– Independent Work Report Fall, 2015 –

A Cross-Platform Programmer's Calculator

Brian Rosenfeld
Advisor: Robert Dondero

Abstract

This paper details the development of the first cross-platform programmer's calculator. As users of programmer's calculators, we wanted to address limitations of existing, platform-specific options in order to make a new, better calculator for us and others. And rather than develop for only one-platform, we wanted to make an application that could run on multiple ones, maximizing its reach and utility. From the start, we emphasized software-engineering and human-computer-interaction best practices, prioritizing portability, robustness, and usability. In this paper, we explain the decision to build a Google Chrome Application and illustrate how using the developer-preview Chrome Apps for Mobile Toolchain enabled us to build an application that could also run as native iOS and Android applications [18]. We discuss how we achieved support of signed and unsigned 8, 16, 32, and 64-bit integral types in JavaScript, a language with only one numerical type [15], and we demonstrate how we adapted the user interface for different devices. Lastly, we describe our usability testing and explain how we addressed learnability concerns in a second version. The end result is a user-friendly and versatile calculator that offers value to programmers, students, and educators alike.

1. Introduction

This project originated from a conversation with Dr. Dondero in which I expressed an interest in software engineering, and he mentioned a need for a good programmer's calculator. Dr. Dondero uses a programmer's calculator while teaching Introduction to Programming Systems at Princeton (COS 217), and he had found that the pre-installed Mac OS X calculator did not handle all of his use cases. When Dr. Dondero wanted to subtract a larger hexadecimal number from a smaller one (as his students do in one of their programming assignments) the calculator would return a bad

result.[1] Instead of using the OS X calculator, he continued to use his handheld Casio CM-100 calculator, which was first introduced in 1986 [9]; however, with COS 217 switching from a 32-bit architecture to a 64-bit one this year, the 32-bit calculator was no longer sufficient. As a result, Dr. Dondero wrote "bobcalc," a postfix, stack-based, command-line calculator that he considered to be a "temporary hack." While Dr. Dondero found it easier to implement the calculator with postfix notation, he noted that his students "think" using infix notation. He thought a well-designed programmer's calculator would be both useful for students working on this assignment and as an educational tool for teaching students how computers represent numbers.

Unable to find a calculator that ran on multiple platforms and wanting more out of existing applications, we decided to build our own programmer's calculator (Sections 2 and 3). To ensure that our application was user-friendly, we employed a rigorous human-computer interaction approach. In this paper, we describe the development process of the calculator:

- We introduce cross-platform development and we explain the decision to build a Google Chrome Application (Sections 4 and 5). We illustrate how using the developer-preview Chrome Apps for Mobile Toolchain enabled us to build an application that could also run as native iOS and Android applications.
- We show the usability techniques that led to the development of a prototype (Section 6).
- We discuss how we achieved support of signed and unsigned 8, 16, 32, and 64-bit integral types in JavaScript, a language with only one numerical type (Section 7).
- We demonstrate how we adapted the user interface for different devices (Section 8).
- We present the calculator's functionality and report back on our initial distribution efforts (Sections 9 and 10).
- We describe our usability testing and explain how we addressed learnability concerns in a second version (Sections 11 and 12). We compare this calculator to existing options and we assess its usability (Section 13).

[1] Subtractions that should produce a negative result (ex. $0 - 1$) always yield a 64-bit number with all ones, except the highest-order bit, which is set to 0.

2. Background

As the name indicates, a "programmer's calculator" is intended for programmers and others who work with computers. While there is no formal definition of a programmer's calculator, we loosely use the term to refer to calculators that model their behavior after that of a computer. As do some scientific calculators, a programmer's calculator should support different numerical bases; however, unlike scientific calculators, programmer's calculators should also use at least one kind of integral data type and offer operators that work with the numbers' underlying bits. In addition to programmers and systems engineers, computer-science students and educators can make valuable use of programmer's calculators.

During the 1980s, Hewlett-Packard and Casio each sold physical programmer's calculators. The HP-16C, released in 1982 by Hewlett-Packard, was designed for debugging programs [24]. It used reverse Polish notation and was programmable. It supported arbitrary word sizes up to 64 bits and worked with 1's complement, 2's complement, and unsigned numbers. This calculator was discontinued in 1989 (presumably due to poor sales) and remains HP's only programmer's calculator. Decades later, some programmers still use their original HP-16Cs [22]. Four years after the release of the HP-16C, Casio released its own programmer's calculator, the CM-100, which it referred to as a "computer math calc" [9] and is still used by Dr. Dondero today.

Unlike these early programmer's calculators, current options take the form of software applications rather than physical devices; instead of purchasing new hardware calculators, users prefer to run software calculators on the devices that they already own. Each of these software calculators, however, is platform-specific, running on only one of Windows, Mac OS X, Linux, Chrome OS, Android, iOS, or the web. Moreover, these calculators feature various limitations. Through our new, cross-platform programmer's calculator we sought to address these issues while creating an application that could be available to users on all of their devices.

3. Existing Programmer's Calculators

3.1. Computer calculators

The Mac OS X, Windows, and Linux operating systems come with pre-installed calculators that include a "programmer" mode [5] [29] [16].[2] All three operating systems' calculators feature some degree of binary, octal, decimal, and hexadecimal input and output and each provides a comprehensive set of operators. However, the OS X and Linux calculators only support 64-bit, unsigned numbers, and the Windows calculator, despite offering four different word sizes, only supports signed numbers. Meanwhile, the Chrome OS calculator, which can also be run as a Chrome Application on computers with the Chrome browser, includes neither a programmer mode nor any other programming-related functionality [21].

3.2. Mobile and web calculators

Independent developers have also written programmer's calculators for mobile and the web. A search for "programmer's calculator" in the Google Play Store,[3] shows one calculator that has surpassed 50,000 downloads and three others that are in the 10,000 to 50,000 range [17]. The top calculator—billed as a calculator for developers—fails our earlier, loose criteria for a programmer's calculator since it does not support any bitwise or logical operators [3]. Moreover, it is limited to 32-bit, signed numbers. Meanwhile, the other popular calculators, despite offering bitwise and logical operators, also fail to offer a complete set of integral modes. For example, one calculator only allows signed integral types [26] while another only supports 32-bit numbers [30]. Nevertheless, a less-popular (but well-designed) Android calculator features individual bit toggling, a floating point mode, and offers signed and unsigned 8, 16, and 32-bit integers; yet, for 64-bit integers, it only supports unsigned numbers [8]. On iOS, we found a useful, paid ($2.99) programmer's calculator that supported 8, 16, 32, and 64-bit signed and unsigned numbers [14]; however, the free offerings on iOS were disappointing due to limited options for modes. For example, the calculator with the

[2] For Linux we are referring to the GNOME Calculator (gcalctool) that is included with some distributions of Linux.
[3] The market for Android applications.

highest number of reviews (and presumably the most downloads), only supports 64-bit numbers and does not allow for binary input [43]. Lastly, Penjee.com, a website for learning how to code, includes an online programmer's calculator that supports signed and unsigned numbers up to 128 bits [39]; however, as a web-based calculator, this calculator cannot be used offline, and on mobile, it renders poorly and is difficult to use.

4. Cross-Platform Development

4.1. Overview

Cross-platform software refers to "software that exists in different versions so that it is available on more than one platform" [6]. An example of cross-platform software would be an email client, like Microsoft Outlook or Gmail, that is available as a web application and as native applications on Android and iOS (among other platforms). On the contrary, an application like the Windows calculator would be considered platform-dependent, for it can only run on Windows computers. Clearly, the advantage of cross-platform software over platform-dependent software is the multi-platform availability. For applications like an email client, this wide-spread availability is almost essential. Meanwhile, for calculators and other similar applications, this availability makes the application more useful to existing users and accessible to new ones.

In the absence of cross-platform tools or frameworks, developing for multiple platforms would consist of writing native code for each target platform. In many cases, these platforms require different skill sets. For example, iOS applications are written in Objective-C and C while Android applications are written in Java. Charland and LeRoux (2011) describe the skill sets needed to develop software for nine mobile operating systems [11]. To approach this process with a small team would be both difficult and time-intensive while using a large team with varied skill sets would be expensive.[4] Nevertheless, there are advantages to using native code. Notably, native

[4] In 2009, Google's engineering vice president Vic Gundotra predicted that aided by economic reasons, the browser would become the dominant platform and would surpass app stores. He said, "What we clearly see happening is a move to incredibly powerful browsers. Many, many applications can be delivered through the browser and what that does for our costs is stunning. We believe the web has won and over the next several years, the browser, for economic reasons almost, will become the platform that matters and certainly that's where Google is investing." [36]

applications have better performance and provide access to lower-level APIs. Moreover, creating a unique application for each platform allows the developer(s) to customize the user interfaces to match platform conventions.

Because of the difficulty, cost, and time involved in developing a unique application for each platform, developers have turned to approaches that allow them to work with one development process and shared source code. Aside from the time and cost savings of writing less code, a smaller code base is frequently easier to maintain [11]. The simplest approach for cross-platform development is to write a web application. The mobile web is inherently cross-platform, for, as Charland and LeRoux[5] explain, "The only thing [mobile operating] systems have in common is that they all ship with a mobile browser that is accessible from the native code" [11]. In fact, using "open Web technology" was the initial plan for third-party iPhone apps until the mobile Web fell out of favor due to the superior performance of native apps. Charland and LeRoux argue that such a comparison is unfair, highlighting two problems: first, the expense of writing a native app for each platform, and second, the "negligible or unnoticeable performance penalty in a well-built business application using Web technology." They foresaw hybrid applications as the likely outcome of the native-vs.-web debate, and we sought such an approach in developing our application.

4.2. Options

Ohrt and Turau (2012) and Marius (2013) provide overviews of cross-platform mobile development frameworks [37][28]. One approach, PhoneGap, produces an interpreted, hybrid application by running a JavaScript web application within a native shell. Meanwhile, other approaches run natively and/or make use of native user-interface elements. For example, Xamarin uses shared C# business logic but makes use of native user interfaces by requiring different user-interface code for

[5]At the time of publication, Charland was the co-founder and CEO at Nitobi Inc., and LeRoux was Nitobi's lead architect. LeRoux also led Nitobi's PhoneGap project. As the two explain, "Phone Gap is an open source framework that provides developers with an environment where they can create apps in HTML, CSS, and JavaScript and still call native device features and sensors via a common JS API. The PhoneGap framework contains the native-code pieces to interact wth the underlying operating system and pass information back to the JavaScript app running in the Webview container" [11]. Later that year, Nitobi was acquired by Adobe, and the two companies jointly donated the PhoneGap code base to the Apache Software Foundation where it became Apache Cordova. Today, PhoneGap refers to Adobe's distribution of Apache Cordova [7]. The Chrome Apps for Mobile toolchain used to build the calculator in this paper is also based on Apache Cordova [18].

each platform. We refer the reader to these papers and vendor websites for more information on mobile development frameworks.

4.3. Academic Work

Lastly, researchers have proposed domain-specific languages (DSLs) for cross-platform development. For example, Heitkötter, Majchrzak, and Kuchen (2013) introduced md^2, "an approach for model-driven cross-platform development of apps" [23]. Intended for business applications with form-based user interfaces, md^2 uses a high-level DSL to describe the applications. Code generators then transform the high-level description into source code for Android and iOS, which the developer can them compile and run normally. Similarly, Macos and Solymosi (2013) proposed ScaMo, a DSL for cross-platform development that consisted of the programming language Scala along with a DSL defined in Scala [27]. ScaMo would make use of the Scala compiler to produce the mobile applications. Because these approaches were not as well-developed as those referred to in the previous section, they were not considered viable approaches for our project.

5. Approach

The key idea behind our calculator was developing it as a Google Chrome Application and using the developer-preview Chrome Apps for Mobile Toolchain. Chrome Apps are built with HTML5, CSS, and JavaScript [20]. They are installable through the Chrome Web Store and can run anywhere that Google Chrome runs, including Windows, Linux, and Mac OS X. Chrome Apps also run as native applications on Chromebooks—laptops that run Chrome OS. Unlike traditional web applications, Chrome Apps run in their own windows (outside of the browser), can be used offline, and can access the file system.

The developer-preview toolchain enables developers to run Chrome Apps on Android and iOS. Based on Apache Cordova, the toolchain wraps the application's web code within a native application shell, producing hybrid applications that can be distributed through Google Play and/or

the Apple App Store [18].[6] Moreover, because our calculator does not make use of the file system, we can take the HTML, CSS, and JavaScript source code and host it as a web-application, making it available through a browser as well. Figure 1 presents a high-level view of how the calculator will be targeted to different platforms.

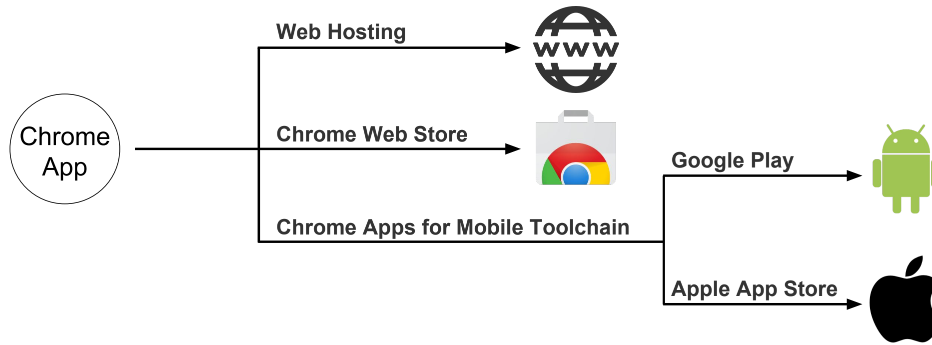


Figure 1: Built as a Chrome App and using the Chrome Apps for Mobile Toolchain, our calculator can also be used as a webpage and hybrid Android and iOS apps.

Existing, platform-dependent programmer's calculators are presumably written as native applications and implemented in languages like C, Objective-C, and Java. We, however, by building a Chrome Application, chose an approach that involves using web technologies like HTML, CSS, and JavaScript. At first consideration, this approach does not seem conducive to developing a programmer's calculator, for JavaScript only includes one numerical type—a double-precision 64-bit format IEEE 754 value [15]. However, once we achieved support of variable-length, signed and unsigned integral types, we were able to take advantage of the mobile web's portability, sharing the entire source code across platforms and writing an adaptive user interface.

As explained earlier, some alternative approaches to cross-platform development use platform-specific user interface code, resulting in applications with native user-interface elements. Because

[6]On August 10, a member of the Mobile Chrome Apps team updated the README in the mobile-chrome-apps Github repository, adding that "[the toolchain] is no longer being actively developed. We intend to keep it functional but do not intent on adding any new features." In response to a developer's request for more information, the author of the commit explained, "Most of the Chrome Apps for Mobile enhancements have been upstreamed to Cordova itself, and most chrome app plugins work with plain Cordova projects" [4]. As long as the toolchain is kept functional, our approach will continue to work. Even with a non-functional toolchain, the upstreamed enhancements mean our approach should work with Apache Cordova substituted for the toolchain. In our case (and for similar projects), it would definitely work because we do not use any plugins. As speculated at `https://groups.google.com/a/chromium.org/forum/#!topic/chromium-apps/EFnfQur6Beo`, I suspect the decision to suspend development is related to Google's plans to run Android apps on Chrome OS, which must have won out over running Chrome Apps on Android.

we are not using tabs, menus, or other built-in widgets, we did not think a native user interface was necessary for our calculator. Instead, we liked the cross-device consistency that came with shared user-interface code. Moreover, we did not consider performance to be critical, for a calculator is inherently limited by the processing speed of its human users. While adding two 64-bit integers may be time-intensive for a human, such an operation (assuming it's implemented properly) requires relatively little effort for a computer. Accordingly, we felt that the speedup achieved with a native application would be negligible from the perspective of the user, for both native and web-based implementations would appear instantaneous and equally responsive.

6. Implementation

In developing the calculator, we emphasized usability and software robustness. We used the human computer interaction techniques presented in Rogers, Sharp, and Preece (2011) [41] and also followed software-engineering best practices like unit testing. Below, we present a high-level overview of the development process. Later, we will explore areas of interest in more depth.

6.1. Evaluating Requirements

First, we sought to identify our users' needs and prepare a list of requirements for the calculator. We surveyed existing calculators, spoke to students, and drew on our own experiences taking and teaching systems courses at Princeton. Using this information, we created scenarios (Appendix A) to describe specific tasks users would like to accomplish with the calculator. A sample scenario reads:

> A COS 217 student is working on the buffer overrun assignment. He wants to calculate the hexadecimal offset needed for a jump. He takes the destination address and subtracts the address of the instruction following the jump. Because the destination address is earlier in memory, this calculation gives a negative jump offset and causes difficulties.

We then prepared a detailed list of functional requirements (what the system should do) and non-functional requirements (constraints on the system and its development) [41]. For example, a

functional requirement would be that the calculator must add two numbers, and a non-functional requirement would be that the calculator must run on both Android and iOS. While the full list of requirements is included in Appendix B, the highest-priority functional requirements were as follows:

- Standard, non-programming functionality: add, subtract, times, divide
- Bitwise operators: and, or, not, xor
- Bit shifts: left and right shifts
- Binary, decimal, and hexadecimal input and display
- Clear, accumulator clear, and delete
- 64-bit support
- Signed and unsigned support

Although some of the other calculators we mentioned earlier only support numbers up to 32-bits, we set 64-bit support as a high-priority requirement. At a minimum, we wanted to accommodate the architecture used in COS 217. We placed a requirement for supporting 8, 16, and 32-bit numbers in the next-highest priority category. We also included lower-priority requirements like keyboard input and shortcuts in a “time-permitting” category.

6.2. Prototyping

Next, we used the requirements to develop prototypes of the calculator so that we could choose between user-interface alternatives. Starting with low-fidelity prototypes that were quick and easy to produce, we sketched various options for the calculator’s mode bar, display, and keypad (Appendix C), and we prepared storyboards (Appendix D). After we evaluated these designs, we prepared a high-fidelity prototype (Appendix E), writing a user interface with HTML, CSS, and JavaScript.

6.3. Application Structure

We wrote both the calculator and view logic in JavaScript. The calculator logic consists of a calculator class and number utility methods that use the open-source BigInteger.js library [40].

Although our final application only offers 8, 16, 32, and 64-bit numbers, the calculator logic supports integral types of any length. Moreover, the calculator can be configured for arbitrary bases. The calculator logic is completely independent of the view and could be used to support multiple different user interfaces.

The view provides an interface for manipulating the calculator's state. Upon user input, the view makes the appropriate call to the calculator and then updates the interface. The view consists of a view class and utility methods that use JQuery for DOM manipulation. To adapt the view to different screen sizes and resolutions, we used the CSS3 Flexible Box layout mode and CSS3 media queries.

In all, we wrote roughly 1,100 lines of JavaScript, 400 lines of CSS, and 160 lines of HTML. The source code can be viewed at `https://github.com/brosenfeld/calculator`.

6.4. Unit Testing

In addition to manual testing, we used unit testing to verify the core calculator functionality, reaching 100% code coverage. Because we wrote our own code for working with integral types in JavaScript, we wanted a quick, reliable way to write new tests and do regression testing. We worked with QUnit, a JavaScript unit testing framework and Blanket.js, a JavaScript code coverage library that is compatible with QUnit.

7. Integral Types in JavaScript

7.1. Problem

As noted earlier, JavaScript only includes one numerical type—a double-precision 64-bit format IEEE 754 value [15]. This data type can represent integers with magnitude up to 2^{53}; however, bitwise operators and shifts only work with numbers in the range $[-2^{31}, 2^{31}-1]$.[7] Accordingly, we could have simply implemented a fixed-length, 32-bit calculator using JavaScript's built in Number

[7] According to the ECMAScript Language specification (the standard for JavaScript), "Some ECMAScript operators deal only with integers in specific ranges such as -2^{31} through $2^{31}-1$, inclusive, or in the range 0 through $2^{16}-1$, inclusive. These operators accept any value of the Number type but first convert each such value to an integer value in the expected range" [15].

type; however, because 64-bit support was one of our high-priority requirements this was not a viable option.

7.2. Alternative Solutions

We looked into working with Google's Native Client,[8] which would have enabled us to use compiled C code as part of a Chrome Application. As introduced in C99, the `stdint.h` header file provides a set of typedefs for exact-width integer types [47]. Using these types for signed and unsigned integers would have let us take advantage of C casts. However, the Native Client feature of Chrome Apps is not available to the mobile apps built with the Chrome Apps for Mobile Toolchain [20]. Because we were aiming for mobile support and had included running on Android and iOS as a non-functional requirement for the calculator, we chose not to pursue the Native Client approach.

Another approach could have been to write a new library that did exactly what we wanted (and no more). Since we planned to work with integers no larger than 64 bits, and because performance was not critical, this implementation did not need to be efficient or optimized. A possible implementation could have used an array where each index corresponded to that number bit. While this approach would have worked, it would have required implementing and testing each operation. Rather than start from scratch, we decided it would be more time-efficient and practical to use a BigInteger implementation and modify or extend it as needed.

7.3. Solution: Using a BigInteger library

In the end, we used the open-source BigInteger.js library [40], which supports arbitrary-length integers. As compared to the exact-width integers declared in C99's `stdint.h`, these integers have seemingly infinite width. Additionally, these integers are represented with separate sign flags and values, and the library does not differentiate between signed and unsigned integers. The advantage to using this BigInteger library was that it included almost all of the operators we wanted. So while significant development work was needed to support signed and unsigned, exact-width integers,

[8] As explained on its welcome page, "Native Client is a sandbox for running compiled C and C++ code in the browser efficiently and securely, independent of the user's operating system" [19].

using the library would save significant development time on the operators. To achieve this support:

- We wrote an instance method for changing the calculator's bit length and another for changing the calculator from signed to unsigned and vice versa. In addition to updating the calculator's configuration, these methods cast the accumulator and the operand. [9]
- We wrote a method, `keepInBounds`, for handling overflow and underflow. We call this method internally on the result of every operation.
- We handled signed input by using the concept of a signed upper bound—the maximum, positive signed number for that bit length. When the user enters the bit representation of a signed number, the calculator checks the input against the signed upper bound. Once the signed bound is reached, the calculator converts the operand to the appropriate negative value. For example, in 8-bit signed mode, after the user enters 'F', the hex display will show F, and the decimal display will show 15. Upon entering the second 'F', the hex display will show FF, and the decimal display will show -1 rather than 255.
- We combined our casting code and the BigInteger library's `toString` method in order to generate hexadecimal and binary representations. While, the BigInteger library does not include methods for getting the binary or hexadecimal representation of a number, there is a `toString` method that accepts a base as an argument. This `toString` method converts the value to the appropriate string and then appends a minus sign to the front if the sign boolean is true. Accordingly, `x.toString(2)` and `x.toString(16)` will only produce correct bit representations for positive numbers.[10] So, to get the correct bit representation for a negative number, we could convert it to the positive number with the same bit representation and then use `toString` with that positive number. For example, we could convert the 8-bit signed value -1 to 255. This transformation is equivalent to casting a signed number to an unsigned number, allowing us to reuse our our existing casting code.

[9] The second argument in a binary operation.

[10] To clarify, this is how the `toString` method should work. It is not a bug. Rather, `x.toString(2)` and `x.toString(16)` are not aliases for `toBinary` and `toHex`.

Lastly, in addition to the operators we used from the library, we implemented logical right shift and rotate left operators. The library already supported a left shift and an arithmetic right shift, so we used these operators in implementing our new ones.

8. Building a Flexible User Interface

8.1. Problem

In developing our calculator, we wrote one user interface to be used across all devices. This meant that phones, tablets, and computers would all run same user interface code. As shown in Table 1, these devices vary greatly in physical size, resolution, and density. For example, a Google Nexus 9 tablet has a higher resolution than a MacBook Pro despite the latter's larger screen. Moreover, we planned to allow our users to resize the Chrome App so that they could work with a window size of their choice. Prioritizing usability, we wanted our interface to resize gracefully and to adjust to the different dimensions of phones, tablets, and computers.

Device	Display (inches)	Width (pixels)	Height (pixels)	Density (pixels / inch)
iPhone 6s	4.7	750	1334	326
Google Nexus 9	8.9	2048	1536	288
MacBook Pro	13.3	1280	800	113

Table 1: These are sample display sizes and dimensions for an iPhone 6s, a Google Nexus 9 tablet, and a 13-inch MacBook Pro (without Retina) computer [44]. In addition to varying in physical screen size, these devices have different pixel densities.

Because of these differences, we could not use fixed font sizes: while a 12 px font may be appropriate for an iPhone 6s, it would occupy only a fraction of the space available on a MacBook Pro. Moreover, we needed the calculator's displays, buttons, and keypads to adapt to the different screen sizes, precluding the use of fixed widths, heights, margins, and padding.

8.2. Alternative Solutions

A potential solution could have involved using the Bootstrap grid system. Bootstrap is a free, open-source front-end framework for web development, and it offers a responsive, 12-column grid system for creating page layouts [48] [1]. We decided against this option because we wanted more

flexibility and finer granularity than a 12-column system. Another option involved using JavaScript to listen for resize events and to calculate and set new layout and text sizes. This option would have been messy to implement and error-prone, so we did not pursue it.

8.3. Solution: Flexible Box Layout Mode

To adapt the calculator's layout to various resolutions, we used the CSS3 Flexible Box layout mode defined in the CSS Flexible Box Layout Module specification.[11] A flex container alters its children's widths and/or heights to fill the device's available space, expanding them to fill unused space and shrinking them to prevent overflow. By setting an item's `flex-grow` and `flex-shrink` (or the shorthand `flex` property), a developer specifies how the item should resize [32][12]. For example, if all items have the same `flex-grow` value, they will equally split the remaining space in the parent container; alternatively, if an item has a `flex-grow` property of 1 and all of the others have a value of 0, then that item will receive all of the extra space.

We used flex layouts throughout the application, specifying how we wanted the interface to use device space. For example, when the screen has extra vertical space, the displays and keypad divide it equally. Among the displays, the binary display receives twice the additional space of the decimal and hexadecimal displays, and within the keypad, extra space is distributed evenly among the buttons.

8.4. Solution: Media Queries and Viewport Units

While flex layouts helped to manage the calculator's layout, we still needed to resize text. For this, we composed CSS3 media queries into piecewise, continuous functions over a screen's width and height, such as the one in Figure 2. Media queries let developers specify custom styling for

[11]This specification has the status of a "Last Call Working Draft" and is supported by recent versions of Firefox, Chrome, Internet Explorer, Opera, and Safari. Some older browser versions use a previous draft of the specification, so we included redundant CSS styles to address those as well. For example, Chrome versions from 21.0 (inclusive) to 29.0 (not inclusive) prefixed flexbox styles with `-webkit`. [32]

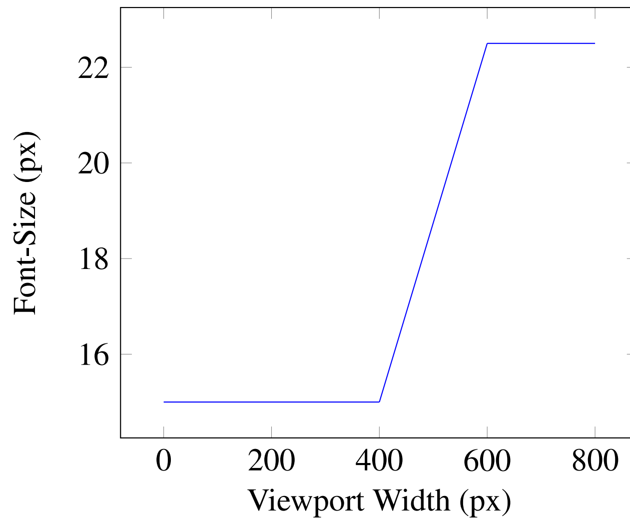


Figure 2: A plot of how font-size could resize with changes in the viewport width, using a piecewise, continuous function.

media (ie. screens or devices) that meet certain criteria [2].[12] We queried on the screen's width and height, dividing the input space into ranges and setting a fixed or dynamic font-size for each range. To dynamically resize text, we used viewport units, which express size as a percentage of the viewport's (window's) width and height or their minimum or maximum [13]. To allow for smooth text resizing, we ensured that font-size was continuous across ranges. We provide example code in Figure 3.

We also used media queries to control when to show a number's binary representation as four 16-bit lines (narrower screens) or two 32-bit lines (wider screens), and we used viewport units for buttons' padding in order to increase button size on larger screens. We present an example of two different size windows in Appendix H.

9. Functionality

We achieved all of the high and medium-priority functional requirements identified in our previously-mentioned requirements analysis (Appendix B). We also satisfied two of our "time-permitting"

[12] As explained in the Media Queries W3C Recommendation, "A media query consists of a media type and zero or more expressions that check for the conditions of particular media features. Among the media features that can be used in media queries are 'width', 'height', and 'color'. By using media queries, presentations can be tailored to a specific range of output devices without changing the content itself" [2].

```
@media all and (max-width: 400px) {
  .text { font-size: 15px; }
}

@media all and (min-width: 400px) {
  .text { font-size: 3.75vw; }
}

@media all and (min-width: 600px) {
  .text { font-size: 22.5px; }
}
```

Figure 3: In this simplified, three-case example, we implement the function shown in Figure 2. First, we set a minimum font-size to be used for viewport widths up to 400px. Then, over the range between 400px and 600px, we set the font-size using viewport units so that the font-size will be 3.75% of the viewport's width and increase linearly from 15px to 22.5 px. Lastly, for viewports wider than 600px, we set a fixed font-size of 22.5px so that the text would not grow to be too large.

goals: circular shifts and keyboard input and shortcuts.[13] Moreover, we added a logical right shift in addition to the arithmetic right shift. In all, the calculator:

- Supports 8, 16, 32, and 64-bit signed and unsigned numbers
- Features 17 operators (Appendix I).
- Offers keyboard input and shortcuts on computers.
- Accepts binary, decimal, and hexadecimal input, helping the user by disabling invalid digits for the active mode.
- Simultaneously displays binary, decimal, and hexadecimal output, showing both the accumulator and operand for binary operations.
- Handles user errors like divide by zero or shifting by a number larger than the bit-length, providing the choice of resetting the calculator or undoing the error-causing operation.

10. Distribution

On December 1, we made the calculator available to students in COS 217 for their buffer overrun assignment, which involved hexadecimal calculations and inspired the first scenario listed in

[13]For circular shifts, we added a rotate-left shift that becomes a rotate-right shift when given a negative operand.

Appendix A. We published the calculator as a webpage and listed it publicly in the Chrome Web Store. Between December 1 and January 4, the calculator was installed 55 times.[14] We provide access instructions in Appendix F.

11. Initial Usability Evaluation

As defined in the ISO 9126 standard for evaluating software quality, usability is "the capability of the software to be understood, learned, used and attractive to the user when used under specified conditions" [25]. In order to identify usability problems with the calculator, we employed a three-part usability evaluation, conducting a streamlined-cognitive walkthrough, a thinking-aloud study, and a heuristic evaluation. In Appendix G, we include a screenshot of the interface at this time.

11.1. Streamlined Cognitive Walkthrough

A streamlined cognitive walkthrough is a structured approach to evaluating a software interface's learnability—a factor of usability, formally defined as "the capability of the software product to enable the user to learn its application" [25]. The evaluator completes sample tasks, and at each step answers two questions intended to assess the application's learnability and usability [42]:[15]

1. Will the user know what to do at this step?
2. If the user does the right thing, will they know that they did the right thing, and are making progress towards their goal?

In our cognitive walkthrough, we used four tasks that were designed to cover the calculator's full functionality and to do so as efficiently as possible. We found that the interface satisfied the second question for all steps, always keeping users informed of the system's status; however, we noticed the following learnability issues when answering the first question:

[14] During this period, I was contacted by a classmate who had been looking for a programmer's calculator for his Chromebook and discovered our application, which was the first one to show up. Only after installing it, did he see my name on the listing and contact me. He explained that for COS 375 (Computer Architecture), he was looking for the value of a specific bit in a 32-bit number and did not want to do the math by hand.

[15] A standard cognitive walkthrough involves four questions rather than two. Spencer (2000) proposed streamlining this process to two questions in order to accommodate the time constraints on teams that were working on large software projects. He concluded that this new version achieves the same goals while requiring less time [42].

1. Changing bases was not fully intuitive.
2. We thought the difference between the arithmetic and logical right shift operators (initially >> and >>> respectively) might be unclear to some users. These were the same operators used in Java and JavaScript, but we suspected that not all users would be familiar with them.
3. We thought some users might expect the shifts to be unary operators rather than binary operators. That is, a user might try to double click >> to shift right by two rather than entering an operand of two after the initial click.
4. The calculator catches user errors (like a divide by zero) and displays the error message. While it was clear how to exit the error mode (by clicking `AC` or `C`), the difference between the two ways was not clear.

We provide the full walkthrough report in Appendix J.

11.2. Thinking-Aloud Study

While the streamlined cognitive walkthrough produced valuable insights, it was limited in that we, the developers of the application, were the ones doing the evaluation. Because we designed the calculator, our prior knowledge of its interface may have masked additional usability issues. To address this concern, we conducted a thinking-aloud study involving potential users.[16] We presented five CS upperclassmen at Princeton with the same four tasks used in the walkthrough, and we instructed them to think aloud while completing the tasks, so we could observe their thought processes. While a participant completed the tasks, we answered yes-or-no questions about his or her behavior and recorded his or her thoughts. Following each task, we asked the participant if it was clear how to use the calculator to complete the task. At the end of the evaluation, we asked the participant to share any other comments on the interface and suggestions for the calculator. We refer the reader to Appendices K and L for the participant and evaluator versions of the tasks. Through this study, we confirmed the suspected learnability problems identified in the walkthrough.

[16] Nielsen (1993) suggested that thinking aloud is the "single most valuable usability engineering method," for it lets us understand how users view a system, and it provides "a very direct understanding of what parts of the dialogue cause the most problems" [33]. Nielsen (2012) provides a more detailed analysis of the benefits and disadvantages of a thinking-aloud study [35].

Respectively,

1. Users were confused when changing bases, and some hesitated before clicking the decimal box to switch to decimal for the first time. Users were particularly confused about switching to binary because unlike the hexadecimal and decimal boxes, the binary box lacked a label. At the end of the task, users said they thought the way of switching bases made sense and would be clear for future use despite the initial confusion.
2. Although each user used the correct shift operator for an arithmetic right shift (`>>`), only one actually knew the difference between the arithmetic right shift (`>>`) and logical right shift (`>>>`) operators. One user went off "intuition," two others guessed (with one stating, "This is where I go to Google"), and the last user did not notice that there were two right shift operators.
3. Only two out of the five users expected the right shift to be a binary operator. One user double clicked the operator to shift right by two and said, "That's not what I expected."
4. Four users used `C` to exit the error mode and undo the last operation, correctly explaining the difference between the `C` and `AC` operators. The user who used `AC` said he always clicks `AC`, and when prompted, did not know what the difference would be.

We also discovered that users had trouble switching bit lengths and between the signed and unsigned modes because they did not realize that the mode text at the top of the calculator was clickable. Users were also unfamiliar with the `ROL` operator. We include a full writeup of the the study in Appendix M.

11.3. Heuristic Evaluation

Following the streamlined cognitive walkthrough and thinking-aloud study, we conducted a heuristic evaluation with the heuristics (Appendix N) and methodology described in Nielsen (1994) [34].[17] In a heuristic evaluation, an evaluator (or team of evaluators) inspect(s) the user interface and compare(s) the elements to a list of "recognized usability principles" (the heuristics); any usability

[17] Due to their quick and low-cost nature, heuristic evaluations are one of the main forms of "discount usability engineering" [34].

problems are listed along with the heuristic(s) they violated. In our evaluation, we classified the usability problems from the cognitive walkthrough and thinking-aloud study in addition to new ones we discovered. In all, we identified 13 usability problems. These problems were mostly concerned with learnability, primarily violating the "consistency and standards" and "help and documentation" heuristics. For example, the "signed" and "unsigned" labels violated the "consistency and standards" heuristic because they did follow the platform conventions for buttons. This caused users to have trouble learning that they could change the calculator's mode by clicking the appropriate label. We provide a full list of usability problems, heuristic(s) violated, and possible solutions in Appendix O.

12. Version Two

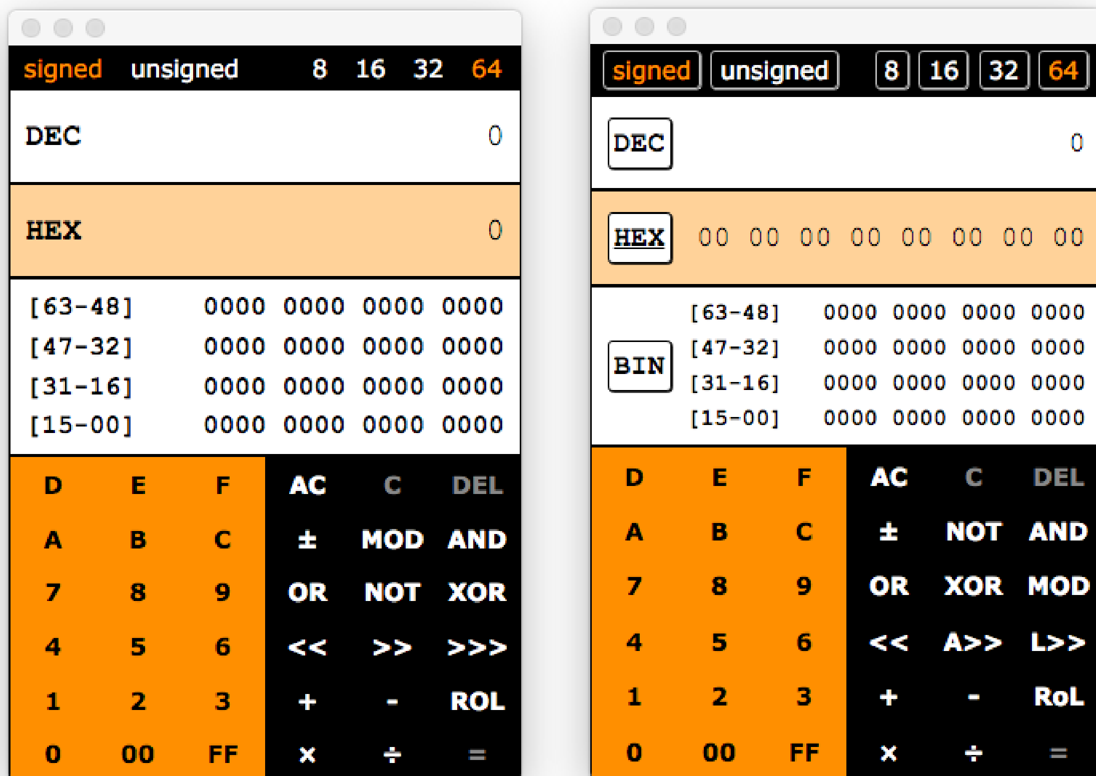


Figure 4: A side-by-side comparison of Version 1 (left) and Version 2 (right).

To address the learnability issues discovered in our usability evaluation, we created a second version of the calculator. In doing so, we focused on consistency both within the user interface and with

standard web-design practices. We present a side-by-side comparison of the two calculator versions in Figure 4. For Version 2, we made the following changes:

- We placed borders and shadows around the mode and base names, making them look more like buttons. We also made only the base name clickable rather than the entire display, further reinforcing the notion of base names as buttons.
- We added a `BIN` label, making the binary display consistent with the other ones.
- We underlined the active base's name. This makes it clearer that the orange display marks the active base rather than serving an aesthetic purpose.
- We moved `NOT` to be with the other unary operators, increasing consistency.
- We renamed the `ROL` operator to `RoL`, matching its naming on other calculators and making its abbreviation more reflective of its full name ("rotate left").
- We changed the cursor to use a standard arrow rather than a pointer when hovering over a

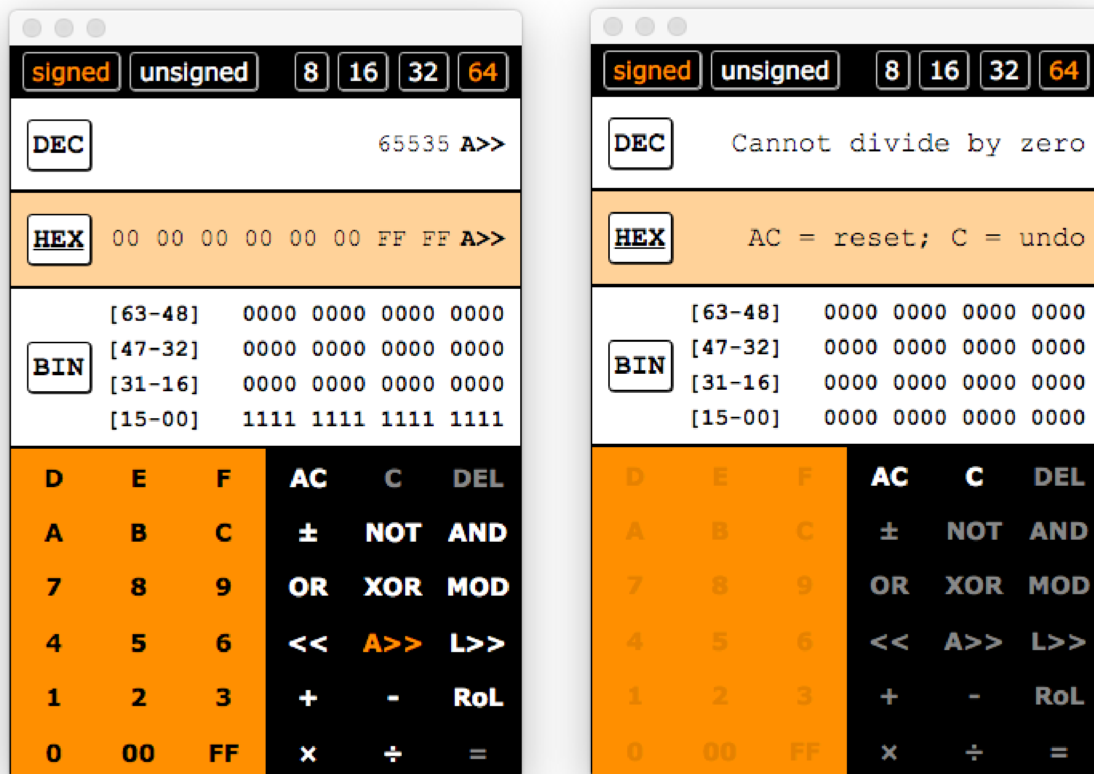


Figure 5: Version 2 in the middle of a binary operation (left) and after catching a user error (right). Figure 10 in Appendix Q shows a side-by-side comparison of Version 1 and Version 2 in these states.

disabled operator, such as `C` or `DEL` in Figure 4. This cursor better suggests that the operator is no longer clickable.

- We extended the keyboard shortcuts to cover all functionality and created a list of shortcuts (Appendix P) to be included in the listing on the Chrome Web Store.
- We zero-padded the hexadecimal display and split the number into its bytes. This better reflects the use of hexadecimal as an abbreviation for binary, and it makes the hexadecimal display consistent with the zero-padded binary one.
- As shown in Figure 5, we displayed the binary operator next to the accumulator when the calculator was in the middle of a binary operation. This emphasizes that the operator is binary and provides redundant signaling to the highlighted button.
- As shown in Figure 5, we used the hexadecimal field to show instructions after the calculator has caught a user error. This text clarifies the difference between the `AC` and `C` operators.

13. Final Evaluation

13.1. Comparison with Past Work

We compared our calculator's availability and functionality to 15 known alternatives—the Windows 8 calculator, the Mac OS X calculator, a pre-installed GNOME calculator on a Debian Linux machine, a web calculator, six Android calculators, and five iOS calculators (Appendices R and S). Our calculator is:

- The only calculator to run on multiple platforms and the only calculator to run on Chrome OS.
- The only free calculator with offline support for 8, 16, 32, and 64-bit signed and unsigned integral types.
- The only calculator to include both arithmetic and logical right shifts. This is significant because in the C programming language, the result of a right shift on a negative signed integral type is implementation-defined [38]. That is, implementations can use either an arithmetic or a logical right shift. We wanted to accommodate both of these options, so we included both

types of shifts.

Additionally, we noticed the following features that were offered by other calculators but not ours. We believe these to be valuable additions to a potential Version 3:

- Seven calculators support individual bit editing.
- Eight calculators include the octal base.
- Two calculators support floating-point numbers.

13.2. Usability Evaluation

After addressing all of the usability problems identified in our first round of evaluations, we wanted to reevaluate our application. Due to time constraints, we could not hold a second thinking-aloud study, so we conducted a modified heuristic evaluation—unable to identify any additional usability problems, we instead assessed how well the application conformed to each heuristic. The calculator performed strongly on the heuristics, satisfying all of them. In Appendix T, we include a full report with arguments for and against why the calculator satisfies each heuristic.

14. Future Work

We are ready to list the calculator in the Google Play and Apple App Stores, and intend to so so as our next step. Following this, future work on the calculator should focus on adding new functionality. We believe individual bit editing and floating point support should be of the highest priority. Moreover, we would be interested in seeing the results of a thinking-aloud study conducted on a mobile device, for we believe the smaller screen-size and use of a touch screen could yield new insights into the application's usability. Lastly, given the prominence of cross-platform development techniques and the unique challenges posed by cross-platform development, we think the opportunity exists for academic work on evaluating the usability of cross-platform applications; an area of interest could be developing new heuristics for this type of software.

15. Conclusion

In this paper, we described the development of the first cross-platform programmer's calculator, built using the novel approach of a Google Chrome Application and the Chrome Apps for Mobile Toolchain. We demonstrated how we used a requirement analysis, scenarios, mocks, and storyboards to develop an initial prototype. We explained how we adapted a BigInteger library to achieve support of signed and unsigned 8, 16, 32, and 64-bit integral types in JavaScript. We showed how we used the CSS3 Flexible Box layout mode, CSS3 media queries, and viewport units to develop an adaptive user interface for a variety of devices. We described how we used a streamlined cognitive walkthrough, thinking-aloud study, and a heuristic evaluation to identify the learnability problems that we addressed in a second version. In all, this led to a calculator whose functionality compares favorably to existing alternatives and whose user interface satisfies widely-accepted usability heuristics.

16. Acknowledgements

I would like to thank my advisor Dr. Dondero for his help and guidance throughout this project. He was an invaluable resource for topics ranging from human computer interaction and usability to number systems and C.

17. Honor Code

This paper represents my own work in accordance with University regulations. /s/ Brian Rosenfeld

References

[1] Bootstrap grid system. [Online]. Available: http://getbootstrap.com/css/#grid

[2] "Media queries w3c recommendation," June 2012. [Online]. Available: http://www.w3.org/TR/css3-mediaqueries/#media0

[3] 12kk, "Hex,dec,oct,bin(dev calc)," Android Application, December 2014. [Online]. Available: https://play.google.com/store/apps/details?id=yumekan.android.devcalc

[4] agrieve, "Update readme.md to say the project is not getting any new features," Code commit to the MobileChromeApps/mobile-chrome-apps GitHub repository, August 2015. Available: https://github.com/MobileChromeApps/mobile-chrome-apps/commit/3aaa46a94278c27521e375aba55f2c4999d8d89d

[5] Apple, "Calculator," Pre-installed on OS X computers, 2015.

[6] J. Bishop and N. Horspool, "Cross-platform development: Software that lasts," *Computer*, vol. 39, no. 10, pp. 26–35, Oct 2006.

[7] P. Blog. (2012, March) Phonegap, cordova, and what's in a name? Available: http://phonegap.com/2012/03/19/phonegap-cordova-and-what%E2%80%99s-in-a-name/
[8] BrownDwarf, "Programmers calculator," Android Application, December 2015. Available: https://play.google.com/store/apps/details?id=com.browndwarf.hexconverter
[9] Calcuseum, "Casio: Cm100." Available: http://www.calcuseum.com/poc_13622.html
[10] V. Chandler, "Hexzombie-programmer's calculator," iOS Application, April 2014.
[11] A. Charland and B. Leroux, "Mobile application development: web vs. native," *Communications of the ACM*, vol. 54, no. 5, pp. 49–53, 2011.
[12] C. Coyier. (2015, November) A complete guide to flexbox. Available: https://css-tricks.com/snippets/css/a-guide-to-flexbox/
[13] C. Coyier. (April, 2012) Viewport sized typography. Available: https://css-tricks.com/viewport-sized-typography/
[14] J. Eckert, "64 bit calculator," iOS Application, April 2012. Available: https://itunes.apple.com/us/app/64-bit-calculator/id320585695?mt=8
[15] *ECMA-262 6th Edition, The ECMAScript 2015 Language Specification*, 6th ed., Ecma International, Geneva, June 2015.
[16] GNOME, "Gnome calculator (gcalctool)," Included with GNOME desktop environment. Available: https://wiki.gnome.org/Apps/Calculator
[17] Google, "Google play store," search for "programmer's calculator". Available: https://play.google.com/store/search?q=programmer%27s%20calculator&c=apps&hl=en
[18] Google. Run chrome apps on mobile using apache cordova. Available: https://developer.chrome.com/apps/chrome_apps_on_mobile
[19] Google. Welcome to native client. Available: https://developer.chrome.com/native-client
[20] Google. What are chrome apps? Available: https://developer.chrome.com/apps/about_apps
[21] Google, "Calculator," Available through the Chrome Web Store, 2015 September.
[22] J. Graham-Cumming, "How i love my hp-16c," June 2006. Available: http://blog.jgc.org/2006/06/how-i-love-my-hp-16c.html
[23] H. Heitkötter, T. A. Majchrzak, and H. Kuchen, "Cross-platform model-driven development of mobile applications with md 2," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, 2013, pp. 526–533.
[24] D. Hicks, "Hp-16c," 2013. Available: http://www.hpmuseum.org/hp16.htm
[25] "Istqb glossary," Web Application, International Software Testing Qualifications Board, March 2015. Available: http://astqb.org/glossary/
[26] F. Ioannides, "Programmer calculator," Android Application, December 2015. Available: https://play.google.com/store/apps/details?id=fidias.ioannides
[27] D. Macos and A. Solymosi, "Scamo: Realisation of an oo-functional dsl for cross platform mobile applications development," *AIP Conference Proceedings*, vol. 1558, no. 1, pp. 327–331, 2013. Available: http://scitation.aip.org/content/aip/proceeding/aipcp/10.1063/1.4825490
[28] P. Marius, "Considerations regarding the cross-platform mobile application development process," 2013.
[29] Microsoft, "Calculator," Pre-installed on Windows 8 computers, 2015.
[30] miwachang, "Programmer's calculator calc-p," Android Application, October 2013. Available: https://play.google.com/store/apps/details?id=com.miwachang.progcalc
[31] J. Montgomery, "Sci:pro calculator," iOS Application, November 2014. Available: https://itunes.apple.com/us/app/sci-pro-calculator/id684978583?mt=8
[32] M. D. Network. Using css flexible boxes. Available: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout/Using_CSS_flexible_boxes
[33] J. Nielsen, *Usability Engineering*. Academic Press, 1993.
[34] J. Nielsen, *Usability Inspection Methods*. Wiley, 1994, ch. Heuristic Evaluation.
[35] J. Nielsen. (2012, January) Thinking aloud: The #1 usability tool. Available: https://www.nngroup.com/articles/thinking-aloud-the-1-usability-tool/
[36] C. Nuttall, "App stores are not the future, says google," July 2009. Available: http://blogs.ft.com/tech-blog/2009/07/app-stores-are-not-the-future-says-google/
[37] J. Ohrt and V. Turau, "Cross-platform development tools for smartphone applications," *Computer*, vol. 45, no. 9, pp. 72–79, Sept 2012.
[38] Oracle, "Sun studio 12: C user's guide," 2010. Available: https://docs.oracle.com/cd/E19205-01/819-5265/bjazt/index.html
[39] Penjee. Programmers 64 bit calculator. Available: http://calc.penjee.com/
[40] peterolson, "Biginteger.js," Public GitHub repository. Available: https://github.com/peterolson/BigInteger.js
[41] J. Preece, H. Sharp, and Y. Rogers, *Interaction Design-beyond human-computer interaction*. John Wiley & Sons, 2011.

[42] R. Spencer, "The streamlined cognitive walkthrough method, working around social constraints encountered in a software development company," in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM, 2000, pp. 353–359.

[43] R. Tamura, "Progcalc rpn programmer calculator," iOS Application, August 2009. Available: https://itunes.apple.com/us/app/progcalc-rpn-programmer-calculator/id294256032?mt=8

[44] L. Verou, "dpilove." Available: http://dpi.lv/

[45] T. Wang, "Calcpro - programmers' calculator," iOS Application, August 2013. Available: https://itunes.apple.com/us/app/calcpro-programmers-calculator/id684106142?mt=8&ign-mpt=uo%3D4

[46] M. T. Weber), "Programmer calculator," Android Application, November 2015. Available: https://play.google.com/store/apps/details?id=org.mosdev.progcalc&hl=en

[47] Wikibooks, "C programming/c reference/stdint.h — wikibooks, the free textbook project," 2012, [Online; accessed 29-December-2015]. Available: https://en.wikibooks.org/w/index.php?title=C_Programming/C_Reference/stdint.h&oldid=2365818

[48] Wikipedia, "Bootstrap (front-end framework) — wikipedia, the free encyclopedia," 2015, [Online; accessed 30-December-2015]. Available: https://en.wikipedia.org/w/index.php?title=Bootstrap_(front-end_framework)&oldid=696859146

Appendices

A. Scenarios

1. A COS 217 student is working on the buffer overrun assignment. He wants to calculate the hexadecimal offset needed for a jump. He takes the destination address and subtracts the address of the instruction following the jump. Because the destination address is earlier in memory, this calculation gives a negative jump offset and causes difficulties.
2. A COS 217 student is preparing for an exam, which contains a disproportionately high number of questions involving bit fiddling. While studying and checking over her answers, the student uses a programmer's calculator for performing bit operations.
3. An operating systems (COS 318) student is working on writing his bootloader. As part of this assignment, he uses magic numbers for various hexadecimal memory constants. He performs various operations with these numbers and, while debugging with GDB, uses the built-in Linux calculator to check the results.
4. A computer architecture student is studying virtual memory. She takes a 64-bit hexadecimal address and uses shifts and bit masks to find the page table, the virtual page number, and the page offset.

B. Requirements

Functional

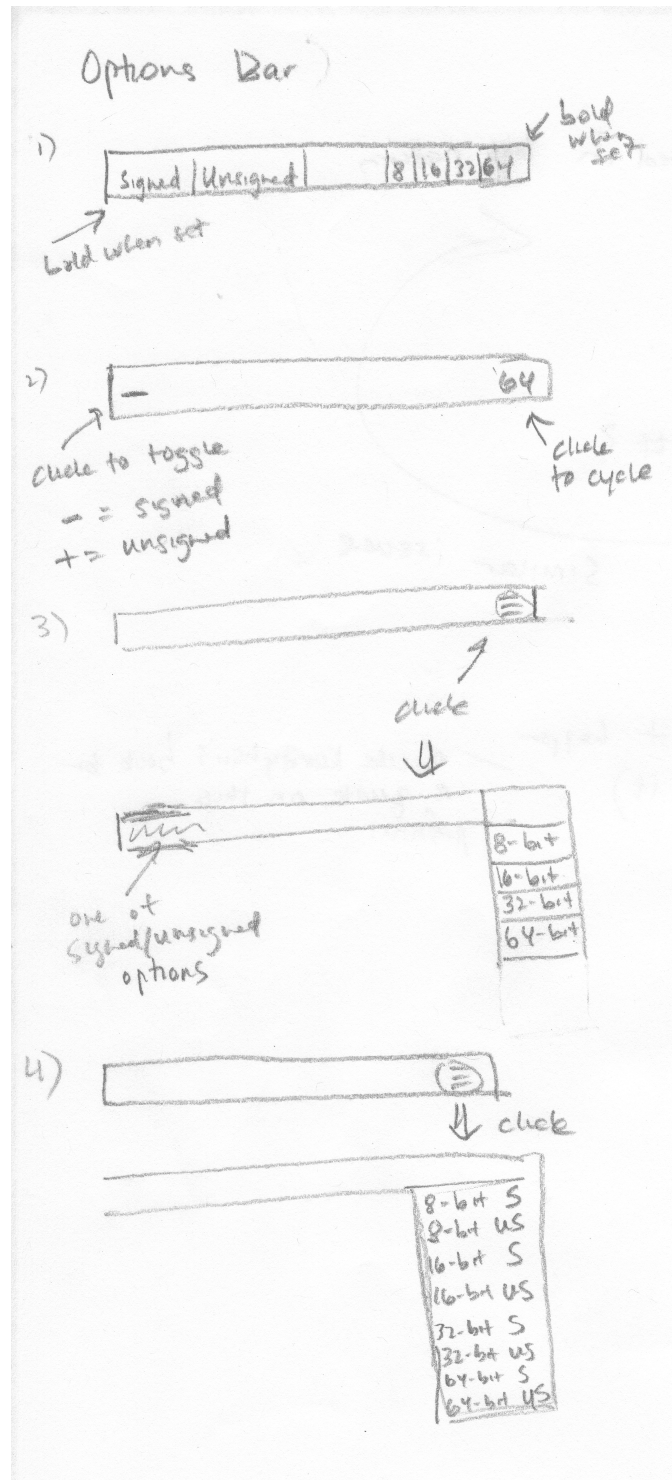
1. High Priority
   - Standard, non-programming functionality: add, subtract, times, divide
   - Bitwise operators: and, or, not, xor
   - Bit shifts: left and right shifts
   - Binary, decimal, and hexadecimal input and display
   - Clear, accumulator clear, and delete
   - 64-bit support
   - Signed and unsigned support
2. Medium Priority
   - 8, 16, and 32-bit support
3. Time Permitting
   - Circular shifts: RoL and RoR
   - Keyboard input and shortcuts
   - Individual bit editing
   - Back and undo buttons
4. Future Consideration
   - Floating point
   - ASCII, Unicode
   - 1s complement
   - Big endian, little endian representations
   - Byte flip, word flip

Non-Functional

1. Runnable as a Chrome App and on iOS and Android

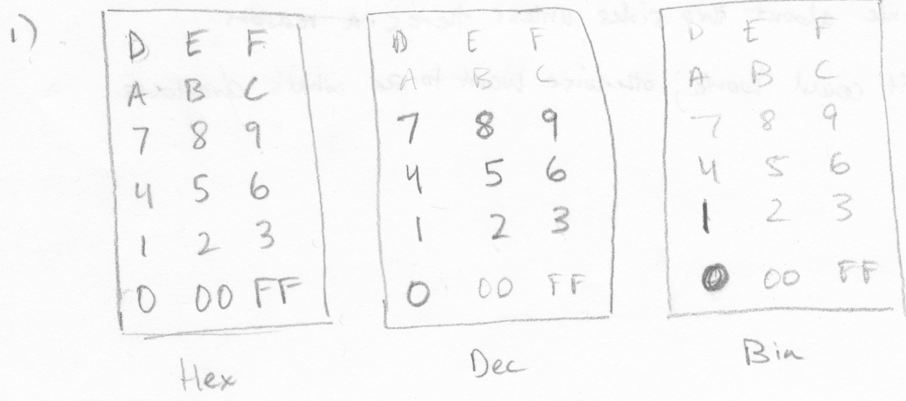
2. Installable through the various app stores
3. Perform as expected
4. Have an easy-to-use interface

C. Sketches

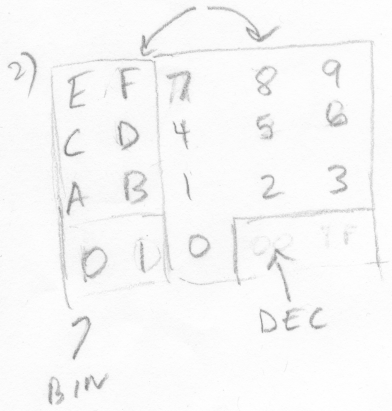


Keypad

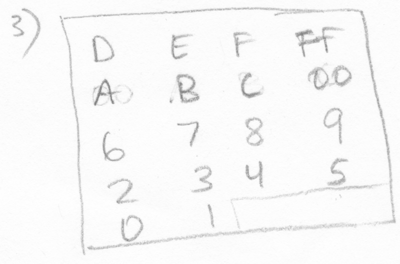
1)



2)

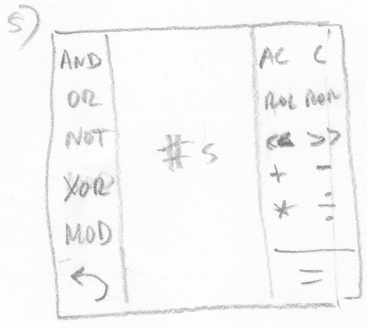


3)

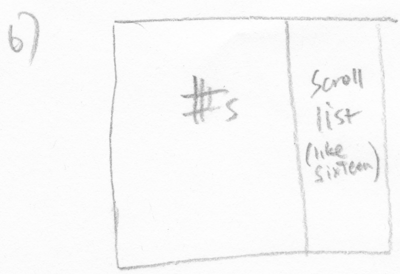


4)

5)

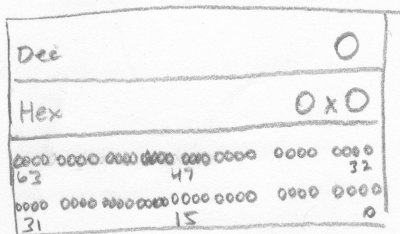


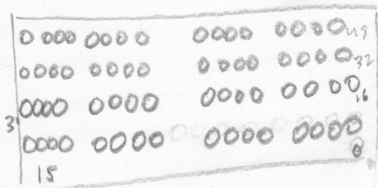
6)



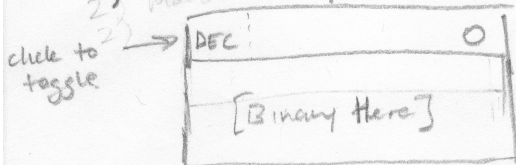
Displays

1) All three at once (like SixTeen)

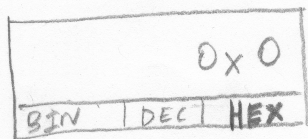




2) One of dec/hex w/ binary (Mac OSx)

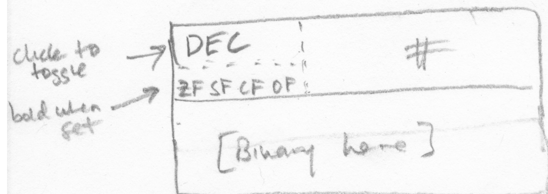


3) One of bin/dec/hex

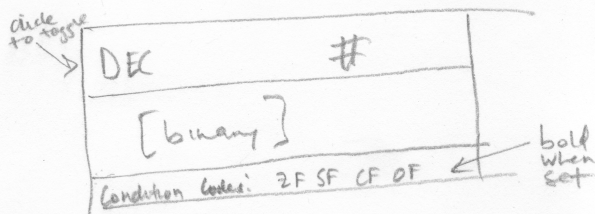


- takes up less space
- less expressive

4) w/ condition codes in display



w/ condition codes below display



D. Sample Storyboard

Figure 6: A storyboard for subtracting a larger hexadecimal number from a smaller one (as in the first scenario).

E. Initial Software Prototype

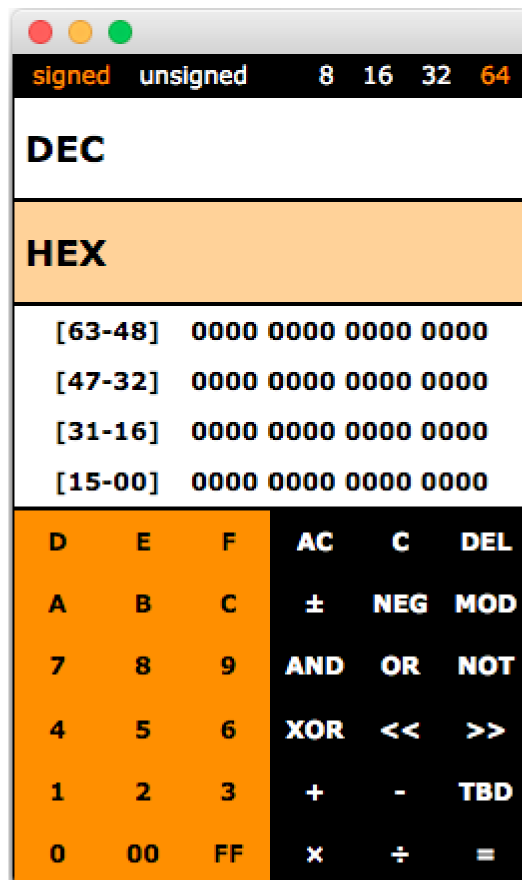


Figure 7: The first implementation of the user interface.

F. Distribution

Chrome Web Store

https://chrome.google.com/webstore/detail/programmers-calculator/pgkgdlpegifkoofoioopnbkkfhjociaj?hl=en-US&gl=US

World Wide Web

http://www.princeton.edu/~brianmr/calculator/[18]

[18] Aliased at http://brianrosenfeld.com/calculator/.

G. Version 1

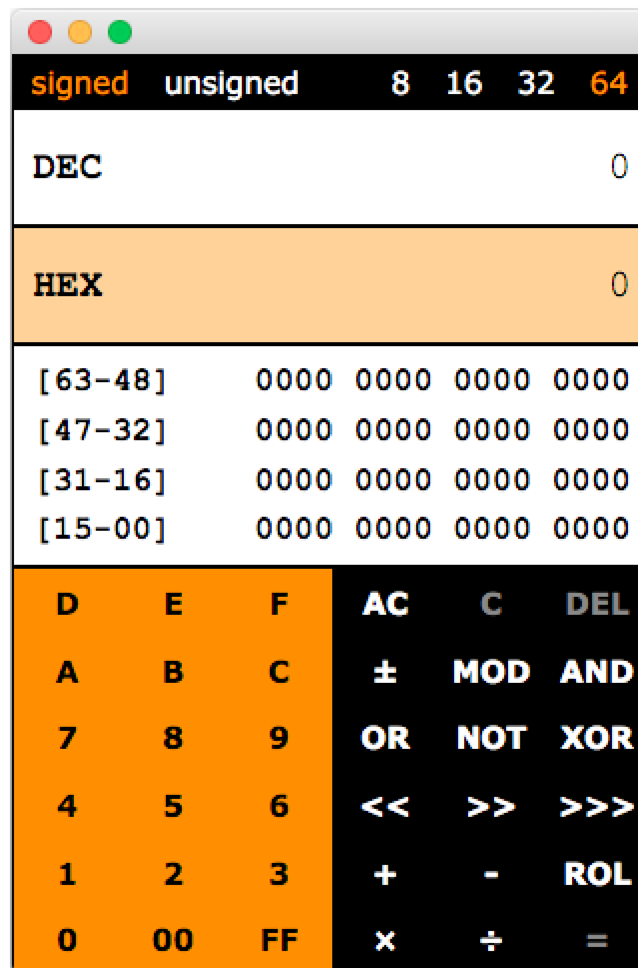


Figure 8: A screenshot of the Version 1 interface.

H. Responsive Layout

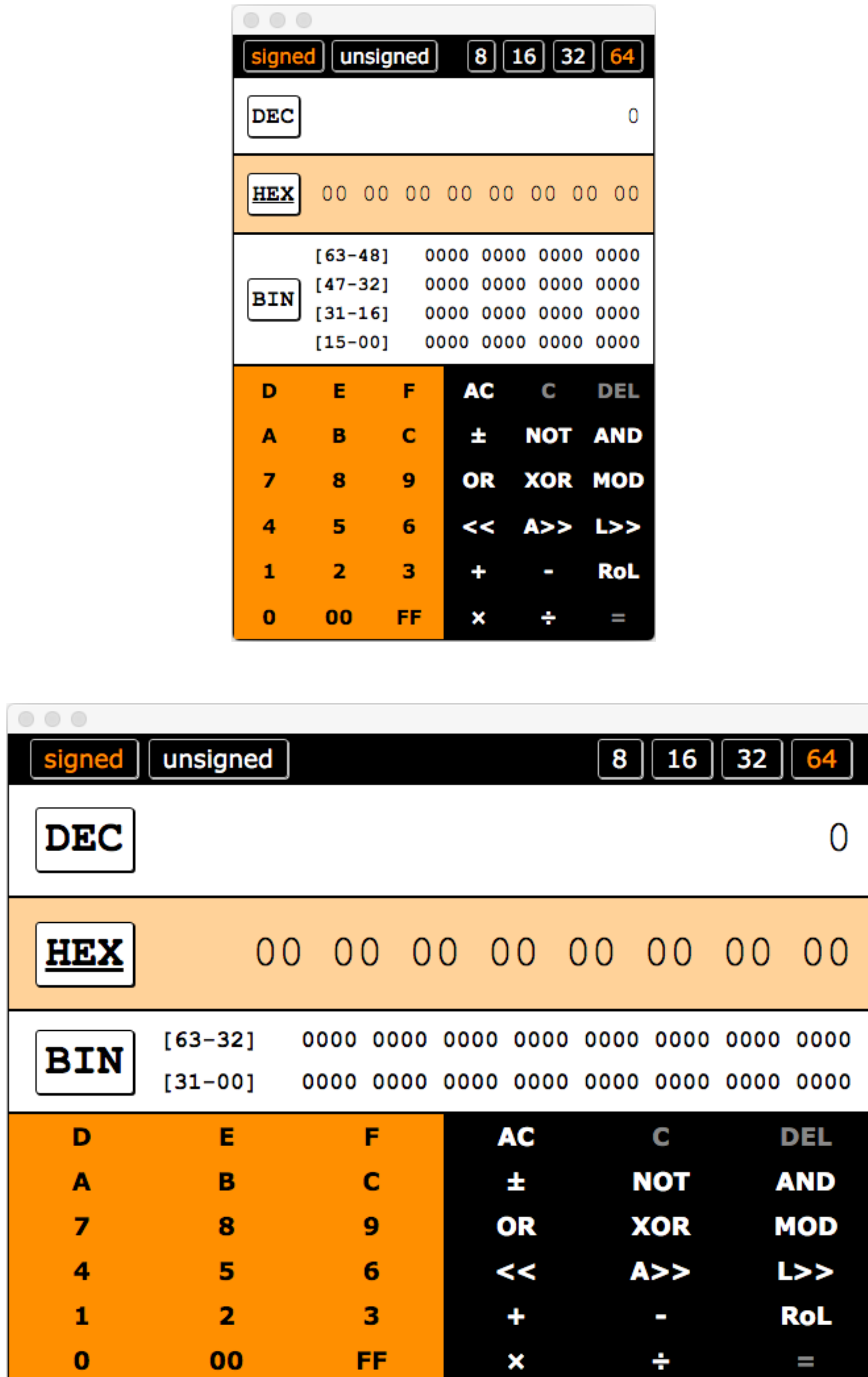


Figure 9: Screenshots of two different sizings of the Version 2 interface, scaled down by 50% to fit this page. In addition to the differences in text and button sizes, notice how the narrower window uses a four-row layout for the binary display and the wider-layout uses a two-row layout.

I. Supported Operators

1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Negation
6. Bitwise not
7. Bitwise and
8. Bitwise or
9. Bitwise xor
10. Modulo
11. Left shift
12. Arithmetic right shift
13. Logical right shift
14. Rotate-left circular shift
15. Clear
16. Accumulator Clear
17. Delete

J. Streamlined Cognitive Walkthrough

Task 1: Add the 32-bit, unsigned numbers $FF8A00_{16}$ and 500_{10} then subtract 101011_2. What is the result in hexadecimal?

1. Click the white “32” in the top bar to set the mode to 32-bits.
    a. Yes. 64 bits are shown in the binary display and the number 64 is orange, so the user should understand that the numbers in the top control the bit length. Moreover, when the user hovers over the word 64, the cursor changes to a pointer, suggesting that it is clickable.
    b. Yes. The number 32 has been changed to orange, the number 64 has been changed to white, and the 32 high-order bits have been grayed out. The user should understand that this means the calculator is in 32-bit mode.
2. Click the word “unsigned” in the top bar to change the mode to unsigned.
    a. Yes. The user just clicked a number in the top bar to change the bit length, so he or she should understand that the top bar is used for changing the mode. Moreover, the word “signed” is currently the same orange as 32-bits, so the user should understand that the calculator is currently in signed mode.
    b. Yes. The word “unsigned” is now highlighted, suggesting that the calculator is in unsigned mode.
3. Click FF then 8 then A then 00.
    a. Yes. The display for hex is currently set to a different color, and the entire keyboard is enabled. Moreover, the cursor turns to a pointer over the different keyboard options, suggesting that they are buttons.
    b. Yes. Clicking a button updates the display to show that value and the button temporarily changes color to show that the click was registered.
4. Click the + button.
    a. Yes. + is the standard symbol for addition.
    b. Yes. The + button temporarily changes color to show that the click was registered, and then the text of the button changes to orange, suggesting that the operation is active.
5. Click the box that has DEC in it.
    a. Most likely. The cursor changes to a pointer over the various display boxes and the HEX box is currently highlighted, so the user should understand that it is active.
    b. Yes. The DEC box is now highlighted instead of the HEX box and the non-decimal digits are grayed out. One suggestion would be to have the cursor no longer be a pointer over these disabled buttons.
6. Click 5 then 0 then 0.

   a. Yes. This is the standard way of entering a number on a calculator.
   b. Yes. The display updates after each digit is entered and each button temporarily changes color when clicked. Moreover, the display shows the plus sign and the number on a separate line, making it clear that the user is in the middle of an addition.
7. Click the – button
   a. Yes. – is the standard symbol for subtraction.
   b. Yes. The – button temporarily changes color to show that it was clicked, the display updates with the result of the addition, the – button becomes highlighted, and the + button is no longer highlighted—all of which suggest that the calculator is now in the middle of a subtraction.
8. Click the box with the binary numbers.
   a. Yes. The user has already changed bases, and this change requires the same type of action as before.
   b. Yes, the box with the binary digits is now highlighted instead of the DEC box and the non-binary digits are grayed out. As with before, an improvement would be to have the cursor no longer be a pointer over these disabled buttons.
9. Enter 1 then 0 then 1 then 0 then 1 then 1.
   a. Yes. The user has already entered numbers on the calculator, and this is standard calculator procedure. Moreover, the display shows the subtraction sign and the number on a separate line, making it clear that the user is in the middle of a subtraction.
   b. Yes. Each of the displays is updated with the new operand.
10. Press =
   a. Yes. This is standard calculator procedure.
   b. Yes. The displays are updated to show one value and no longer show the subtraction sign, and the subtraction sign is no longer highlighted, suggesting that the operation has been completed.
11. Read the number FF8BC9 from the HEX box.
   a. Yes. This box is labeled HEX, so the user should understand it contains a hexadecimal value.
   b. Yes. The user will have their final value.

Task 2: Shift the 32-bit, signed integer -1 right 2 with an arithmetic shift (a shift in which the sign bit is shifted from the left).

1. Press AC to clear the calculator.
    a. Yes. AC is standard for accumulator clear on calculators.
    b. Yes. The displays now show zero.
2. Click the “signed” text.
    a. Yes. The user already knows how to do this from the previous task.
    b. Yes. The “signed” text is now highlighted instead of “unsigned”.
3. Click the button 1.
    a. Yes. This is standard procedure for entering numbers.
    b. Yes. The number 1 is now displayed.
4. Click the button with the plus-minus sign.
    a. Yes. Because the button has a plus-minus sign, the user should understand that it is used for changing sign.
    b. Yes. The display now shows -1 for decimal and the appropriate hex and binary representations.
5. Click the DEC box (see previous task)
6. Click the >> button.
    a. Maybe. >> is clearly a right shift, but so is >>>. In Java and JavaScript, >> is defined as an arithmetic right shift and >>> is defined as a logical right shift. If the user is familiar with bitwise operators in these languages, then the difference between the calculator’s right shifts will be clear. If the user is programming in C, the meaning of these operators may be unclear, for C only has one right shift operator (>>) and that operator has undefined behavior on negative numbers. An alternative could be A>> and L>> to express that these are arithmetic and logical right shifts respectively.
    b. Yes. The >> button temporarily changes color to show that the click was registered, and then the text of the button changes to orange, suggesting that the operation is active.
7. Click the button 2.
    a. If it’s clear that >> is a binary operator rather than a unary operator, then yes. Otherwise, the user may try to click >> twice in order to shift by two bits. Highlighting the operator shows that the operation is in progress, so the user should understand that this is a binary operator.
    b. Yes. See step 3.
8. Press = (see previous task)

Task 3: In the following code snippet, what is y equal to in decimal?

```
uint16_t x = 27023;
int8_t y = x;
```

1. Press AC to clear the calculator (see previous tasks)
2. Click “unsigned” (see previous tasks)
3. Click “16” (see previous tasks)
4. Click DEC (see previous tasks)
5. Enter 2 then 7 then 0 then 2 then 3 (see previous tasks)
6. Click “8” in the top black bar (see previous tasks)
7. Click “signed” (see previous tasks)

Task 4: Try to divide 10_{16} by zero. Now exit the error mode and divide by two instead.

1. Click 1 then 0 (see previous tasks)
2. Click ÷ (see previous tasks)
3. Click 0 (see previous tasks)
4. Click = (see previous tasks
5. Click C
    a. It’s not very clear what the difference is between AC and C though after the first time, users would know the difference. An option would be to use the HEX box to show instructions.
    b. Yes, the error is gone and the divide has been undone with the divide operator still active.
6. Click 2 (see previous tasks)
7. Click 0 (see previous tasks)

Takeaways

- Use a regular cursor for disabled buttons
- Possibly use A>> and L>> instead of >> and >>>.
- Use the HEX box to give instructions during an error.

K. Thinking-Aloud Study: Participant Instructions

Please complete the following tasks while thinking out loud.

1. Add the 32-bit, unsigned numbers $FF8A00_{16}$ and 500_{10} then subtract 101011_2. What is the result in hexadecimal?

2. Shift the 32-bit, signed integer -1 right 2 with an arithmetic shift (a shift in which the sign bit is shifted from the left).

3. In the following code snippet, what is y equal to in decimal?

```
uint16_t x = 27023;
int8_t y = x;
```

4. Try to divide 10_{16} by zero. Now exit the error mode and divide by two instead.

L. Thinking-Aloud Study: Evaluator Instructions

1. Add the 32-bit, unsigned numbers $FF8A00_{16}$ and 500_{10} then subtract 101011_2. What is the result in hexadecimal?
    a. Does the user change to unsigned, 32-bit mode correctly? Yes / No
    b. Does the user use FF and/or 00 to input the hex number? Yes / No
    c. Can the user change to decimal? Yes / No
    d. Does the user change to binary? Yes / No
    e. Does the user hit equals before the subtraction key? Yes / No
    f. Does the user report the correct answer (FF8BC9)? Yes / No
    g. Other observations and user's thoughts

    h. Was it clear?

2. Shift the 32-bit, signed integer -1 right 2 with an arithmetic shift (a shift in which the sign bit is shifted from the left).
    a. Does the user know which right shift to use? Yes / No
    b. Is it clear that the right shift is a binary operator? Yes / No
    c. Other observations and user's thoughts:

3. In the following code snippet, what is y equal to in decimal?

```
uint16_t x = 27023;
int8_t y = x;
```

    a. Does the user set the correct mode initially? Yes / No
    b. Does the user cast correctly (expected answer is -113)? Yes / No
    c. Other observations and user's thoughts
    d. Was it clear?

4. Try to divide 10_{16} by zero. Now exit the error mode and divide by two instead.
   a. How did the user exit the error mode? AC / C
   b. Other observations and user's thoughts:

Debriefing

Read: These tasks were designed to expose you to various parts of the calculator's interfaced and to observe how you interacted with it. Aside from what we have discussed, is there anything that was unclear? Do you have any further suggestions for either the interface or functionality?

M. Thinking-Aloud Study: Results

- **Participants**: three seniors and two juniors in the Computer Science department at Princeton.
- **Program**: Calculator version 0.0.1, which was downloaded from the Chrome Web Store and run on my MacBook Pro. The calculator was run at the default size.

Task 1: Add the 32-bit, unsigned numbers $FF8A00_{16}$ and 500_{10} then subtract 101011_{2}. What is the result in hexadecimal?

- None of the five users set the initial mode correctly.
  - Because users worked with modes correctly in future tasks, I believe this does not reflect a usability issue and instead resulted from users viewing "32-bit, unsigned" as merely a detail in the task. I did not ask why users did not change the mode, for I did not want to influence them in future tasks (notably, task three that focused specifically on modes).
  - One user explicitly stated that the result would be the same for 64-bits numbers, so he wasn't going to change the bit length. This property of the task represents a shortcoming in the task itself and should be changed if the task is to be used again in the future.
- Only one user uses the FF and/or 00 shortcuts, and three users used the keyboard.
- There was some confusion when changing between hexadecimal, decimal, and binary inputs.
  - Two users started to input 500 as a hexadecimal number before clearing it after realizing that it was not decimal as requested.
  - The first time that users tried to change the base, there was some hesitation before clicking the decimal box.
  - Three users were confused about switching to binary because the binary box has a different display than the decimal and hex boxes and lacked a BIN label.
    - One user was going to convert the number to hex.
  - When entering a binary number, one user thought the bits were buttons.
- At the end of the task, users said they thought the way of switching bases made sense and would be clear for future use (despite confusion on the first use).

Task 2: Shift the 32-bit, signed integer -1 right 2 with an arithmetic shift (a shift in which the sign bit is shifted from the left).

- Every user used the correct right shift (>>), however only one actually knew the difference between (>>) and (>>>).
  - One went based off of "intuition", explaining that he was familiar with the two different operations.
  - Two guessed with one of them stating, "This is where I go to Google" and explaining that he "never keeps [the difference] in mind."
  - One didn't notice (>>>), so he used (>>).
- Only two out of the five users expected the right shift to be a binary operator.
  - One user double clicked the operator to shift right by two and said, "That's not what I expected." When prompted, he explained that he thought (>>) was a right shift by one and (>>>) was a shortcut to right shift by two.
- Two users entered -1 as "-" then "1" rather than by using the negation button.
  - One of these users used the keyboard, which maps the underscore key to subtraction rather than negation. This user suggested providing a list of keyboard shortcuts with hovering over an operation showing a description of the operation and its keyboard shortcut.

Task 3: In the following code snippet, what is y equal to in decimal?

```
uint16_t x = 27023;
int8_t y = x;
```

- Three users performed this task without any issues.
  - One explained that she hadn't initially seen the signed/unsigned labels but noticed it later.
  - One user was still in binary from the previous task, and when he entered "2" he said that he expected an automatic change to decimal.[i] He asked for keyboard shortcuts for changing base and explained that he "wants speed".
- One user who performed the task incorrectly by not setting the initial mode said it wasn't clear at first that the text labels were modes and clickable.
- One user brought up the fact C has specific promotion rules and that the changes were atomic.[ii] He suggested using the mode buttons 8s 8u … 64s 64u to insure the correct casting.

Task 4: Try to divide 1016 by zero. Now exit the error mode and divide by two instead.

- Four users used C and correctly explained the difference in behavior between AC and C.
    - One of the users had initially tried to exit the error mode by pressing delete on the keyboard or by typing.
    - Even so, one of these users thought it was unclear out of normal context.
- One user used AC instead of C. He said he always clicks "AC", and when prompted, he didn't know what the difference would be.

Other user thoughts and comments

- One user said he currently uses Google for these types of tasks but that he thought the calculator would be helpful.
- One user explained that he tried to use the keyboard because he normally works with Emacs. Another user said he normally uses the numerical keypad when he is at a workstation.
- One user commented that he likes that unusable keys became grey.
- One user commented that it was easy to switch between decimal, hexadecimal, and binary, which is normally a pain point for him. Another user suggested that there be buttons for switching the base rather than clicking in the box.
- Three users asked what ROL was.[iii]
- One user suggested that on startup it be made clearer which box you're typing into (which base you're using). He thought the different color for hex represented an aesthetic difference rather than a functional one.
- One user said that he wants to see the calculation how it is in his head. For example, for the first task he would want to see $FF8A00_{16} + 500_{10}$ in a box above DEC.

Usability problems

1. It is unclear to some users that the mode text labels are clickable.
2. Switching bases is confusing for first time users. It is not intuitive that the boxes are clickable. Moreover, the binary box is not consistent with the other days, making it confusing to switch to binary, and the initial selection of the hex box looks like an aesthetic difference rather than a functional one.
3. The difference between >> and >>> is unclear.
4. It is not clear that shifts are binary operators.
5. It is unclear to some users what the difference between AC and C are in the error mode.
6. There is no documentation for keyboard shortcuts.
7. It is not possible to change bases from the keyboard.
8. Users do not know what ROL is.

Other user suggestions:

- Using the mode buttons 8s 8u … 64s 64u to insure the correct casting. This is less aesthetically pleasing, but helps users who aren't familiar with the rules for C promotions.
- A separate display that shows the operation as the user entered it, for example it could show $FF8A00_{16} + 500_{10}$.

Notes

[i] This is not possible because it could be a hexadecimal number.

[ii] We realized that were the task as follows, changing to signed before changing the bit length would produce -1 rather than the correct result of 255 that is achieved when changing the bit length first.

```
uint8_t x = 255;
int16_t y = x;
```

[iii] ROL is rotate left.

N. Nielsen (1994) Heuristics [34]

1. **Visibility of system status:** The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.
2. **Match between system and the real world:** The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.
3. **User control and freedom:** Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.
4. **Consistency and standards:** Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.
5. **Error prevention:** Even better than good error messages is a careful design which prevents a problem from occurring in the first place.
6. **Recognition rather than recall:** Make objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.
7. **Flexibility and efficiency of use:** Accelerators—unseen by the novice user—may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.
8. **Aesthetic and minimalist design:** Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.
9. **Help users recognize, diagnose, and recover from errors:** Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.
10. **Help and documentation:** Even though it is better if the system can be used without documen-

tation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

O. Heuristic Evaluation

Problem	Heuristic(s) violated	Potential solution(s)
1. It is unclear to some users that the mode text labels are buttons.	• Consistency and standards • Help and documentation	• Make these look more like buttons • Tooltip instructions
2. It is unclear that the display boxes are clickable for changing bases.	• Consistency and standards • Help and documentation	• Make the labels look like buttons • Tooltip instructions • Separate buttons for changing base
3. The binary box lacks a label, making it different than the other boxes and harder to realize that it is clickable.	• Consistency and standards	• Add a BIN label
4. It is unclear that hex is initially selected rather than just a different color.	• Visibility of system status	• Bold the label of the selected box
5. The difference between the right shifts is unclear.	• Help and documentation	• Rename to A>> and L>> • Tooltip instruction.
6. It is unclear that shifts are binary operators.	• Help and documentation	• Tooltip instructions
7. The difference between AC and C in the error mode is unclear to some users.	• Help users recognize, diagnose, and recover from errors	• Provide instructions in the HEX box. • Tooltip instructions.
8. There is no documentation for keyboard shortcuts	• Help and documentation	• Tooltip with keyboard shortcut. • Provide separate list of shortcuts.
9. It is not possible to change base from the keyboard.	• Flexibility and efficiency of use	• Add a shortcut
10. Users do not know what ROL stands for.	• Help and documentation	• Provide a tooltip instruction
11. NOT is mixed in among the binary operators.	• Consistency and standards	• Move NOT to before the binary operators.
12. Hex is not consistent with binary because it is not zero padded on the left.	• Match between system and the real world.	• Padded hex on the left with zeros.
13. Disabled buttons still have a pointer for a cursor.	• Consistency and standards	• Use a normal pointer.

Note: The heuristic evaluation was conducted after the streamlined cognitive walk through and thinking-aloud study, so many of the usability problems are cross-listed. The list here can be considered a "master" list.

P. Keyboard Shortcuts

Operation	Shortcut
signed	Shift+S
unsigned	Shift+US
8-bit	Shift+1
16-bit	Shift+2
32-bit	Shift+4
64-bit	Shift+8
binary	Shift+B
decimal	Shift+D
hex	Shift+H
[0-F]	[0-F]
00	Shift+0
FF	Shift+F
AC	esc or Shift+A
C	Shift+C
DEL	delete
±	Shift+- or p
NOT	Shift+` (~) or n
AND	Shift+7 (&)
OR	|
XOR	Shift+6 (^)
MOD	Shift+5 (%) or m
<<	<
A>>	>

L>>	Shift+> or l
+	Shift+= or keypad plus
–	- or keypad minus
×	Shift+8 or x or keypad multiply
÷	/ or keypad divide
ROL	r
=	= or enter/return

Q. Version 1 vs. Version 2

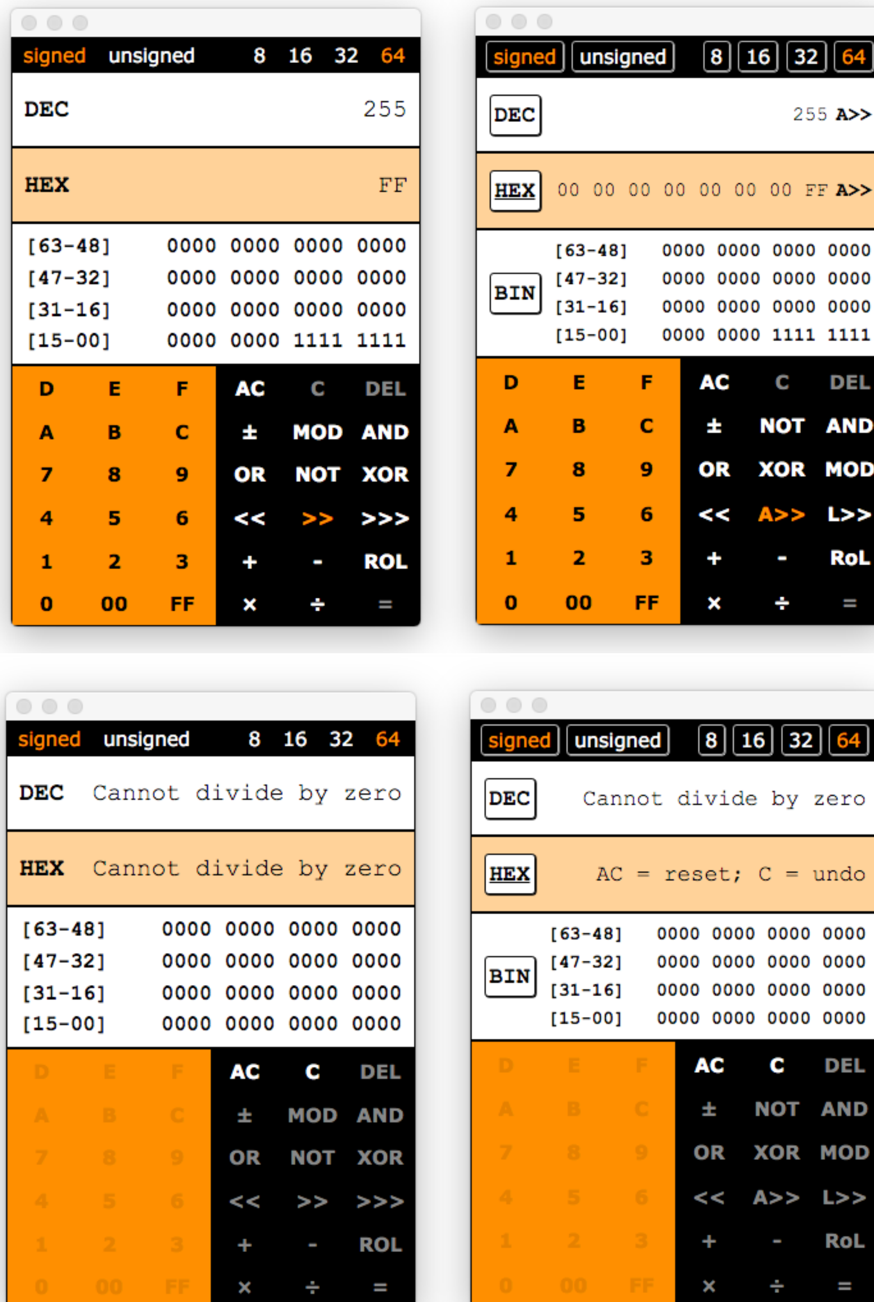


Figure 10: The first row shows Version 1 (left) and Version 2 (right) in the middle of a binary operation. Notice how Version 2 makes the calculator's status clearer by showing the operator next to the accumulator. The second row shows Version 1 (left) and Version 2 (right) after a divide by zero error. Notice how Version 2 provides instructions explaining the two different ways to exit the error mode.

R. Availability Comparison

Calculator	Platform(s)	Offline	Free
Our calculator	Chrome, OS X, Windows, Linux, Android, iOS, Web	✓	✓
Windows [29]	Windows	✓	✓
OS X [5]	OS X	✓	✓
GNOME [16]	GNOME / Linux	✓	✓
Penjee [39]	Web[a]		✓
DevCalc [3]	Android	✓	✓
CALC-P [30]	Android	✓	✓
Mosdev[b] [46]	Android		✓
SixTeen [8]	Android	✓	✓
Ioannides [26]	Android	✓	✓
CalcPro [45]	iOS	✓	✓
HexZombie [10]	iOS	✓	✓
Sci:Pro [31]	iOS	✓	✓
ProgCalc [43]	iOS	✓	✓
64 BitCalc[c] [14]	iOS	✓	

[a] The application does not render well on mobile, and it does not use a responsive layout.

[b] The application locks the calculator when the device is not connected to the internet, explaining that the app is funded by ads and requires an internet connection.

[c] Costs $2.99.

S. Functionality Comparison

	Modes						Input		Right Shifts	
Calculator	8	16	32	64	128	FP[a]	Bases	Bit Editing	Arith.	Log.
Our calculator	S/US	S/US	S/US	S/US			2,10,16		✓	✓
Windows	S	S	S	S			2,8,10,16	✓		✓
OS X				US			8,10,16	✓		✓
GNOME				US			2,8,10,16	✓[b]		✓
Penjee	S/US	S/US	S/US	S/US	S/US	✓	2, 10, 16	✓		✓
DevCalc				S			2,8,10,16			
CALC-P			S/US				2,10,16			✓
HexCalc	S/US	S/US	S/US				2,8,10,16			✓
Mosdev	S/US	S/US	S/US	US			2,8,10,16	✓	✓	
SixTeen	S/US	S/US	S/US	US		✓	2,10,16	✓	✓	
Ioannides	S	S	S	S			2,8,10,16		✓	
CalcPro	S	S	S				2,10,16			
HexZombie[c]			S/US	S/US[d]			10,16			✓
Sci:Pro				US			2,8,10,16			✓
ProgCalc				S/US			8,10,16[e]		✓	
64 Bit Calc	S/US	S/US	S/US	S/US			2,8,10,16	✓	✓	

[a] Floating Point.
[b] Bit editing only works for the first 48 bits.
[c] Does not show the binary representation.
[d] 64-bit support is only on the iPad.
[e] The calculator shows the binary representation, but there is no way to enter a binary number.

T. Heuristic Assessment

We assed the application against each of Nielsen's (1994) ten heuristics [34]. We believe the calculator satisfies all ten. Below, we present the arguments for and against each one. Negative factors are emphasized in italics.

- **Visibility of system status**
  - The system highlights the active signed/unsigned status, bit length, and base.
  - Throughout the system, buttons change color when clicked, recognizing the user's action.
  - System displays update promptly to user action.
  - When an operation is in progress, the system highlights the binary operator and displays it next to the accumulator.
  - When in an error mode, the system provides a message explaining the error.
- **Match between system and the real world**
  - All operator symbols match real world conventions except `A>>` and `L>>`. We chose those two operators because >> and >>> (the Java and JavaScript operators) were unclear to users.
  - The calculator uses infix notation, which is how users think.
  - The calculator offers common, real world bit lengths and bases.
- **User control and freedom**
  - The calculator provides `AC`, `C`, `DEL` for varying degrees of clearing or deletion.
  - The calculator allows users to switch binary operators by clicking a new operator before the operand is entered.
  - *The calculator does not feature redo.*[19]
- **Consistency and standards**
  - The interface groups similar operators together.
  - The interface follows platform conventions for the cursor.
  - The interface's clickable text looks like buttons.
- **Error prevention**

[19] We did not consider this to be needed.

- The calculator disables digits that are not valid for the base.
- The calculator disables operators that are not valid for the signed/unsigned mode.
- The calculator bounds checks user input, only allowing the user to enter valid numbers.

- **Recognition rather than recall**
  - The user interface remains relatively constant throughout use. The only changes include activating and disabling buttons and updating the display.
  - The Chrome Web Store listing links to keyboard shortcuts.
- **Flexibility and efficient of use**
  - The calculator features 00 and FF for quicker input.
  - The calculator offers keyboard shortcuts for use on computers.
- **Aesthetic and minimalist design**
  - The interface contains only the information that is needed.
  - *Version 1 was more "minimalist" than Version 2.*[20]
- **Help users recognize, diagnose, and recover from errors**
  - The calculator's error messages use plain language.
  - The calculator provides instructions for resolving user errors.
- **Help and documentation**
  - The calculator does not need documentation to be used.
  - *The calculator does not include documentation.*[21]

[20] We added the additional visual cues to ease learnability—a worthwhile tradeoff.

[21] As noted above, the calculator does not need documentation to be used. We believe that a feature that requires documentation really just needs to be made more intuitive. If enough users request documentation, we will happily prepare and provide it.