Bifröst: Visualizing and Checking Behavior of Embedded Systems across Hardware and Software

Will McGrath[1,2], Daniel Drew[1], Jeremy Warner[1], Majeed Kazemitabaar[1,3], Mitchell Karchemsky[1], David Mellis[1], Bjoern Hartmann[1]

[1]UC Berkeley
EECS Department
{ddrew73,jwrnr,mkarch, mellis,bjoern}@berkeley.edu

[2]Stanford University
Computer Science Department
wmcgrath@stanford.edu

[3]University of Maryland
Computer Science Department
majeed@cs.umd.edu

ABSTRACT
The Maker movement has encouraged more people to start working with electronics and embedded processors. A key challenge in developing and debugging custom embedded systems is understanding their behavior, particularly at the boundary between hardware and software. Existing tools such as step debuggers and logic analyzers only focus on software or hardware, respectively. This paper presents a new development environment designed to illuminate the boundary between embedded code and circuits. Bifröst automatically instruments and captures the progress of the user's code, variable values, and the electrical and bus activity occurring at the interface between the processor and the circuit it operates in. This data is displayed in a linked visualization that allows navigation through time and program execution, enabling comparisons between variables in code and signals in circuits. Automatic checks can detect low-level hardware configuration and protocol issues, while user-authored checks can test particular application semantics. In an exploratory study with ten participants, we investigated how Bifröst influences debugging workflows.

ACM Classification Keywords
H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

Author Keywords
embedded systems; debugging; IDE; visualization

INTRODUCTION
User-friendly embedded system prototyping tools such as the Arduino platform [23] have exposed new classes of users to programming and electronics as part of the Maker movement. Many of these projects take the form of interactive devices which make use of sensors, actuators, and an embedded processor to measure, control, and respond to events in the physical world. These projects inherently span both hardware and software. Accordingly, they often require a diverse set of skills and tools for their successful development.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

UIST 2017 October 22–25, 2017, Quebec City, QC, Canada

© 2017 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-4981-9/17/10.
DOI: https://doi.org/10.1145/3126594.3126658

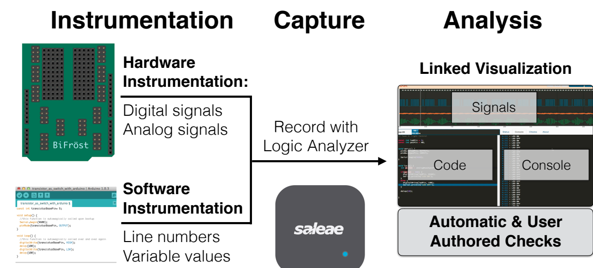

Figure 1. A high-level overview of the process that Bifröst uses to record and visualize execution traces of a user's embedded system.

Despite growing interest in these projects across education and experience levels, embedded interactive systems remain especially hard to understand and debug, for several reasons:

Lack of Visibility Compared to regular software, their operating state remains opaque and hidden from the developer, forming an *information barrier* [19] to understanding.

Difficulty of Fault Localization Faults occur in many locations, and determining whether they are located in software, hardware, or at the intersection of the two is problematic [4].

Real-time Requirements Systems that respond in real-time to user input or sensor data cannot easily be stopped for breakpoint debugging without disrupting their functioning.

A persistent challenge of developing embedded systems is that key information, such as progress through a user's program, when electrical signals are read or written, and interactions with peripheral devices are inherently invisible. State only has a noticeable impact on the outside world when a developer has proactively chosen to write to and monitor console logs or through actuators such as LEDs, speakers, or motors.

A suite of specialized inspection and debugging tools exist to make parts of embedded systems more visible. Multimeters and oscilloscopes display analog data; logic analyzers show digital signals; serial terminals and breakpoint debuggers show program data. However, these tools cannot readily interface with one another, and as a result, users cannot easily correlate data collected with different tools. This incompatibility makes it difficult to interpret behavior that spans from the electrical domain to the software running on the microcontroller.

We believe that giving developers additional insight into how hardware and software interact in a richly-linked environment with a unified interface will enable more efficient debugging of

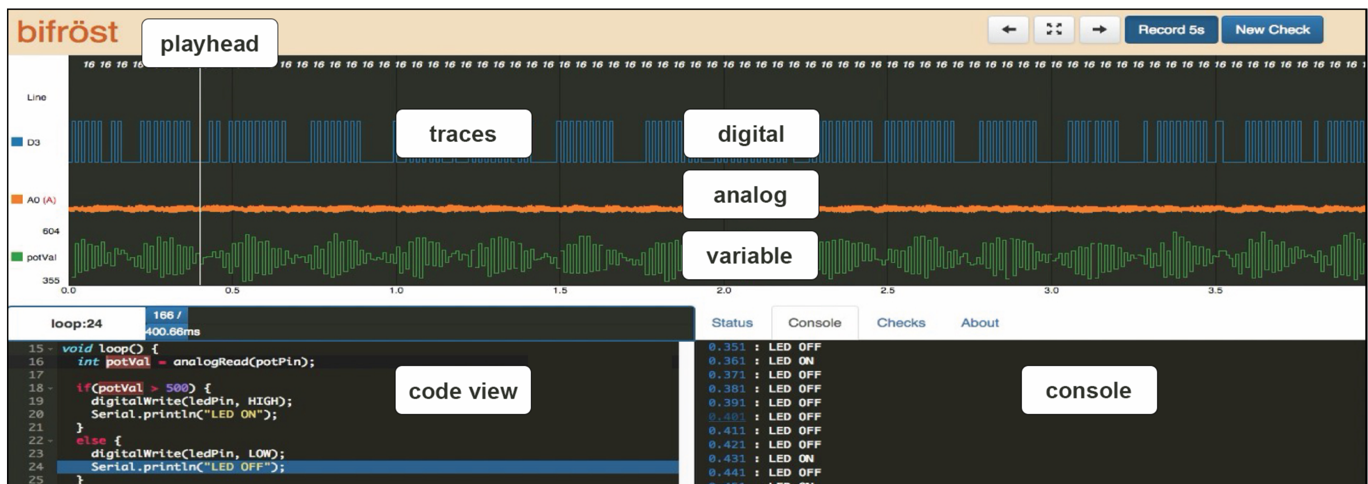

Figure 2. An overview image of the Bifröst UI. The trace view panel allows for digital signals, analog signals, and variable values to be visualized in the same location. The code view panel is a standard code editor with the added ability to use the line numbers to jump to the next point in the recorded runtime trace when that line was executed. It also allows for users to double click on any variable to include its values from the code alongside trace data. The serial console adds timestamps to each print statement and links them to their respective execution point in the trace view.

embedded systems. To test this intuition, we built Bifröst, an integrated hardware/software debugging tool[1] (see Figure 1). Bifröst consists of three major components: instrumentation, an IDE, and validation. First, we introduce hardware and software instrumentation to simultaneously capture key program and electrical behavior of an embedded system. This includes execution timing for each line of code, variable values, digital circuit signals, analog signals, and communication protocols (e.g., I2C or SPI).

Second, the Bifröst IDE (Figure 2) allows a user to easily navigate and analyze the data captured by the instrumentation and generated by the validation. These visualizations allow reasoning about timing between hardware and electrical signals (e.g., was a button actually ever read while it was pressed?) and comparisons between values in software and hardware (e.g., is a variable in code tracking an analog sensor or overflowing as in Figure 3). *In contrast with prior work which has focused on providing tools for debugging software [18, 22, 13, 21] or circuits (e.g., Toastboard [8]), we directly address the interconnected nature of embedded systems projects.*

Third, Bifröst provides an extensible validation infrastructure for creating and running "checkers", test functions that have access to the information captured by the automated instrumentation. For example, an automatic checker tests whether a write issued to a hardware pin succeeded or failed (because of miswiring or pin misconfiguration), or whether a bus read resulted in data returned from a sensor or not (because of an incorrect connection or protocol implementation).

Bifröst supports both *retrospective* (runtime trace-based) and *introspective* (breakpoint-based) debugging. In retrospective mode, an instrumented version of a user's program is run for a set duration. The instrumentation transmits state information on a side channel (a hardware serial port). This side channel is then captured, along with the electrical activity occurring on the microcontroller's pins, using a logic analyzer. Simultaneous capture provides tight time synchronization between software and hardware states. This mode is ideal for situations where it is important for a program to run continuously in near real-time in order for a user to be able to interact with inputs or for adherence to a communication protocol.

Introspective mode is an extension of standard breakpoint debugging. It allows a user to set breakpoints in their code where program execution will stop and variable values can be observed. After a breakpoint is hit, Bifröst captures electrical activity as a user steps through their code line by line. Introspective mode is particularly useful when a user needs to examine complex program state at precise points in time.

The Bifröst GUI consists of three main views: a standard horizontal graph that can show measured electrical activity as well as variable values in code over time; a code editor to display the user's program; and a tabbed pane that can alternatively display checker settings, checker results, or a dynamic view of program serial console output. All views are linked through timestamps so that navigating to a particular time in one view causes the other views to update correspondingly.

A key advantage of Bifröst's integrated approach is that it allows a user to better understand the context surrounding any measured event. If the user notices that a checker has flagged that an output changed incorrectly, they can easily click on the check to focus the trace view to the event and view what line of their code caused the output to occur, as well as easily navigate backwards in time to the lines and pin transitions preceding the change.

We hypothesize that these interactions provide three primary benefits: (i) They *enable rapid navigation and linking* between a variety of program events: code, signals, checks, and console

[1]Bifröst takes its name from the rainbow bridge connecting Earth (Midgard) and the realm of the gods (Asgard) in Norse mythology.

messages; (ii) They *facilitate direct comparison* between system hardware and software state; (iii) Checks can *proactively inform users* of problems and thus help users determine or eliminate possible root causes of problems.

To investigate these hypothesized benefits, we conducted an exploratory study where 10 participants located problems in embedded projects with and without Bifröst. We found that participants particularly appreciated being able to view and compare hardware state and variables on the same set of axes. Notably, users took advantage of the linked navigation features with greater frequency during the more complex study tasks.

RELATED WORK

We first review studies of embedded debugging scenarios and end-user debugging more generally to distill implications for our system. The most closely related technical contributions are tools and user interface techniques for the debugging of interactive and tangible systems.

Studies of Interactive Device Debugging

Booth [4] studied debugging during a physical computing task. They found that while most bugs occurred in programming, the majority of major, failure-causing bugs were due to hardware issues. Further, participants often misdiagnosed hardware errors as software bugs and tried to fix the perceived problem in code. Teteroo [28] discusses key challenges faced by end user developers of interactive devices and points out that issues on embedded devices are particularly difficult for users to understand and debug. Drew [8] reports on insight from electronics teachers and domain experts. These experts noted that generating accurate hypotheses was the most challenging aspect of debugging, which led to a common strategy of falling back on a prescribed, exhaustive test pattern.
Implications: These findings motivate our work in bridging the hardware-software gap during debugging, especially in helping user localize faults in software or hardware.

We would also like to draw attention to concurrent research efforts at UIST that cover generative embedded system design [2], prototyping of analog circuits [26], automatic recognition of components on a breadboard, [35] and measurement of current flows [34] for breadboard circuits.

Studies of End-User Software Debugging

Since we target Makers who are frequently not professional electrical or software engineers, understanding requirements of effective end-user debugging tools is important.

Ko describes six learning barriers for end-user programmers; effective debugging in particular requires overcoming *understanding barriers* about interpreting externally observable state and *information barriers* that keep internal state obscured [19].
Implications: Make more internal state visible via instrumentation; provide context to interpret externally visible behavior.

Gugery and Olson [10] show that novice programmers have difficulty generating high-quality hypotheses about faults.
Implications: Automatically generate and test application-independent hypotheses, e.g. that software commands lead to corresponding events on hardware pins and vice versa, and provide hints when such tests fail.

LaToza [20] notes that difficulty in searching through code and traces for relevant information as well as difficulty exploring program control flow leads to the most time consuming debugging events. Kissinger's study [16] of spreadsheet debugging showed that the most important explanation need was about "Oracles" – finding out if a value in the user's code is right or wrong. They also point out that debugging tools should support global analyses, not just local.
Implications: Provide interactions to rapidly and effectively navigate recorded trace information; provide context to facilitate higher-level reasoning.

Techniques and Interfaces for Software Debugging

Algorithms and user interfaces to help developers with debugging tasks have long been a topic of research. Survey papers of prior work cover techniques for monitoring program execution [24], intelligent or automated debugging techniques [9, 27], and fault localization [33]. In addition to technical work, research has also described fundamental causes of program errors, and how such an understanding can help professional software engineers and end-user developers with their debugging practices [36, 17].

A number of recent systems contribute user interfaces for record/replay and trace debugging. Omniscient Debugging [30, 25, 21] lets developers move forward and backward in time through a program execution, based on a complete trace of all program state. Hoffswell et al. [14] provide recording and retrospective analysis of event traces in interactive data visualizations. Theseus [22] provides always-on visualization cues about runtime behavior as well as a logging scheme that simplifies visualization of control flow. Their user study feedback brought up questions related to the tradeoff between instrumentation overhead and the concept of omniscience. Unravel [12] allows for real-time monitoring of web based source code and explored methods of organizing the large amount of interconnected information. Telescope [13] uses a conceptually similar approach to instrumenting and monitoring the relation between Javascript code and web pages. d.tools [11] and DejaVu [15] both incorporate visualizations of program state along with time-series sensor data in order to aid program development, at different levels of abstraction from Bifröst.
Implications: Develop trace navigation tools that preserve context. Users respond positively to omniscient tools but can be overwhelmed by too much additional information.

In contrast to trace visualizers, the Whyline [18] introduces the idea of interrogative debugging, seeking to answer end user questions related to causality. In our system, checkers point out potential root causes, but they cannot yet determine which of multiple root causes is most likely.

Hardware Debugging Tools

Work in this area can be divided into tools developed specifically for embedded systems and tools developed solely for interacting with circuits. The Toastboard [8] introduces checks at the circuit level, based on a user-provided schematic. We extend their work to checks that cross hardware and software. Recent commercial products, for example the Digilent Analog Discovery or Electronics Explorer [7], target students and aim

Design Goals	Realization in Bifröst
Support users with localizing a fault across software and hardware [4]	Present information about both domains in a common interface
Make internal state visible [19, 31]	Hardware and software instrumentation at the granularity of a single line of code and signal transition; Protocol decoding
Provide context to interpret externally visible behavior [19, 31]	Console output and writes to hardware pins are linked to code through timestamps
Automatically generate and test application-independent hypotheses [10]	Automatic checkers
Rapidly and effectively navigate recorded trace information [20]	Linked views: access traces from code and vice versa; panning and zooming
Support global analyses [16]	User-defined checks: *when x happens in hardware, ensure line y executes within z milliseconds*

Table 1. Mapping design goals extracted from prior work to interaction techniques and the realization of features in Bifröst.

to replace the various pieces of bench-top equipment needed for hardware debugging; they combine the input/output capabilities of multiple tools (e.g., oscilloscope, logic analyzer, arbitrary function generation), but do not bridge hardware and software. The standardized JTAG interface allows for execution control of most commercial microprocessors.

INTERACTION DESIGN

The goal of Bifröst is to improve debugging of embedded systems by displaying and linking hidden aspects of execution context and running proactive automated checks. Table 1 summarizes how implications from prior studies of debugging find expression in Bifröst. Our system is further motivated by Victor's essays on Learnable Computing [31] and Seeing Spaces [32], which argue that programming environments should: make time visible and tangible; enable analysis at multiple granularities; eliminate hidden state and show the data; and facilitate comparisons.

Context and Navigation

A central idea underlying Bifröst is that viewing data about the program and electrical state histories of an embedded system simultaneously can make it easier to discover bugs. The main part of the UI shows program execution history on the same time axis as the electrical activity on the pins of the microcontroller (see Figure 2). The code editor displays the program that generated the trace and links selections on both views. The visualizations allow users to test hypotheses by comparing expected values at different points in their system side by side to discover any issues. Users can step forwards and backwards through code lines with arrow keys or the UI.

We have designed and implemented a variety of linking mechanisms between the events Bifröst captures to make it easy for a user to jump to events of interest in a trace before exploring their vicinity for clues about the source of an error. These linking mechanisms are summarized in Table 2. At the most basic level, clicking on a location in the overview trace zooms the UI onto the line of code that was running at that instant. When a user clicks on a code line number in the editor, the UI jumps

Location	Result
Click on trace or progress bar at time `t`	Focus on line executing at `t` and highlight line in editor
Left or right arrow	Focus on previous or next line execution and highlight line in editor
Click on line `l` in editor	Focus on next execution of line `l` and highlight line in editor
Click on timestamp `l` in console	Focus on line executing at `l` and highlight line in editor
Click on checker result in checkers	Focus on line executing at `t`, show check time bounds, and highlight line in editor

Table 2. Summary of navigation links between data types in Bifröst.

to the *next* execution of that line in time (it may execute many times). In this way, a user can quickly skip forwards from a less "interesting" instance of a line to search for a time when the same line executes under more "interesting" conditions. This also enables testing if a line never executes. Since there is no event for the UI to jump to, a warning is displayed instead.

Debugging using print statements is pervasive [1], especially in embedded code. Bifröst enhances ordinary "print statement" debugging by turning each printed line into a link back to the execution context that generated the line of text. A user can quickly skim the integrated console window for a given string or values and then click on the automatically-generated timestamp to focus the UI on when the line of code that printed the text ran. From there they can use the navigation buttons to view the surrounding program and electrical state.

Proactive Checking

The goal of checks in Bifröst is to enable the system to proactively aid the user by offering suggestions about likely errors and their causes and by validating the behavior of their program. The two kinds of checks implemented in Bifröst correspond to these two goals.

Automatic Checks

Bifröst comes with an extensible set of checks that automatically infer all information necessary for them to run based on the user's code. For example, if a user authors code to write a value to a digital output pin, that write will only succeed if that call is preceded by a call to set that pin as an output. The system automatically validates that this relationship holds for all writes that the user issues by examining the execution history derived from the captured traces (see Figure 4). The full set of automatic checks is shown in Table 3.

User-Defined Checks

There are a variety of important program properties that cannot be directly inferred from the code. For instance, consider when an application is waiting for a button to be pressed before taking an action. It is reasonable to assume that when the voltage on the input pin changes due to the button being pressed the program should take some action, but the checker infrastructure cannot automatically determine what that action should be. In these cases we allow for the user to author checks of the form "when D4 goes low expect line 65 in 20 ms" using the interface shown in Figure 5. These checks can validate any combination of a pin, transition, line number, and time bound. Clicking the failed check links to the time during the trace at

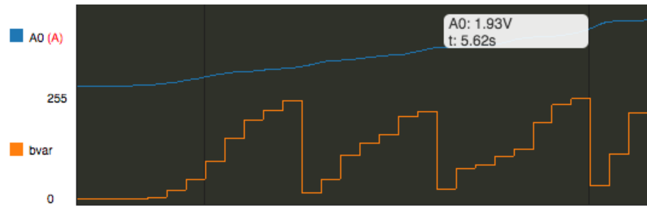

Figure 3. Visualizing internal variable values alongside electrical trace data. A0 is an analog input pin; `bvar` is a byte variable that stores values read from A0. This example shows how side-by-side comparison exposes an overflow of 10-bit analog data in the 8-bit `bvar`.

Name	Description
pinMode	Detects if pins are properly initialized with `pinMode` statements before any read and write operations on that pin
digitalWrite	Detects whether digital software write operations update the hardware state of the target pin successfully
SPI Protocol	Detects whether protocol write operations result in the correct message on the external bus

Table 3. Automatic checks supported by Bifröst.

which it occurred so that the user can begin to examine the context around the error.

Transparent Instrumentation

In Bifröst, we instrument and modify a user's program in order to make internal variable and execution state available in the editor. This is implemented through source code rewriting. While the instrumentation overhead can negatively affect the performance of the user's program, the user is never confronted with the rewritten code. In particular, when stepping through code, Bifröst performs the bookkeeping to step at the granularity of the original, uninstrumented source program, not the rewritten program.

A STROLL ACROSS BIFRÖST

This section presents two running examples to illustrate how the system would be used in realistic scenarios.

The People Counter

Freya wants to build a system that counts the number of people who enter and leave her shop over the course of the day. She decides to first make a prototype using an Arduino, a breadboarded circuit, and Bifröst. She'll use two pairs of infrared LEDs and IR sensors to detect whether the people are entering or leaving. A timeout condition will make sure that a momentary false positive on a single one of the sensors – for example, from someone turning around in the doorway – will not cause undesirable behavior. Her circuit schematic and a depiction of the intended operation are shown in Figure 6a.

Immediately after compiling and uploading her code to the Arduino, the Bifröst status panel notifies her that one of the built-in system checkers has found an error. Inspecting the errors list, she sees that she has failed to initialize the pin mode for one of the output LEDs (Fig. 6b).

She adds a pin mode initialization and re-compiles the code; during the recording she simulates someone entering and then leaving by sweeping her finger between the two IR detectors

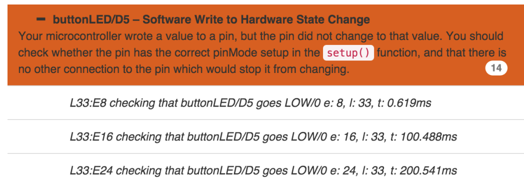

Figure 4. A subset of the Bifröst check status UI showing three instances of a failed `digitalWrite` check. Each test instance lists details about the environment where it failed including the line number and time.

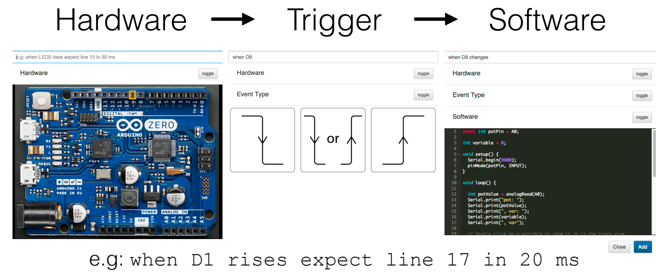

Figure 5. The interface for constructing user-defined checker functions. Users can select an Arduino pin, then select an edge type trigger (rise, fall, change), and a target line. The result is a unique semantic identifier for the check, which can also be modified by the user.

and emitters. She can see voltage values rising when her finger crossed in front of the sensors but the counter, visible in the Bifröst serial console, still doesn't change. Noting that the recorded analog values do not get all the way to 3.3V, Freya decides to see what the actual variable value is. In the code editor, double clicking on the variable that stores the analog read will track it during the next recording. She rerecords the data. By hovering over the new variable trace she can see that it never reaches the threshold she had set in her code (Fig. 6c).

After fixing this thresholding problem the counter is still not incrementing. Freya decides to use the Bifröst's ability to tie code line execution to the corresponding point in the trace to figure out why. She clicks repeatedly on the line number of the timeout function and notices that it is occurring far too often. Tracking the relevant variables tells her why; she has set the timer to microseconds and her timeout value is coded in milliseconds (Fig. 6d).

Finally, the counter is incrementing when she "enters," but now it rapidly increases even when she "leaves." She suspects it is an issue with her state flags and double clicks to track these before re-recording. Clicking on one of these incorrectly incremented values in the serial console focuses the interface on the point in the trace recording where it occurred. By using the left and right arrow keys, she can then move backwards and forwards in time through the program. After some exploration it becomes clear that the flag for "entering" is remaining set even after she has completed the action (Fig. 6e).

A Maker's Lock

To further equip her shop, Freya next decides to create her own Arduino-controlled combination lock. A potentiometer acts as the dial and a button latches in the current value. After entering a three-value combination in the correct order, a green

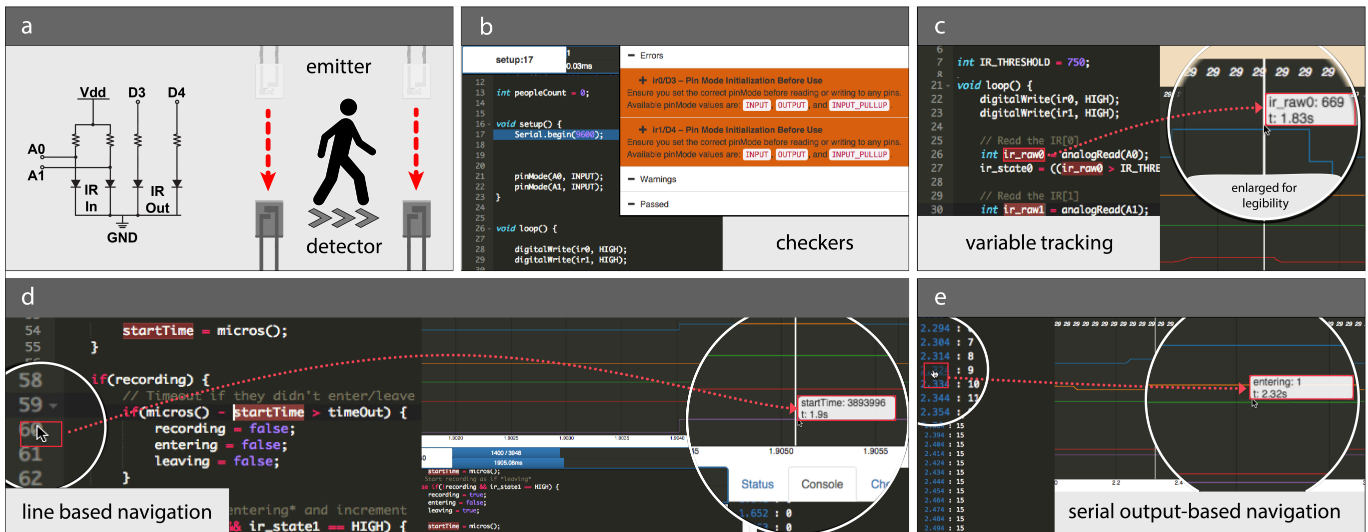

Figure 6. a) Freya's circuit schematic and an annotated picture of the breadboarded circuit. b) The system checker interface flagging a pinMode error. c) The trace view showing analog signals alongside their internal system representation. d) Clicking on a line number in the editor moves the trace view focus to that point in the recorded execution. e) Clicking on a serial print statement in the Bifröst serial terminal also moves the trace view focus to that point in the recorded execution. From there, the code can be stepped through in time using the left and right arrow keys.

LED will light up (she will replace it with an actuator later). If an incorrect value is given, a red LED will flash twice.

Currently the program seems to have a bug where, after the combination is correctly entered and the green light comes on, it immediately switches off and signals a bad value input. Freya switches to introspective debugging mode in Bifröst. She sets a breakpoint at the "correct combination" location in her code (Figure 7a) and runs the program; once she reaches that point, execution halts. She can hover over any variable in the editor window view the current value. She can also "step" to capture another slice of program and electrical activity. Since execution is paused except between steps, it's easy to try different combinations of button presses and potentiometer values. It soon becomes clear that, by prematurely clearing one of her system flags, she is immediately dropping in to the "incorrect value" statement after the "correct combination" case (Figure 7b), which explains why the green LED is immediately being turned off.

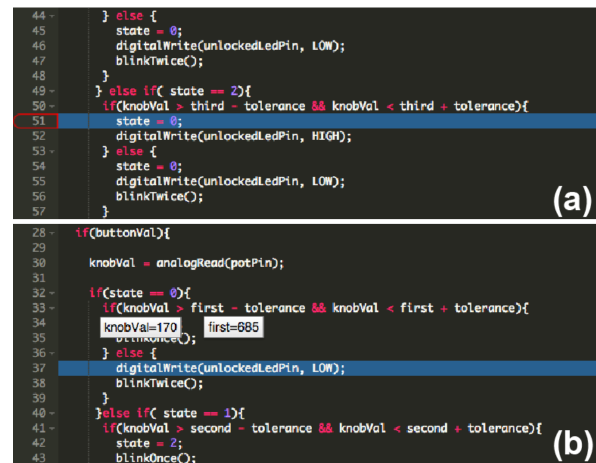

Figure 7. a) The IDE in introspective mode showing a breakpoint at the "success" case. b) Hovering over variables reveals their current values.

IMPLEMENTATION

Bifröst captures program activity as well as analog and digital electrical signals simultaneously using a 16 channel programmable logic analyzer and runs checks on the collected data and visualizes the code and data in its IDE (Figure 1).

Program information is collected in one of two ways: In *retrospective* mode, the user's code is instrumented to transmit line numbers, parameters of system calls, and variable values through a hardware serial port on the microcontroller. When the program is run, the port is also captured by the logic analyzer so electrical and code information are captured simultaneously. In *introspective* mode, the code runs with minor instrumentation until a breakpoint is reached. Bifröst then uses the GNU Debugger, GDB, and OpenOCD, an on-chip debugging library, to interrogate program state at the breakpoint.

Once data has been collected, either after a complete run in retrospective mode or a single step in introspective mode, Bifröst loads and runs its library of built-in and user-specified checker test functions against the recorded data.

Hardware

Bifröst uses an off-the-shelf microcontroller and logic analyzer; the two are connected through a custom PCB. We describe important design requirements and how our hardware choices satisfy these requirements.

Microcontroller: The target microcontroller needs to provide two important hardware resources: first, a hardware UART not used by the user's program to emit side-channel program information for retrospective mode; and second, a debugging interface for introspective mode. Our system currently supports the Arduino Zero [3], a 32-bit ARM platform popular

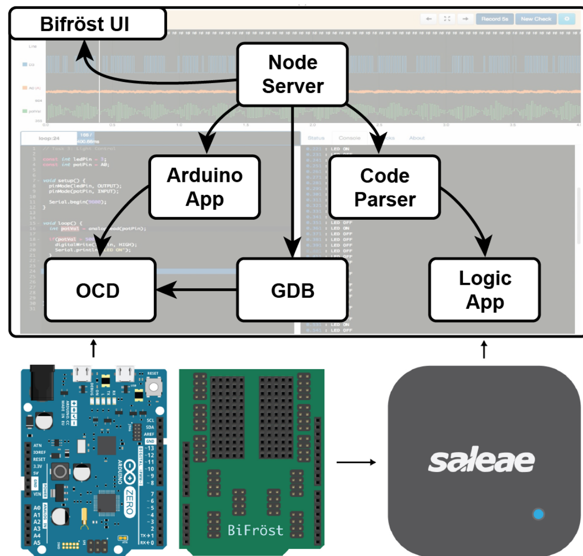

Figure 8. The connections between Bifröst's hardware and software.

with makers, which runs programs written in C or C++ and offers OpenOCD-compatible debug support over USB.

Logic Analyzer: The logic analyzer needs to support programmatic configuration and data capture. We use a Saleae Logic Pro 16 analyzer with 16 channels and configure it through its scripting API to set capture durations, sample rates (up to $500 MS/s$ digital and $50 MS/s$ analog), triggers, and channel configurations (analog or digital). The API provides protocol analyzers to decode bus activity (I2C and SPI) as well as our side-channel serial program information.

Instrumentation through a Custom Shield: The Bifröst PCB is an Arduino shield that facilitates connecting the logic analyzer to the Arduino's pins while allowing them to still be connected to a user's circuit. Users can also choose to build their circuit directly on the shield. One trade-off of this instrumentation approach is that we only capture activity at the microcontroller pin, not at arbitrary locations within the circuit.

Software

The Bifröst backend, written in node.js, manages the embedded build process for the user's code (instrumentation, compilation, and upload); it configures the logic analyzer and reads its data; and it controls the target microcontroller's on-board debugger in introspective mode. The backend also serves the browser-based user interface (see Figure 8).

Retrospective Mode

When a user clicks "Record" in the UI to collect a retrospective trace, their program is parsed and instrumented to output execution data to the microcontroller's side-channel, the logic analyzer is configured to capture, the program executes, and then data from the trace captured by the logic analyzer is parsed in order to create the Bifröst UI. Our instrumentation assumes that user programs follow the Arduino convention of an initial setup function followed by a main loop.

We parse the user's program and perform these modifications:

Trigger We write a short pulse to a reserved microcontroller pin at the start of the program that serves as a trigger for the logic analyzer to start capturing program behavior.

Line numbers To provide location information, we print the line number of each of the user's statements to the spare hardware UART, before executing the statement itself.

Console messages We redirect any console output (Serial.print) to the side-channel so it is captured and timestamped by the logic analyzer.

Variable values For variables identified by the user, we print values to the side-channel at the end of each main loop iteration. This approach is simple, but misses some transitions. This is discussed further in the Limitations section.

Core API parameters We modified Arduino's core library to print the arguments of commonly used functions like `digitalWrite()` to the side-channel. This allows checkers to inspect when an Arduino core function is called either by user code or even by otherwise uninstrumented library code and test whether it succeeded in hardware.

Different types of messages are distinguished using custom opcodes on the side-channel. Instrumentation is not free — these operations introduce an overhead that decreases program performance. Empirically, we determined that the slowdown on several example programs from the Arduino library ranges from practically none to as high as 9x (Table 4). The worst case occurs when a loop calls core functions without any pauses. In our system, the root causes for the delay are blocking UART writes when the transmit buffer is full, which occurs because the processor runs at 48MHz while the side channel can only output data at 2M Baud. However, the presence of even milliseconds-long pauses in the user's loop give the serial output time to catch up and avoid blocking execution. Wider buses, faster communications peripherals, or selective instrumentation could help mitigate this effect in the future.

We evaluated one alternative side-channel for inclusion in our system - SEGGER's JTAG-based RTT (Real Time Transfer) library. In this approach, debug information is written to a ring buffer in memory that is read transparently over JTAG. However, even though the JTAG interface could reliably run at 5x the speed of the UART, this approach incurred additional overhead for accurate timestamping, which cut into the potential performance gain. The unpredictable nature of the JTAG reads also made it difficult to synchronize the buffered data with the execution of the program and we ultimately did not integrate it into Bifröst. Although we did not evaluate them, we are also aware that some M-series microcontrollers include a sub-module that facilitates instruction-level trace output that could be captured to address the synchronization issue.

The last step before a retrospective trace can be captured is to configure the logic analyzer. The user specifies a capture duration in seconds, which is passed directly to the analyzer. The user can select any pin in the UI to capture an analog trace for. Based on its internal mapping from Arduino pins to logic analyzer channels, Bifröst translates user-specified pin selections to the correct logic analyzer channel so that the user does not need to keep track of the correspondence.

	Button	Debounce	Fading	Blink
Uninstrumented	1.00	1.00	1.00	1.00
Line Execution	2.81	3.32	1.00	1.00
Core Functions	5.54	7.01	1.00	1.00
Line & Core	8.86	9.82	1.00	1.00

Table 4. Summary of performance impacts of code instrumentation. The cell values represent slowdown factor where 1.0 is full speed.

Introspective Mode

When introspective mode is started, just as in retrospective mode, the user's code is parsed and instrumented, the logic analyzer is configured, the code is run, and the results are displayed in the UI. However, introspective mode allows breakpoint debugging using a GDB interface directly to the EDBG (embedded debug) module on the Arduino Zero's board.

Introspective mode involves less instrumentation as program state can be retrieved through GDB commands. The introspective parser only adds the initial logic analyzer trigger pulse code and wraps each user program statement with two statements that signal the start and end of the command. The instrumentation step must also build a correspondence between the line numbers of user code (used in the UI) and the line numbers of the instrumented program (used by GDB).

After flashing a new program, the Bifröst server opens up a GDB session, which in turn communicates with OpenOCD, which then communicates with the EDBG module on the Zero. We then reset the microcontroller and run until a breakpoint is reached. Each "step" command in introspective mode works similarly to a short retrospective mode trace. When the user performs a step in the UI, Bifröst starts the logic analyzer, orders the microcontroller to step, and then parses and appends the incremental results in the UI.

A key benefit of introspective over retrospective mode is that GDB can instantaneously read any variable value within the current scope of program execution. This precision allows for two convenient interactions within the UI. First, a user can select variables to "watch" and plot their values over the execution of the program at a finer granularity than retrospective mode (each write, instead of each loop). Second, Bifröst allows users to mouse over any variable in their code while execution is stopped and see the current value as a tooltip even if the variable is not currently being tracked (Figure 7b).

User Interface

The user interface is served by a node.js server and is responsible for parsing the files generated by the instrumentation, building a unified history of program execution, and displaying an interactively-linked visualization (written in d3 [5]) based on the execution data. In introspective mode, the frontend shares responsibility with the backend in managing the GDB debugging session. The frontend communicates with the server over REST and a websocket and makes use of Bootstrap [29], the open-source Ace editor [6], and jQuery.

Checker Functions

Automatic checks test for behaviors where the correct outcome is not application-specific. Checks have access to the user's source code, a log of Arduino core and line events, and the trace activity for each pin. The checks are implemented as a set of Javascript functions that take program trace information as input, find events of interest within the trace, and evaluate a relationship for those events. The system then receives a list of those events and corresponding messages about whether the relationship held, which are displayed in the user interface. For example, the `digitalWrite()` check finds instances of digital writes in a trace and ensures that the output pin's voltage level matches the argument of the function. Psuedo-code for the `digitalWrite()` checker is shown in Figure 9 and the list of currently implemented checks is shown in Table 3.

```
def digitalWriteCheck(event, index, all)
  if event.core == digitalWrite  # only test writes
    pin = event.targetPin        # target pin (D1)
    val = event.targetVal        # write value (HIGH|LOW)
    cur = all.traces[pin][index] # read HW pin state
    # read next value of the target pin
    nextVal = all.traces[pin][index+1]
    if nextVal != val
      return Error   # write failed to change HW pin
    else if cur == val
      return Warning # pin was already set to value
    else nextVal == val
      return Passing # pin successfully changed
```

Figure 9. Pseudo-code for `digitalWrite()` Checker.

User-defined checks are explicitly defined by the user, and can express expected behavior of a particular program. User checks are represented as simple strings describing what to check, e.g. "when D4 changes expect line 65 in 20 ms". Our system parses these strings and generates a new function based on the parameters and structure of the input string. This new checker function is executed over the last recorded data trace, and the checker results are included in the Status tab.

USER EXPERIENCES WITH BIFRÖST

In order to explore how Bifröst changes debugging workflows, we conducted an exploratory evaluation with 10 participants (7 male, 3 female), all university students. Participants were screened for previous experience in both embedded systems programming and multimeter use. Their experience levels ranged from novices with little circuit prototyping and Arduino programming experience (4 participants), to intermediate builders with moderate Arduino experience (3 participants), to expert embedded systems developers that had experience working with logic analyzers (3 participants).

Methodology

After a brief introduction to Bifröst's core functionality, participants were asked to perform six debugging tasks. Each participant was given a pre-built circuit connected to a Bifröst shield, a diagram describing intended functionality of the device, and an Arduino pre-loaded with code containing known errors. For the first five diagnostic tasks, each program contained a single error. Participants were asked to find and correctly diagnose the root cause of the problem and briefly explain how they would address the issue within seven minutes. If the allotted time for a diagnostic task expired, participants were asked to make a diagnosis before moving on. Each participant diagnosed three tasks with Bifröst and two tasks using a multimeter and the Arduino IDE's built-in serial monitor and plotter as a control condition. While we are not primarily

interested in quantitative performance differences, this design enables participants to directly compare their experience with and without Bifröst. The last debugging task was larger in scope: it contained three problems for participants to both identify and resolve using Bifröst within 30 minutes.

Diagnosis - T1: LED not flashing A circuit to blink two LEDs, but one is missing a `pinMode()` and does not blink.
Diagnosis - T2: Broken Toggle A non-debounced button causes an LED to toggle randomly.
Diagnosis - T3: Noisy Potentiometer A potentiometer that should turn on an LED at a threshold voltage causes the LED to flicker near the threshold due to a noisy signal.
Diagnosis - T4: Incorrect Timer A stopwatch that should count upwards, but contains a counting logic bug.
Diagnosis - T5: Complex Program A bar of LEDs controlled by a potentiometer using 200-lines of switch-case statements. There is an error in the output logic.
Debug - T6: People Counter Users were presented with the "People Counter" example from the case study and given 30 minutes to find three errors: (i) an incorrect threshold value causing analog signals from the IR sensors to not trigger a flag variable, (ii) a too short error-rejection timeout causing the system to reject valid inputs, and (iii) directionality flags persisting after a successfully count, causing the system to increment its count incorrectly.

We concluded the user-study with a survey regarding the usability and perceived utility of several Bifröst features using a Likert-scale (ranging from 0 to 5). The survey also contained a number of open-ended questions about debugging strategies. For the Likert-scale questions we report means (μ); a score of 5 corresponds to *"Very Useful"* or *"Not Difficult at All."*

Findings

Diagnostic Tasks (T1-5): Nine of ten participants clearly preferred Bifröst and showed very different patterns of tool usage across conditions and tasks. Tasks T1-T5 were easy enough that task completion was not a particularly meaningful metric. Participants that were assigned Bifröst diagnosed the problem correctly 80% of the time (24/30); control participants diagnosed the problem 90% of the time (18/20).

Debugging Task (T6): All participants used Bifröst. Seven participants were able to fix at least some of the errors, but only two self-described "expert" users completed all parts of this task. This suggests that either the task was too difficult to complete in the allotted 30 minutes, or that there may be a skill threshold to effective Bifröst usage.

Comparison of Bifröst and Control Usage: Bifröst's transparent instrumentation strategy seemed to both (i) reduce the cognitive load associated with instrumentation and contextualizing information as well as (ii) reduce the number of errors introduced in the act of instrumentation.

Control group participants often ended up neglecting instrumentation entirely, opting to parse the code and try to keep a mental map of execution. Conversely, all ten participants found the trace visualization and the variable tracking features of Bifröst to be useful (μ=4.6) and easy to use (μ=4.7). One Bifröst user noted that *"I don't have to play computer in my head, I can track the variables!"* Another opted to track every possible variable at once after deciding that the required effort to instrument in this manner was less than the effort required to re-record a trace. Participants remarked on the advantages of the "omniscient" nature of retrospective debugging; a third user said *"one more thought on why I like Bifröst over serial plotter is that you can step forward and backwards in time rather than just having a continuous real time monitor."*

In the control scenario, introducing program instrumentation to track variables (through serial prints) led to errors such as forgetting to initialize serial communication. In four cases the participant became confused by the functionality of the Arduino serial plotter - which variables correspond to which line, if the monitor could be used at the same time as the plotter, and by the proper code syntax to use. Two participants wasted time ensuring their instrumentation code was functioning as expected by printing test text.

When asked about which debugging method they preferred, all participants except one (who was neutral) favored Bifröst over traditional methods. Participants mentioned that *"Bifröst works very well for thinking through the problem"*, *"Bifröst saves a lot of time on displaying results"*, and *"Especially with these real-time programs, being able to slowly analyze the state of the code was nice"*. While acknowledging the usefulness of various features of Bifröst, three participants mentioned that they needed more time to learn how the tool could help them, or felt overwhelmed with the amount of information available.

Bifröst Debugging Strategies: For debugging *hardware-interfacing* problems (T3) we observed a *Variable/Signal Comparison* strategy that involved hovering over traces in the UI to compare the values of tracked variables to raw I/O signals. We also observed usage of *Automatic Configuration Checks* to verify or find a pin configuration error.

For debugging *timing* issues (the second part of T6, where the system always times out), we observed the following debugging strategies: (i) *Time-Sensitive Tracking* in which participants tracked a variable in time and monitored how its value changed over time in the visualization, (ii) *Statement Frequency Evaluation* in which consecutive line number clicks on a particular line showed the user how frequently a particular statement was executed and (iii) *In-situ Inspection* in which participants used the console to print a variable, and then used the serial time-stamp to advance the environment to a specific iteration of the main loop.

For debugging *logical* issues (T4: Incorrect Timer), participants primarily used the (i) *Multi-Variable Comparison* strategy, in which participants visually compared the values of multiple variables by hovering over the traces in the trace-view. We also noticed a (ii) *Statement Reachability Evaluation* strategy, in which users used line number clicks as a way to examine if a statement was ever executed.

System Understanding: Participants began to exploit the linked navigation features of Bifröst as they spent more time with the tool in T6. One subject said *"I really like the fact that I can see when the lines of code were executed [by clicking*

on the line numbers]!" Another reported that *"being able to go from 'time in the trace view' to 'line in code' was really useful, since I didn't have to find the guilty line of code"*. These statements reflect both an understanding of the interplay between the interface elements and an appreciation for the transparent instrumentation.

Sources of Confusion: Users occasionally encountered problems when attempting to take advantage of the navigation and visualization features. Three participants mentioned that they were frustrated because the trace view did not maintain the zoom level when stepping through code, resulting in a loss of information context. One user pointed out that the variable values did not always seem to align with the electrical trace so they no longer knew if they could believe the visual model.

Some of the users experienced confusion regarding the nature of retrospective debugging. One user asked: *"If I highlight them [the variables], do I need to re-run the thing?"*. Another user stated that *"one thing that isn't very clear is whether I use 'select active' before or after recording?"*. A third user asked *"is it running right now?"* before starting a recording.

Finally, we noticed that three novice users did not use some of the key features. One novice and one intermediate user never used the line number navigation links, and one other novice user used print statements instead of line number clicks in order to determine if a line of code was ever executed.

The above issues might have been reduced by spending more time training users before the study tasks. The struggles of the inexperienced users and the difficulties they faced with T6 highlight the fact that Bifröst assumes a level of debugging familiarity: users have to have an appropriate mental model of their embedded system in order to effectively use the information and navigation affordances that Bifröst provides. Finding ways to reduce this threshold will be interesting future work.

LIMITATIONS

Not All Errors Can Be Diagnosed: Bifröst focuses on revealing state inside the user's code and directly at the pin level of the microcontroller running the code. It assumes that the microcontroller board and its core API are not sources of errors. It also does not capture information about the user's circuit (e.g., a schematic), doesn't facilitate collecting traces at arbitrary points in the user's circuit, and does not instrument library code the user includes. These choices are based on the assumption that many errors lie either in the user's code, or at the interface between the microcontroller and the circuit.

An equally fundamental limitation is that Bifröst runs an instrumented and hence modified version of a user's program. The instrumentation necessarily affects precise timings within a program, which often are immaterial, but does open the possibility of errors that are either hidden due to or even caused by the instrumentation.

Performance Overhead: Bifröst's approach of adding lines of code to a user's program to instrument it decreases program performance. We hypothesize that this is unlikely to impact most projects built in educational settings and by hobbyists. However it is not as suited for applications with microsecond-level real time execution constraints, such as "bit-banging" (emulating a communication protocol with precisely-timed digital output) or in industrial embedded systems.

Resource Requirements: Bifröst requires exclusive use of three digital GPIO pins, including one Serial bus, on the target microcontroller to enable its instrumentation and communication. Whether the loss of three GPIO pins is significant is application-dependent (a regular Arduino board has 14 GPIO pins available). Currently, Bifröst handles Serial print statements by rerouting user Serial debugging commands into its side channel and UI, effectively "sharing" the bus. However, this precludes Serial communication for other purposes. A simple fix would be to duplicate serial messages and send them to the original port and the side channel.

Manual Configuration: Because there are more pins available to the user than logic analyzer channels in our implementation, users have to select which subset of pins to instrument by manually plugging cable headers into four of six possible positions. This limitation could be overcome with a bigger analyzer or an analog crossbar on the PCB.

Scaling: Our UI can render tens of thousands events occurring during a trace without perceptible delay. However, much larger traces (e.g. minutes of execution) will require rewriting the rendering engine.

Limited Granularity: For performance and simplicity, the values of variables that are tracked in retrospective mode are only updated at the end of every execution of the main loop. This could be addressed by searching for and instrumenting every assignment of a variable in a future iteration.

CONCLUSIONS AND FUTURE WORK

Bifröst is a novel system that combines tools and techniques from hardware and software debugging in a unified environment in order to support debugging embedded systems. It provides a checking infrastructure that supports both checks inferred from the user's code to proactively provide hypothesis testing and user-defined checks that can continuously validate program properties.

Surveys from a ten-participant user study confirmed that on average, users found Bifröst's features intuitive and effective. A longer term deployment could provide additional insight into usage patterns and strategies that arise when the tool is used in a greater variety of scenarios.

Future work will focus on incorporating more measurements about a user's circuit into Bifröst's checking and display infrastructure. We will also attempt to more smoothly bridge the gap between Bifröst's introspective and retrospective modes in order to better address the unique mixture of time domains encountered while designing interactive devices.

ACKNOWLEDGEMENTS

This work was supported in part by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and by NSF awards CNS 1505728 and IIS 1149799.

REFERENCES

1. Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. An analysis of patterns of debugging among novice computer science students. *ACM SIGCSE Bulletin* 37, 3 (2005), 84–88.
2. Fraser Anderson, Tovi Grossman, and George Fitzmaurice. 2017. Trigger-Action-Circuits: Leveraging Generative Design to Enable Novices to Design and Build Circuitry.. In *Proceedings of the 30th Annual Symposium on User Interface Software and Technology (UIST 2017)*. ACM, Quebec City, QC, Canada.
3. Arduino. 2017. Arduino Zero. https://www.arduino.cc/en/Main/ArduinoBoardZero. (2017).
4. Tracey Booth, Simone Stumpf, Jon Bird, and Sara Jones. 2016. Crossed Wires: Investigating the Problems of End-User Developers in a Physical Computing Task. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 3485–3497. DOI: http://dx.doi.org/10.1145/2858036.2858533
5. Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D^3 data-driven documents. *IEEE transactions on visualization and computer graphics* 17, 12 (2011), 2301–2309.
6. Cloud9. 2017. Ace - The High Performance Code Editor for the Web. https://ace.c9.io/. (2017).
7. Digilent. 2016. Digilent Electronics Explorer Board. (2016). https://reference.digilentinc.com/electronics_explorer:electronics_explorer Online; accessed 30-March-2016.
8. Daniel Drew, Julie L. Newcomb, William McGrath, Filip Maksimovic, David Mellis, and Björn Hartmann. 2016. The Toastboard: Ubiquitous Instrumentation and Automated Checking of Breadboarded Circuits. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 677–686. DOI: http://dx.doi.org/10.1145/2984511.2984566
9. Mireille Ducassé. 1993. A pragmatic survey of automated debugging. *Automated and Algorithmic Debugging* (1993), 1–15.
10. L. Gugerty and G. Olson. 1986. Debugging by Skilled and Novice Programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '86)*. ACM, New York, NY, USA, 171–174. DOI: http://dx.doi.org/10.1145/22627.22367
11. Björn Hartmann, Scott R Klemmer, and Michael Bernstein. 2005. d. tools: Integrated prototyping for physical interaction design. *IEEE Pervasive Computing* 4 (2005).
12. Joshua Hibschman and Haoqi Zhang. 2015. Unravel: Rapid Web Application Reverse Engineering via Interaction Recording, Source Tracing, and Library Detection. In *Proceedings of the 28th Annual Symposium on User Interface Software & Technology (UIST '15)*. ACM, New York, NY, USA, 270–279. DOI: http://dx.doi.org/10.1145/2807442.2807468
13. Joshua Hibschman and Haoqi Zhang. 2016. Telescope: Fine-Tuned Discovery of Interactive Web UI Feature Implementation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST 2016)*. ACM.
14. Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2016. Visual debugging techniques for reactive data visualization. In *Computer Graphics Forum*, Vol. 35. Wiley Online Library, 271–280.
15. Jun Kato, Sean McDirmid, and Xiang Cao. 2012. DejaVu: Integrated Support for Developing Interactive Camera-based Programs. In *Proceedings of the 25th Annual Symposium on User Interface Software and Technology (UIST '12)*. ACM, New York, NY, USA, 189–196. DOI: http://dx.doi.org/10.1145/2380116.2380142
16. Cory Kissinger, Margaret Burnett, Simone Stumpf, Neeraja Subrahmaniyan, Laura Beckwith, Sherry Yang, and Mary Beth Rosson. 2006. Supporting End-user Debugging: What Do Users Want to Know?. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '06)*. ACM, New York, NY, USA, 135–142. DOI: http://dx.doi.org/10.1145/1133265.1133293
17. Andrew J Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, and others. 2011. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)* 43, 3 (2011), 21.
18. Andrew J. Ko and Brad A. Myers. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 301–310. DOI: http://dx.doi.org/10.1145/1368088.1368130
19. Andrew J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*. IEEE, 199–206. DOI: http://dx.doi.org/10.1109/VLHCC.2004.47
20. Thomas D LaToza and Brad A Myers. 2010. Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 185–194.
21. Bil Lewis. 2003. Debugging backwards in time. *arXiv preprint cs/0310016* (2003).
22. Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing Misconceptions About Code with Always-on

Programming Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 2481–2490. DOI: http://dx.doi.org/10.1145/2556288.2557409

23. David Mellis, Massimo Banzi, David Cuartielles, and Tom Igoe. 2007. Arduino: An open electronic prototyping platform. In *Proc. CHI*, Vol. 2007.
24. Bernhard Plattner and Juerg Nievergelt. 1981. Special feature: Monitoring program execution: A survey. *Computer* 14, 11 (1981), 76–93.
25. Guillaume Pothier, Éric Tanter, and José Piquer. 2007. Scalable omniscient debugging. *ACM SIGPLAN Notices* 42, 10 (2007), 535–552.
26. Evan Strasnick, Maneesh Agrawala, and Sean Follmer. 2017. Scanalogue: Interactive Design and Debugging of Analog Circuits with Programmable Hardware. In *Proceedings of the 30th Annual Symposium on User Interface Software and Technology (UIST 2017)*. ACM, Quebec City, QC, Canada.
27. Markus Stumptner and Franz Wotawa. 1998. A survey of intelligent debugging. *AI Communications* 11, 1 (1998), 35–51.
28. Daniel Tetteroo, Iris Soute, and Panos Markopoulos. 2013. Five key challenges in end-user development for tangible and embodied interaction. In *Proceedings of the 15th ACM on International conference on multimodal interaction*. ACM, 247–254. DOI: http://dx.doi.org/10.1145/2522848.2522887
29. Twitter. 2017. Bootstrap. http://getbootstrap.com. (2017).
30. David Ungar, Henry Lieberman, and Christopher Fry. 1997. Debugging and the Experience of Immediacy. *Commun. ACM* 40, 4 (April 1997), 38–43. DOI: http://dx.doi.org/10.1145/248448.248457
31. Bret Victor. 2012. Learnable programming. (2012). http://worrydream.com/LearnableProgramming/
32. Bret Victor. 2014. Seeing Spaces. (2014). http://worrydream.com/SeeingSpaces/
33. W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.
34. Te-Yen Wu, Hao-Ping Shen, Yu-Chian Wu, Yu-An Chen, Pin-Sung Ku, Ming-Wei Hsu, Jun-You Li, Yu-Chih Lin, and Mike Y. Chen. 2017a. CurrentViz: Sensing and Visualizing Electric Current Flows of Breadboarded Circuits. In *Proceedings of the 30th Annual Symposium on User Interface Software and Technology (UIST 2017)*. Quebec City, Canada.
35. Te-Yen Wu, Bryan Wang, Jiun-Yu Lee, Hao-Ping Shen, Yu-Chian Wu, Yu-An Chen, Pin-sung Ku, MING-WEI HSU, Yu-Chih Lin, and Mike Y. Chen. 2017b. CircuitSense: Automatic Sensing of Physical Circuits and Generation of Virtual Circuits to Support Software Tools. In *Proceedings of the 30th Annual Symposium on User Interface Software and Technology (UIST 2017)*. ACM, Quebec City, QC, Canada. DOI: http://dx.doi.org/10.1145/3126594.3126634
36. Andreas Zeller. 2009. *Why programs fail: a guide to systematic debugging*. Elsevier.