Bifröst

An Interface for Visualizing and Debugging the Behavior of Embedded Systems

Will McGrath, Jeremy Warner, Björn Hartmann, et al.

Introduction

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. Bifröst is 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 detect low-level hardware configuration and protocol issues, while user-authored checks test particular application semantics.

Motivation

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

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.

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

System Overview

Bifröst consists of hardware and software to make it easy to simultaneously capture the programmatic and electrical behavior of an embedded system, an extensible system for creating and running "checkers", test functions that have access to the information captured by the system, and an IDE that allows a user to easily navigate and analyze the captured trace and checker results. In order to capture an interactive device's electrical activity, Bifröst makes use of a custom Arduino shield that breaks out all the Arduino's commonly-used pins out to headers.. The signals are captured using a 16 channel Logic Analyzer produced by Saleae.

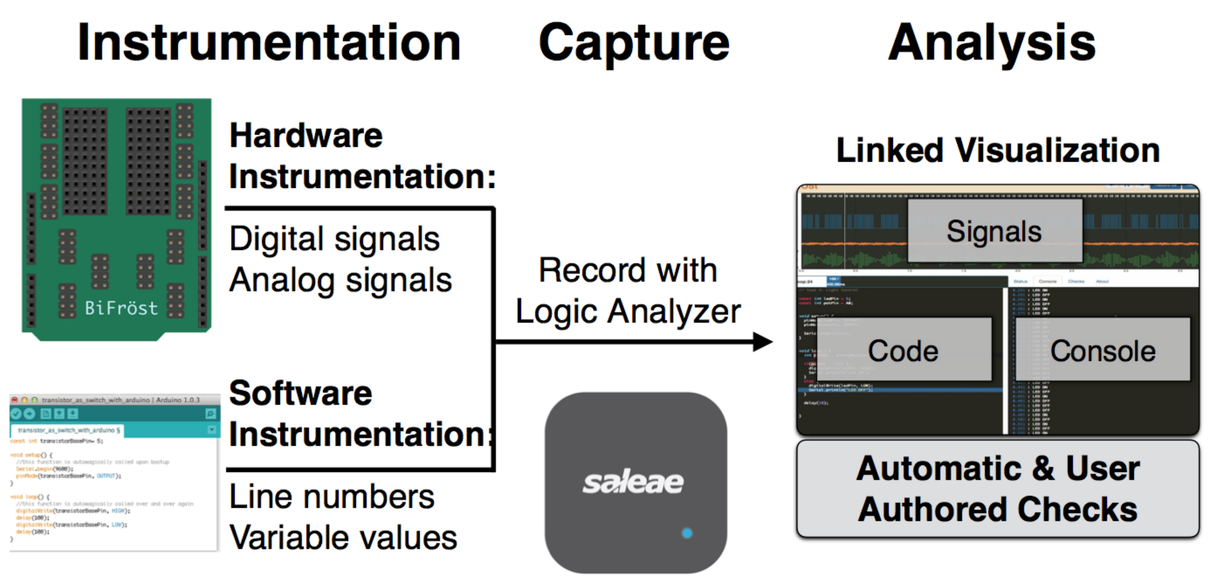

Figure 1: An overview of the hardware and software components of Bifröst's architecture.

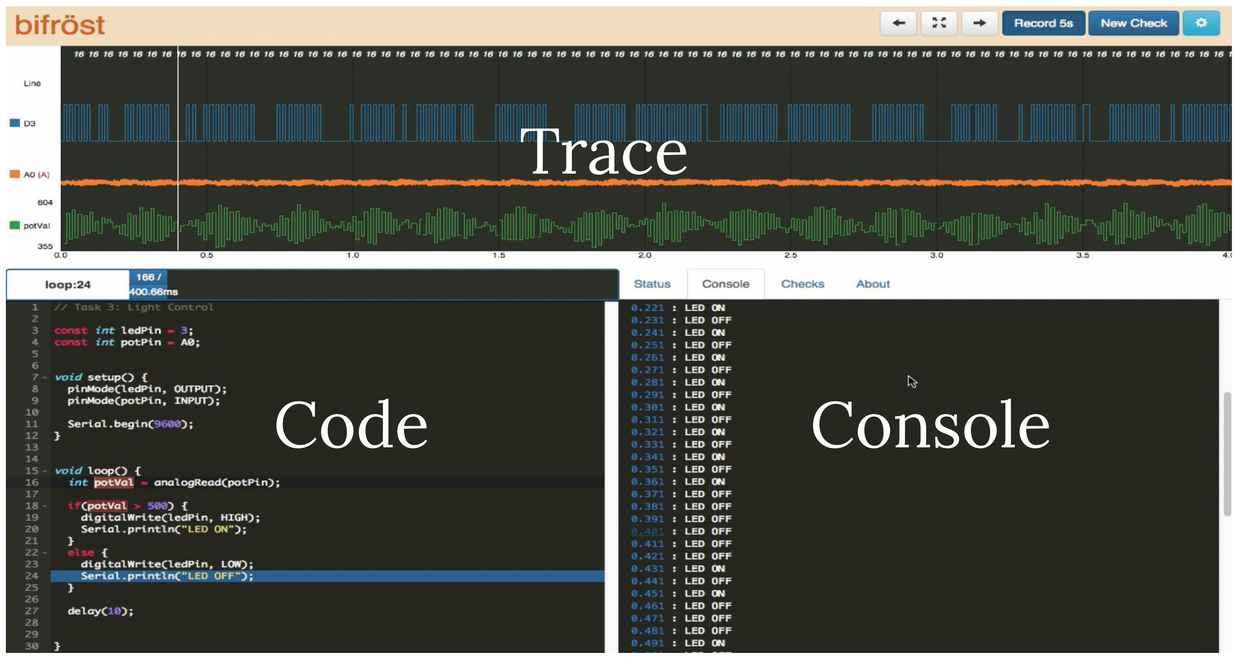

Figure 2: An overview of Bifröst's UI: an integrated code editor, serial console, and trace visualization.

Checks

After each capture, Bifröst loads and runs its library of checker test functions against the data. Checkers have complete access to the device's software and its behavior (i.e. the text of a given line and when it ran) as well as all the electrical behavior of the microprocessor's pins. For example, if a user has code to set a value to a 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.

Bifröst GUI

A central idea underlying Bifröst is that viewing data from 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. The editor shows the source code of program that generated the trace and links the selections on both views. The visualizations let users test hypotheses by comparing expected values at different points in their system directly to discover any issues. Users can easily step forwards and backwards through executed lines of code with arrow keys. If the user sees that a check flagged an output changing incorrectly, they can click on the graph and view what code line caused the output to occur as well as easily jump backwards to the lines and pin transitions preceding the change.

Automatic Instrumentation

In Bifröst, we instrument and modify a user's program to make internal variable and execution state available while the system is running. With this we let users focus on the behavior of own program, not system code or instrumentation.

```
void readRegister(char registerAddress, int numBytes, signed char * values){
  //Since we're performing a read operation, the most significant bit of the register address should be set.
  Serial.write(45);
  char address = 0x80 | registerAddress;
  //If we're doing a multi-byte read, bit 6 needs to be set as well.
  Serial.write(47);
  if(numBytes > 1)address = address | 0x40;

  //Set the Chip select pin low to start an SPI packet.
  Serial.write(50);
  digitalWrite(CS, LOW);
  //Transfer the starting register address that needs to be read.
  Serial.write(52);
  SPI.transfer(address);
  //Continue to read registers until we've read the number specified, storing the results to the input buffer.
  for(int i=0; i<numBytes; i++){
    Serial.write(55);
    values[i] = SPI.transfer(0x00);
  }
```

Figure 3: A function automatically instrumented by Bifröst to output user code line execution events.