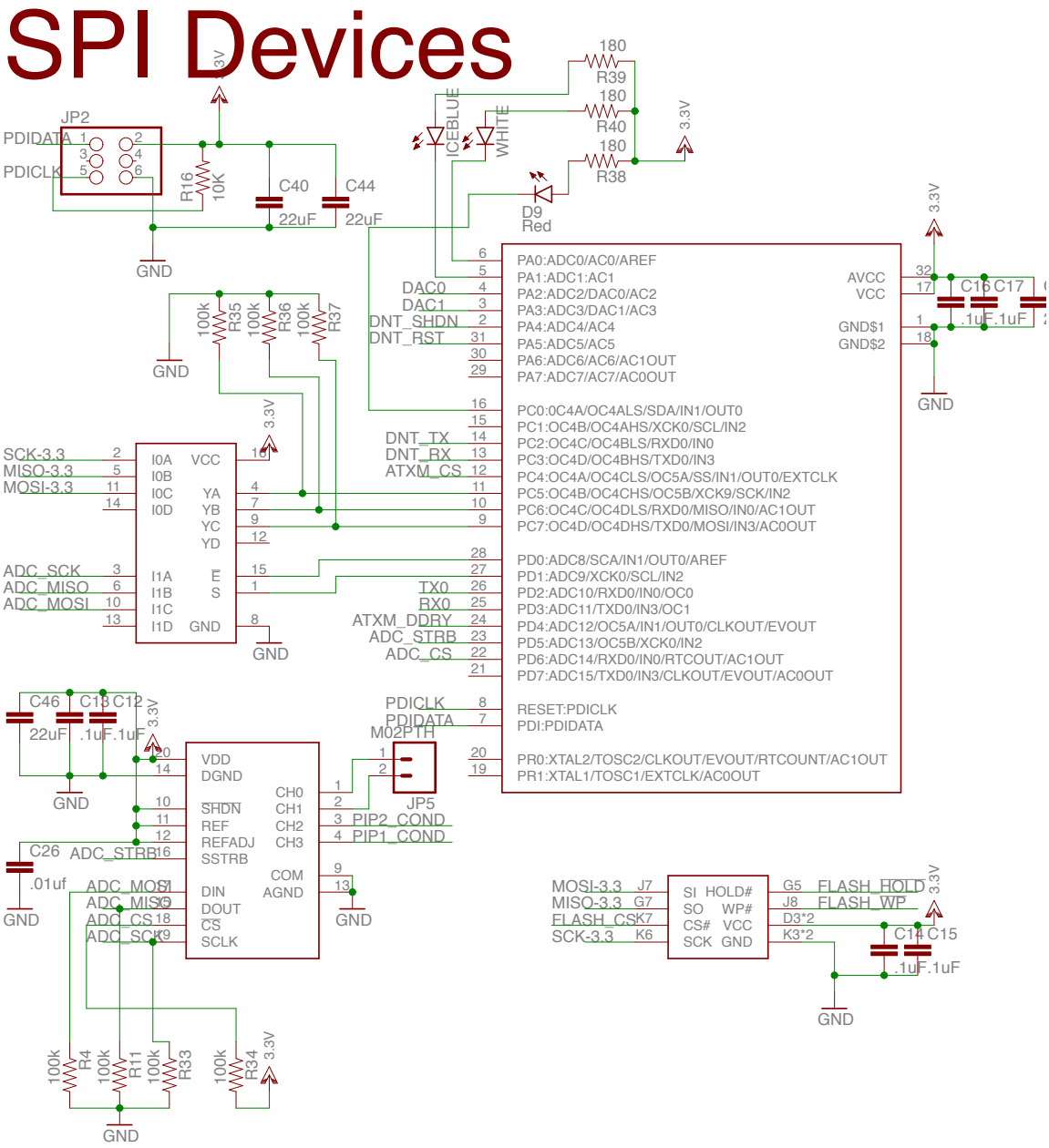


Figure 15: SPI Devices & Coprocessor Schematic

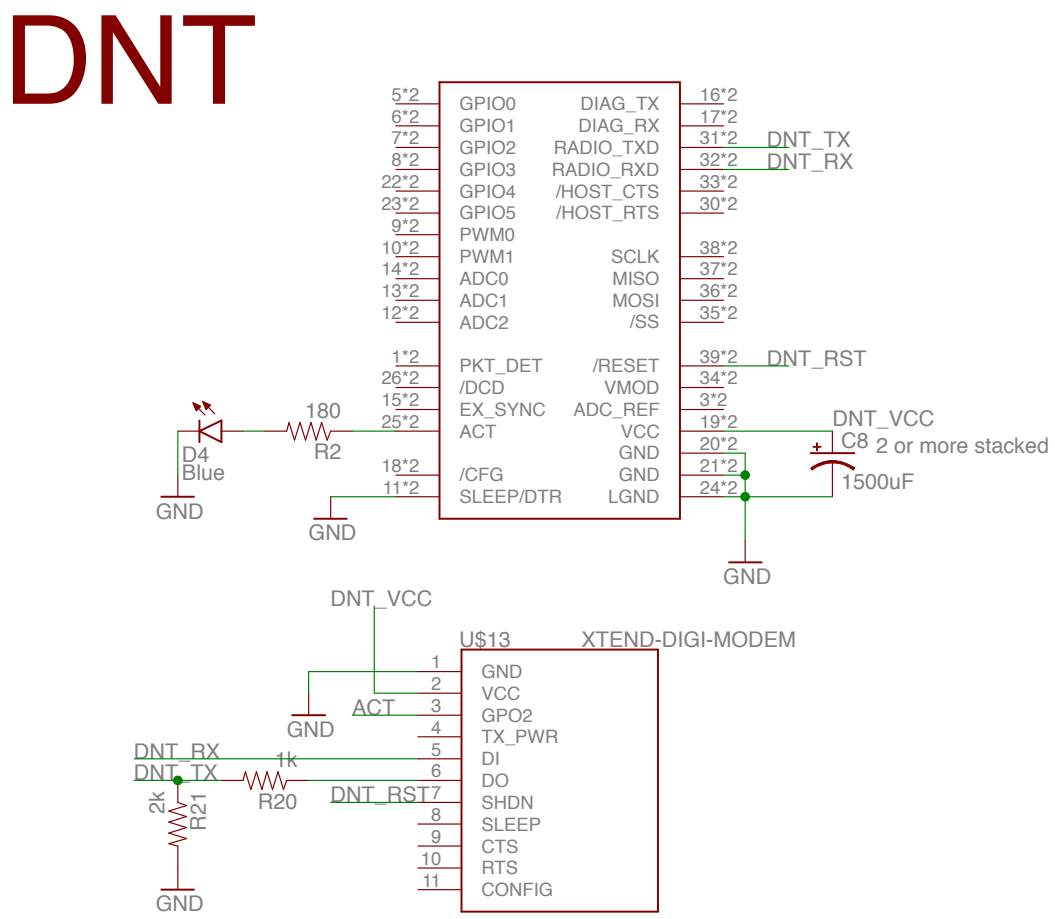


Figure 16: Radio Schematic

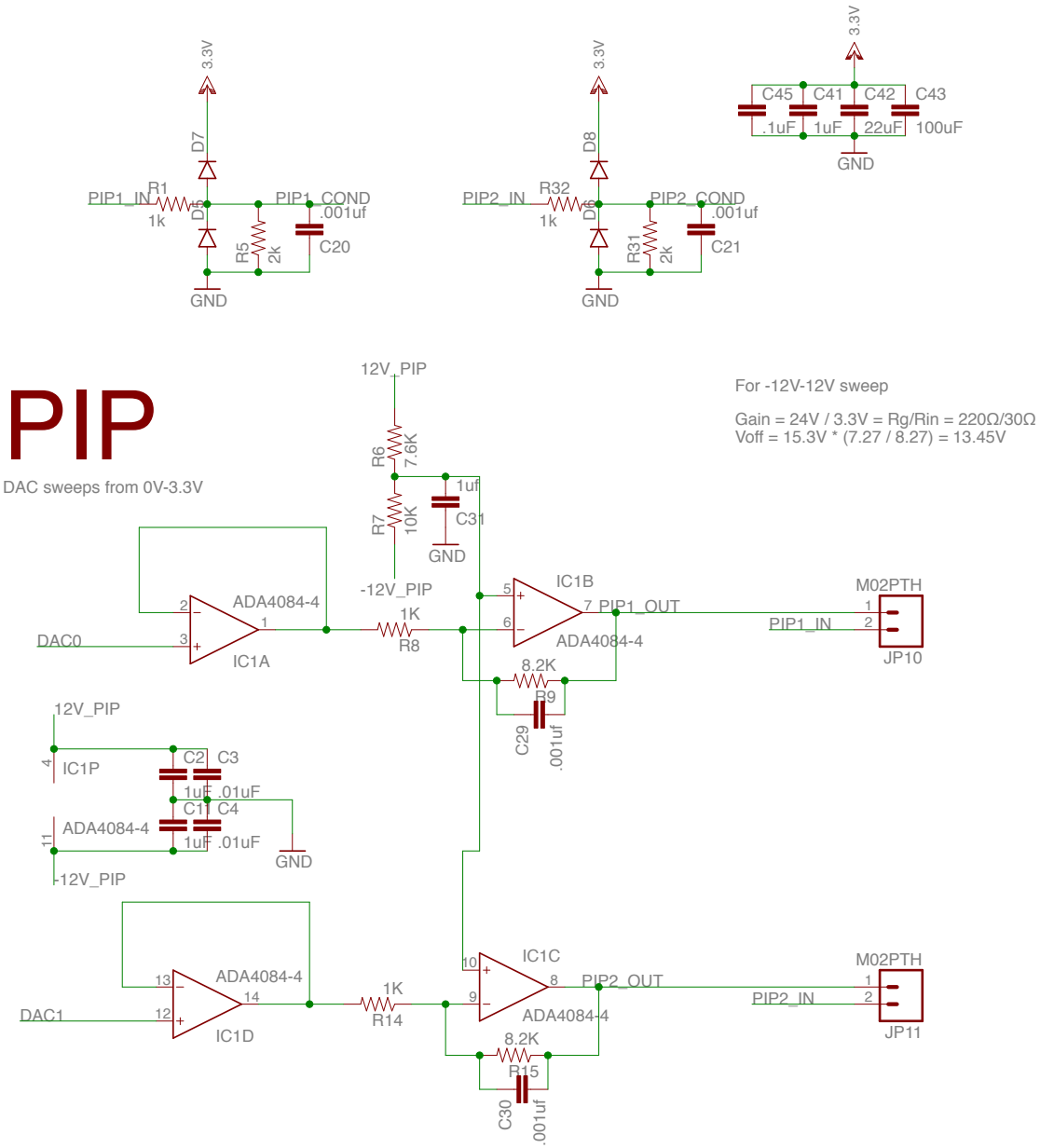


Figure 17: PIP IO Schematic

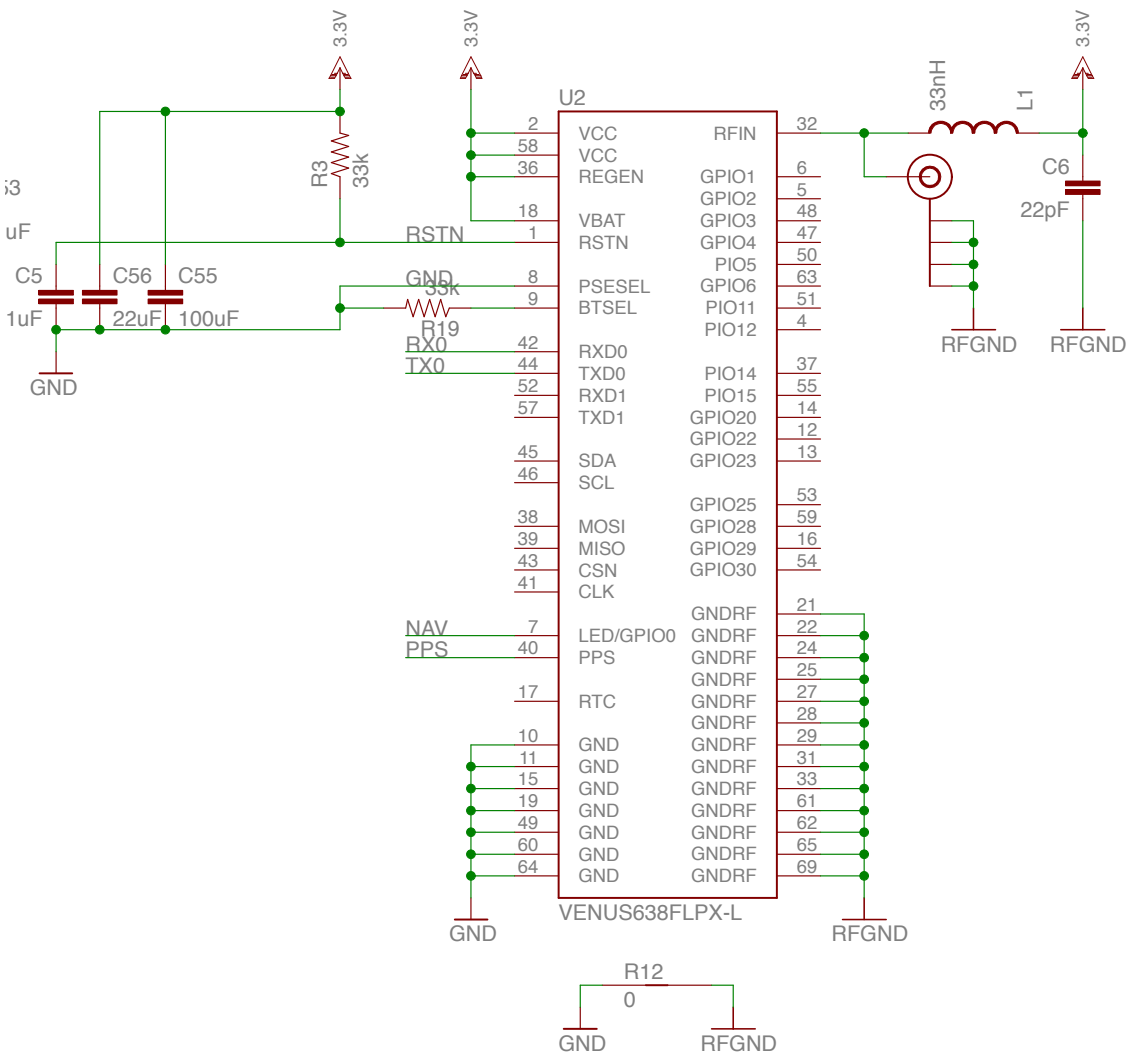


Figure 18: GPS Schematic

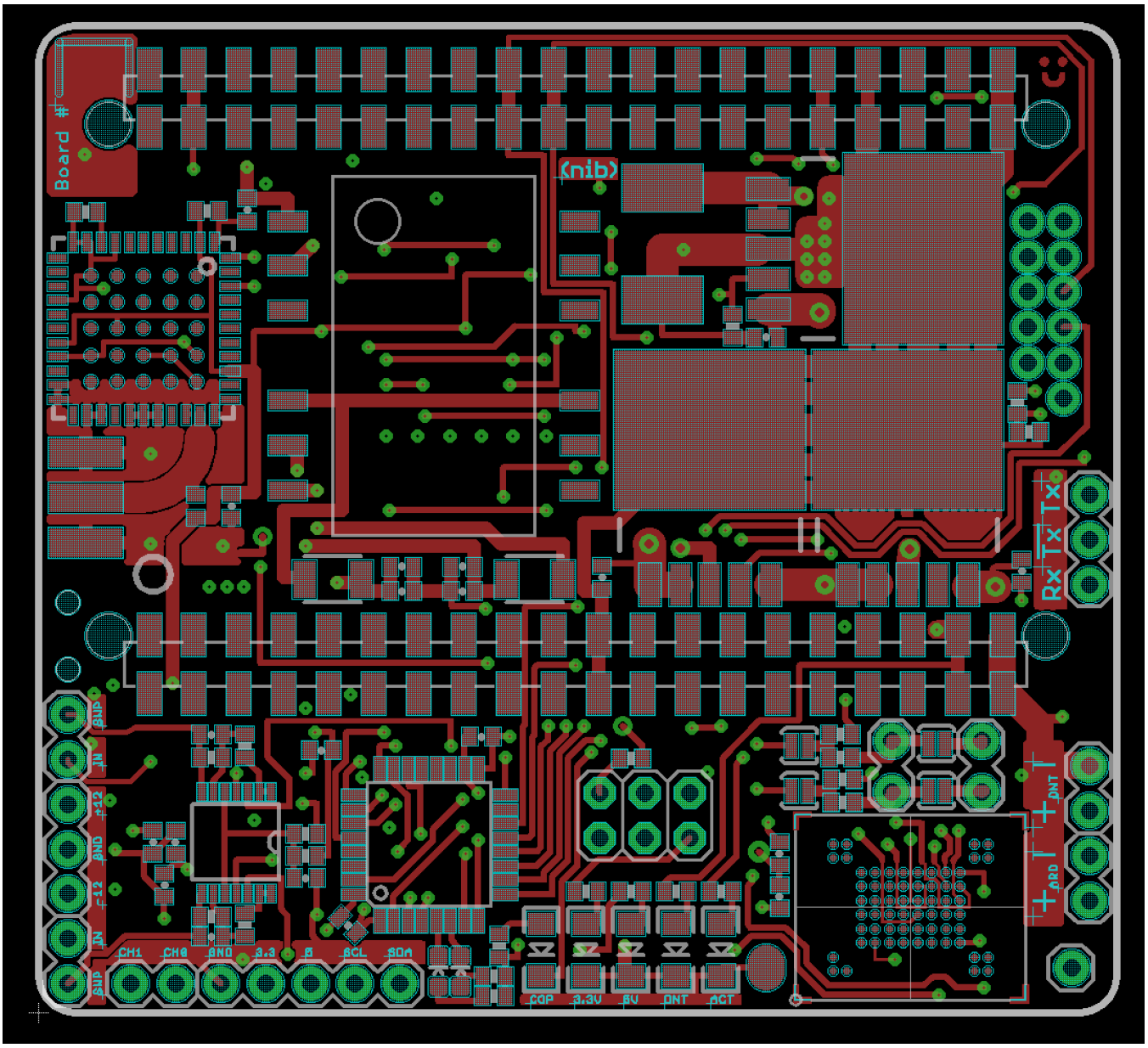


Figure 19: Top layout

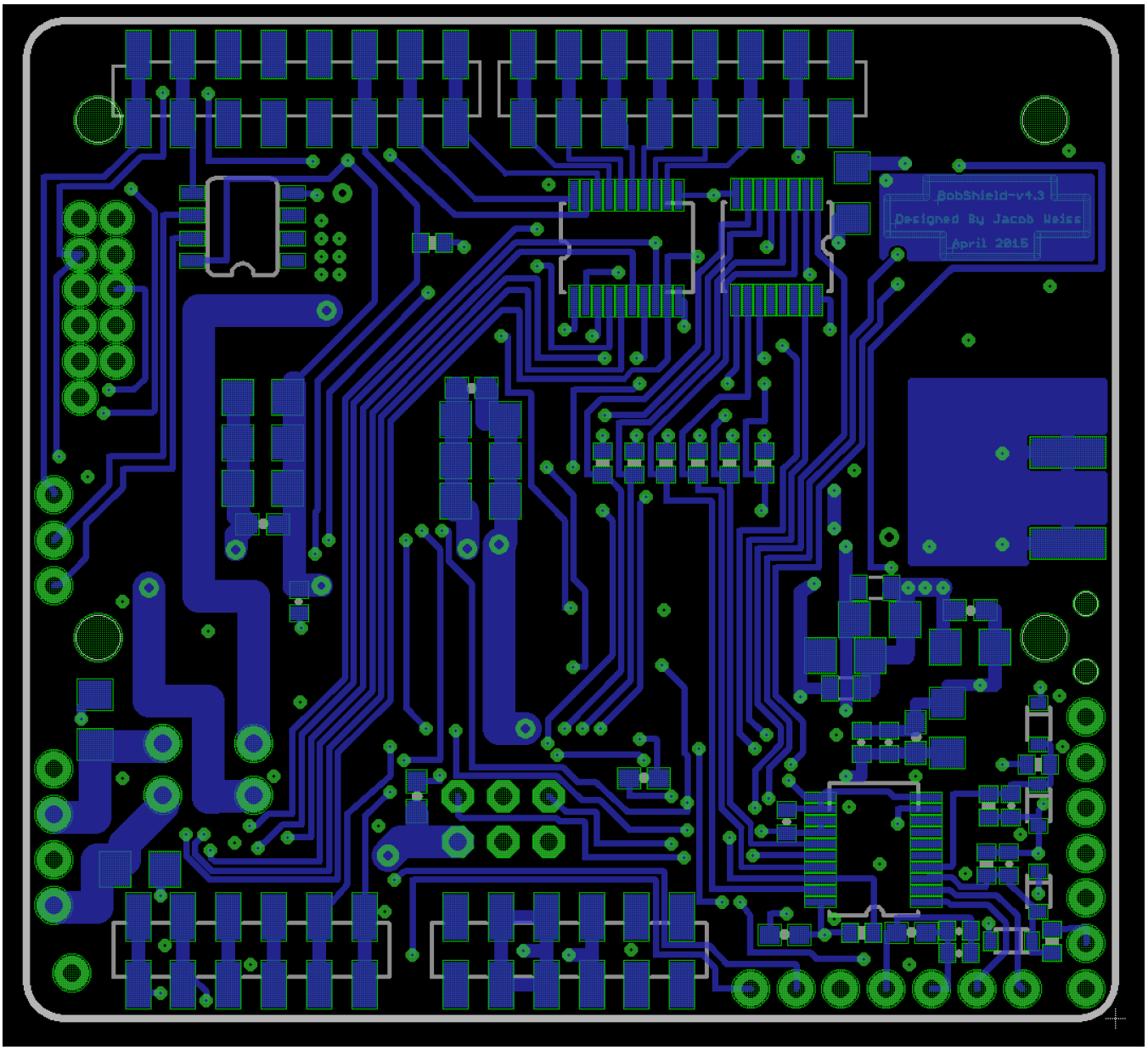


Figure 20: Bottom Tayout

2.4 Firmware API Reference

`BobShield()`

Constructs and initializes a BobShield object. The shield is powered off and back on in order to initialize it to a known state.

`uint8_t status()`

Return The status bitmask.

Returns the current status of the BobShield as a single byte of flags. Currently unimplemented.

`void configureSweep(bool pip0, bool pip1, uint16_t delay, uint16_t avg_num, uint16_t num_samples, uint16_t sweep_min, uint16_t sweep_max)`

pip0 Whether or not the configuration should be applied to pip0.

pip1 Whether or not the configuration should be applied to pip1.

delay The delay between stepping the DAC and reading the ADC.

avg_num The number of samples to average per step.

num_samples The number of steps to take in the sweep.

sweep_min The starting value for the sweep.

sweep_max The ending value for the sweep.

Configures the parameters of a sweep measurement. During a sweep, the DAC output is stepped from **sweep_min** to **sweep_max** in increments of $\frac{\texttt{sweep\_max}-\texttt{sweep\_min}}{\texttt{num\_samples}-1}$. After setting the DAC, it delays for a time period proportional to **delay** and then samples the ADC **avg_num** times, averaging the results. The resulting measurements are stored for later retrieval.

```
void sweep(bool pip0, bool pip1)
```

pip0 Should sweep pip 0.

pip1 Should sweep pip 1.

Performs the sweep configured by **configureSweep**. During the sweep, the Arduino will not be able to communicate with the coprocessor.

```
void sweepSendGet(bool pip0, bool pip1,
    bool toDNT, bool toMaster,
    uint16_t* sweep0, uint16_t* sweep1)
```

pip0 Should get pip0 data

pip1 Should get pip1 data

toDNT Send the data directly to the radio.

toMaster Send the data directly to the SPI master.

sweep0 A pointer to an array to return pip0 data to.

sweep1 A pointer to an array to return pip1 data to

Retrieves the sweep data from the coprocessor.

```
void dntReset()
```

Resets the DNT radio by powering it off, waiting for several seconds, and then powering it back on. This call will block until power has been restored to the radio.

```
uint8_t dntBytesAvailable()
```

Return The number of bytes in the receive buffer.

Returns the number of bytes that are waiting in the coprocessor's receive buffer.

```
void dntSendData(uint8_t* data, uint8_t length)
```

data	The data to send.
length	The length (in bytes) of data.

Sends **length** bytes of **data** over the radio.

```
uint8_t dntReceiveData(uint8_t* data, uint8_t max)
```

data	A pointer to an array to store the received data.
max	The most bytes you are willing to accept.
Return	The number of bytes actually received.

Retrieves at most **max** bytes of data from the radio and stores it into the given **data** buffer. If more than **max** bytes have been buffered, then only **max** bytes will be retrieved (the rest can be acquired with a future call to **dntReceiveData**). If fewer than **max** bytes have been buffered, then all are stored in **data**. The number of bytes actually received is returned. The returned value is always less than or equal to **max**.

```
uint8_t gpsSendGet(bool toDNT, bool toMaster, uint8_t* data)
```

toDNT	Should send the data to the radio.
toMaster	Should return the data to the arduino.
data	The buffer that the data is returned in (if toMaster = true).
Return	The number of bytes returned in the buffer.

The shield gets data from the GPS and buffers it once per second. Calling gpsSendGet will take that buffer and send it to the radio and/or the Arduino. If no GPS data has been buffered, then the function returns 0.

```
void flushBufferSPI()
```

Writes a bunch of 0x00 bytes to the SPI buffer to effectively reset the coprocessor into a known state.

```
void writeDAC(uint16_t data, bool ch0, bool ch1)
```

data	The 12-bit value to write
ch0	Should write this value to ch0.
ch1	Should write this data to ch1.

Writes a given value to the DAC. The output op-amp is inverted, so a large **data** will produce somewhere around -12V, and a small value will produce +12V.

```
void readADC(bool ch0, bool ch1,
    bool toDNT, bool toMaster,
    uint16_t* ch0Data, uint16_t* ch1Data)
```

ch0	Should read the ch0 of the ADC.
ch1	Should read ch1 of the ADC.
toDNT	Should send the data over the radio.
toMaster	Should return the data to the Arduino.
ch0Data	The 14-bit ADC reading of CH0.
ch1Data	The 14-bit ADC reading of CH1.

Takes and returns a reading from the MAX1147 ADC.

```
void setBaudDNT(BaudOptions baud)
```

baud	The choice of baud as defined by the BaudOptions enum.

Sets the baud between the coprocessor and the radio.

```
bool isReady()
```

Return Whether or not the shield is not busy.

Returns true if the shield is ready to communicate with the Arduino.

```
void waitUntilHigh()
```

Waits until the DDRY line from the coprocessor to the arduino goes high.

```
void waitUntilLow()
```

Waits until the DDRY line from the coprocessor to the arduino goes low.

```
void setLEDs(bool led0, bool led1)
```

led0 The state of LED0.

led1 The state of LED1.

Sets the states of the two user definable LED's on the shield.

```
double ardBatteryVolts()
```

Return The current voltage level of the Arduino's battery.

Reads the current voltage of the Arduino's battery. The measurment is returned in volts.

```
double dntBatteryVolts()
```

Return The current voltage level of the radio's battery.

Reads the current voltage of the radio's battery. The measurment is returned in volts.

`void waitForDntPower()`

Waits for the radio to be powered on.

`void on()`

Turns the shield on by enabling its regulator.

`void off()`

Turns the shield of by disabling its regulator.

2.5 NAND Flash API Reference

```
Flash(bool enable_write, bool restart_address_counter)
```

enable_write True to enable writing to the memory.

restart_address_counter True to restart the address counter.

Initializes the flash logging device.

```
void writeBytes(byte* bytes, int length)
```

bytes The data to write.

length The length of the data in bytes.

Writes **length** bytes of **bytes** to the flash device. If the cache gets filled up, then it is automatically transfered to the main array.

```
void cacheToArray()
```

Manually move the cached data into the main array.

```
void dumpArray(uint32_t startAddress, uint32_t endAddress, int amount)
```

startAddress Where to start dumping the data from.

endAddress Where to stop dumping data.

amount The dumber of bytes per page to dump.

Dumps the **amount** bytes from each page in the range [**startAddress**, **endAddress**).

`void dumpBeforeEnd(uint32_t numPages)`

numPages The number of pages to dump.

Dumps from the **numPages** most recently written to pages.

`uint16_t readID()`

Return The ID of the flash device.

Reads the ID of the NAND Flash device. If the device is functioning, the ID should be 11314.

`void restartAddressCounter()`

Restarts the address counter.

3 Summary

Bloon is Mac application that wraps a configurable real-time parser and an interactive real-time plotter in an easy to use graphical user interface. To find more information about Bloon, including some screencasts demoing its features, visit `http://www.bloonapp.com`.

References

[1] **Matlab**, *http://www.mathworks.com/products/matlab/*

[2] **Matplotlib**, *http://matplotlib.org/*

[3] **PyQtGraph**, *http://www.pyqtgraph.org/*

[4] **COSMOS**, *http://cosmosrb.com/*

[5] **MakerPlot**, *http://www.makerplot.com/*

[6] **MegunoLink**, *http://www.megunolink.com/*

[7] **Realtime Plotter**, *https://github.com/sebnil/RealtimePlotter*

[8] **Arduino IDE**, *https://www.arduino.cc/en/Main/Software*

[9] **IOComp Plot Pack**, *http://www.iocomp.com/*

[10] **KST**, *https://kst-plot.kde.org/*