A Software Debugger for E-textiles and Arduino Microcontrollers

Michael Schneider
Michael.J.Schneider@colorado.edu
University of Colorado, Boulder
Boulder, Colorado

Chris Hill
Christian.N.Hill@colorado.edu
University of Colorado, Boulder
Boulder, Colorado

Ann Eisenberg
eisenbea@colorado.edu
University of Colorado, Boulder
Boulder, Colorado

Mark Gross
mdgross@colorado.edu
University of Colorado, Boulder
Boulder, Colorado

Arielle Blum
amblum@colorado.edu
University of Colorado, Boulder
Boulder, Colorado

ABSTRACT

Today's STEM classrooms have expanded the domain of computer science education from a basic two-toned terminal screen to now include helpful Integrated Development Environments(IDE) (BlueJ, Eclipse), block-based programming (MIT Scratch, Greenfoot), and even physical computing with embedded systems (Arduino, LEGO Mindstorms). But no matter which environment a student starts programming in, all students will eventually need help in finding and fixing bugs in their code. While the helpful IDE's have debugger tools built in (breakpoints for pausing your program, ways to view/modify variable values, and "stepping" through code execution), in many of the other programming environments, students are limited to using print statements to try and "see" what is happening inside their program.

Most students who learn to write code for Arduino microcontrollers will start within the Arduino IDE, but the official Arduino IDE does not currently provide any debugging tools. Instead, a student would have to move on to a professional IDE such as Atmel Studio or acquire a hardware debugger in order to add breakpoints or view their program's variables. But each of these options has a steep learning curve, additional costs, and can require complex configurations. Based on research of student debugging practices[3, 7] and our own classroom observations, we have developed an Arduino software library, called Arduino Debugger, which provides some of these debugging tools (ex. breakpoints) while staying within the official Arduino IDE. This work continues a previous library, (redacted), which focused on features specific to e-textiles development boards. The Arduino Debugger library has been modified to support not only e-textile boards (LilyPad, Adafruit Circuit Playground) but most AVR and ARM based Arduino boards. We are also in the process of testing a set of Debugging Code Templates to see how they might increase student adoption of debugging tools.

Permission to make digital or hard copies of all or part 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 components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FabLearn '20, October 10–11, 2020, New York, NY, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7543-6/20/04...$15.00
https://doi.org/10.1145/3386201.3386222

CCS CONCEPTS

• **Social and professional topics** → **Computing education**; • **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Debugging, Arduino, Serial Monitor, Computer Science Education

ACM Reference Format:
Michael Schneider, Chris Hill, Ann Eisenberg, Mark Gross, and Arielle Blum. 2020. A Software Debugger for E-textiles and Arduino Microcontrollers. In *FabLearn 2020 - 9th Annual Conference on Maker Education (FabLearn '20), October 10–11, 2020, New York, NY, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3386201.3386222

1 PROBLEM & MOTIVATION

Current tools for debugging embedded systems are intended for experienced programmers and engineers. These tools consist of IDEs which provide code breakpoints, code tracing (aka "stepping through code"), and viewing or modifying variable values. Engineers also have access to hardware debuggers, physical devices which can connect to a development board and provide more detailed information about the embedded system, such as register values and memory. High school students do not usually use the professional tools when working with Arduino devices. Instead, they usually start with the offical Arduino IDE, which provides a basic text editor, library manager, serial monitor, and can compile and upload code onto the Arduino microcontroller. This basic IDE works well to start with but lacks the necessary debugging tools for students as they expand their knowledge and skills.

It is possible for instructors to provide professional debugging tools such as Atmel Studio [10], Visual Micro (plug-in for Visual Studio) [8], or a hardware debugger like J-Link [13] or Atmel ICE [9]. These tools would allow students to create breakpoints and view system information such as variables, registers, and memory. Unfortunately these tools are not always ideal for K-12 classrooms or introductory undergraduate classes because of expensive licensing, complex configurations, and a learning curve that may not fit within shorter lesson plans (a few weeks vs a whole semester of programming). There needs to be an intermediary that helps beginners to gain basic debugging skills for fixing their novice code and then prepares them to move toward more advance tools when they are ready. The Arduino Debugger library aims to be this intermediary by providing methods which emulate debugger tools,

breakpoints and variable "watching", while keeping the students within the Arduino IDE and not requiring additional hardware. To help encourage students to use the Arduino Debugger, we have paired it with a series of Debugging Coding Templates which provide guidelines for how students can place breakpoints in different types of coding structures (loops, if/else, methods) which we believe can help encourage students to regularly use debugging tools.

2 BACKGROUND & RELATED WORK

The lack of beginner friendly debugger tools for Arduino is a known concern [2, 15] and tools have been created that provide debugging features via plug-ins to Visual Studio or Atmel Studio [2, 8, 14]. These plug-ins allow the student to use the IDE's debugger tools with the Arduino core library they are familiar with and do not require a hardware debugger (on supported Arduino boards). This pairing of the familiar Arduino libraries with an advance IDE, makes these plug-ins a great option for more advanced undergraduates and professional engineers. But these tools still require costly licenses and more intensive setup than the Arduino IDE. Also, these plug-ins provide features that are not needed by or accessible to novice programmers, such as viewing registers and memory. Instead of providing a complex toolbox to work with, Yago et al.[15] created a customized version of the Arduino IDE and its drivers to provide breakpoints and allow students to step through their code . This is similar to our approach except Yago et al. did not allow for students to view their variables and required students to work with a customized Arduino IDE, which lacks long-term support.

Our tool was designed to avoid the shortcomings of the professional tools (cost and complexity) and other research tools (GDB plugin [2] or modified Arduino IDE [15]). Instead, our tool has been designed to work with the standard Arduino IDE and support as many Arduino-based devices as possible.

3 APPROACH & UNIQUENESS

The design of the Arduino Debugger is based not only on our research of past works, but also on our classroom observations of novice programmers. Classroom observations were taken in an undergraduate embedded systems class where students created projects with an Arduino-based device. Students were not required to have coding experience for the course, with many of the students learning to program for the first time. As the students created their projects, we offered assistance when they ran into problems. In our observations, we noticed a trend in the project errors. When an error arose, students would often start with the assumption that the error was hardware related (wire is not connected or there is not enough power running to the connected devices). In our discussions with the students, many of the students had a false assumption that if the code compiled then it must be working correctly. This misconception often led the students to painstakingly take their projects apart and re-wire them back together. These unnecessary teardowns could have been avoided if the students had more thoroughly tested their code.

Even though students considered the hardware to be a likely starting point for system errors, nearly all of the errors we assisted in debugging ended up being software related. For this reason we have created a software library that focuses on assisting students with known issues of debugging. These issues are as follows:

- **Error Location**
  The process of actually locating a bug in their program is considered the most difficult part of debugging. [4, 6] Our tool helps students to quickly narrow down potential bug sources.
- **Adding New Errors**
  When debugging, students will often add new errors that in turn require more debugging [12]. The Arduino Library's code is meant to be minimally intrusive with very few lines of new code required, thereby limiting the potential of adding new bugs to the student's program.
- **Common Errors**
  Some of the most common bugs students create in their programs are assignment errors, iteration errors, and array errors[5]. By observing these errors in our students' code, we have adapted the Arduino Debugger to provide helpful information so that students can recognize when one of these errors is occurring.

3.1 Library Features

The three main features of the debug library are as follows:

- **Breakpoints & Pausing**
  A call to the breakpoint() method allows students to pause a running application and view program information.
- **Hardware Pins**
  Students view the current state of hardware pins (digital & analog) and also update the state of the digital pins.
- **Variable Watch List**
  Students can add program variables to "watch list" which allows them to view and update a variable's value.

Additional features are provided by the Arduino Debugger library, but they cannot all be detailed here. To review the library's code, documentation, and guides, please follow the link to our Instructable at https://www.instructables.com/id/Arduino-Debuger-Getting-Started/.

Example Code 1: Buggy Student Sketch

```
1  #include <Debugger.h>
2  //Standard configuration for Arduino Uno.
3  ArduinoDebugger debugger =
   ↪  Debugger::initialize(false, true, false);
4  void setup() {
5    Serial.begin(9600);
6    while(!Serial) {}
7  }
8  void loop() {
9   /*
10   Intended program behavior:
11     The program is meant to turn on a set of LEDs
   ↪  which are connected via the pin #'s in the
   ↪  pins array.
12   Actual Behavior:
13     The program only turns on every other LED
```

```
14      Error:
15        The for loop contains an update to the for
   ↪    loop counter i, causing it to update by 2,
   ↪    instead of 1.
16     */
17      byte pins[] = {3,4,5,6};//Pins connected to LEDs
18      byte i = 0;//for loop counter
19      //add variables to the watch list
20      debugger.add(&i, BYTE, "i");
21      debugger.add(&pins, BYTE_ARRAY, "pins_4");
22      debugger.breakpoint("Before For Loop");
23      for(i; i < sizeof(pins)/sizeof(pins[0]); i++){
24        debugger.breakpoint("Start For Loop");
25       digitalWrite(i, HIGH);//Turn on the chosen LED
26        i++;//Error!
27        debugger.breakpoint("End For Loop");
28      }
29    }
```

3.1.1 *Breakpoints.*

In a standard IDE, a breakpoint allows a programmer to pause a running application at a specific line of code. This can be done to pause the program once a desired condition has been met (for example, inside a conditional statement) and/or to evaluate the current values of variables. The Arduino Debugger's breakpoint() method offers similar functionality. When called, it will pause the running Arduino program and present a menu via the Serial Monitor, as seen in Figure 1. The menu allows a student to choose between viewing and updating variables in the watch list or the microcontroller's hardware pins. In Example Code 1, two breakpoints have been added on lines 24 & 29. In each, an optional character array has been passed which defines the name for the breakpoint. The name helps to identify which breakpoint has been reached, in case multiple breakpoints have been added to a program.

3.1.2 *Hardware Pins.*

Selecting "Hardware Pins" will display the current state of the board's digital & analog pins, as seen in Figure 2. The state of a pin refers to the current value retrieved via digitalRead() or analogRead(). For Arduino, digital pins have only two states (HIGH/LOW), but the library expands this to three:

HIGH This means the pin is ON and providing POWER OUT
LOW This means the pin is OFF
HIGH(Power In) This means the pin is receiving POWER IN from an external power source

After displaying the digital & analog pin states, the Arduino Debugger will allow the student to update a digital pin to either HIGH or LOW.

3.1.3 *Variables.*

Selecting "Variables" will display the current value of the variables that have been added to the watch list and allow the student to update any variable's value. To add a variable to the watch list, a student will use the add() method which accepts the variable's memory address, data type, and the variable's name. Examples of using the add method can be found in code example 1 on lines 22 & 23. Currently, the Arduino Debugger only supports the data primitives byte, int, long, float, char, bool, and arrays of those data primitives.

3.2 Debugging Code Templates

The Arduino Debugger can help students analyze their code, but only if they use it. Murphy et. al. found that even though their students had access to an IDE's debugger, only a few used it. Worse yet, not many students used print statements [11]. We found similar behavior in our own observations, where students did not regularly use print statements to help evaluate their buggy code. For this reason, we created the code templates to help students see potential ways of using the debugger in their programs.

Figure 1: Main menu

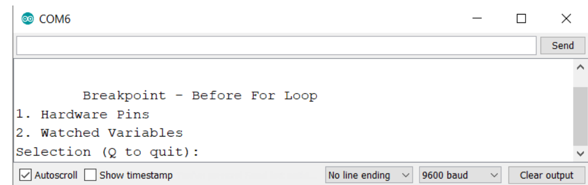


Figure 2: Hardware Pins

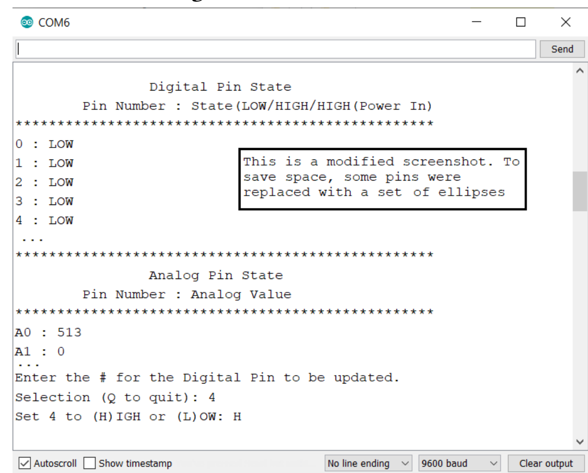
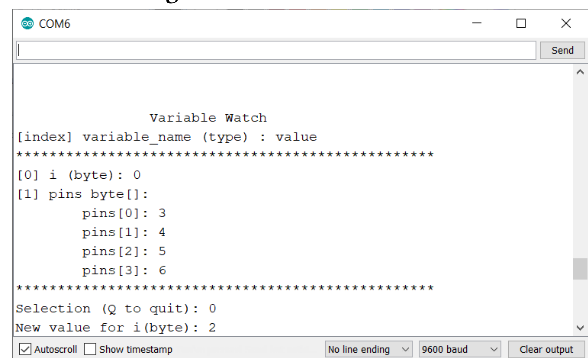


Figure 3: Variable Watch List



The debugging code templates have been designed to help overcome common issues when students are attempting to debug their programs:

- **Not enough detail**
  When using print statements for debugging, students will leave out important information (not display a necessary variable's value) or even create more confusion (using the same print message within if-else statements) [11].
- **Under utilization of debugger tools**
  Whether the tool is an IDE-enabled debugger (breakpoints, variable watch list) or print statements, only a minority of students utilize these tools to assist in debugging. [11]
- **Sporadic debugging behavior**
  Students will often jump around their code in a haphazard fashion trying to find the source of a bug without trying to methodically test specific regions of their code. [1, 11, 16]

Our initial tests of the coding templates focused on showing students where they should place breakpoints in and around different coding structures. The coding structures we are testing with are loops, conditional statements, and methods. Besides breakpoint locations, the coding templates remind students of important variables they should add to the variable watch list. Please note that these coding templates are not meant to be all encompassing, but should be viewed as "general rules". The intent is to first encourage use of breakpoints and adding variables to the watch list. Over time, we hope to see how students start to expand beyond these starter coding templates.

Example Code 2: Coding Template - If/Else

```
1  //Add variables for conditional statements
2  //debugger.add(&var; TYPE; "var");
3  if(condition){
4    debugger.breakpoint("Condition 1 True");
5    //Your code here
6  }
7  else if(other_condition){
8    debugger.breakpoint("Condition 2 True");
9    //Your code here
10 }
11 else{
12   debugger.breakpoint("All Conditions False");
13   //Your code here
14 }
```

4 DISCUSSION

The pairing of the Arduino Debugger with the Debugging Code Templates endeavors to encourage students to regularly use debugging tools. During the next stage of classroom observations, we will gather data on how students follow the code templates, modify them, and if any common debugging strategies can be found within the students' code.

We do not believe the Debugging Code Templates encourage students to move away from the depth-first bug search patterns of novices to the breadth-first bug search patterns of experts [16]. The code templates only show potential areas to place breakpoints and remind students of important variables they should add to their watch lists. For students who have short programs (similar to example code 1), the code templates should be fine. We hope our observations show how the students use the code templates in progressively larger programs. Do the students use the debugger breakpoints in a haphazard way, do they methodically place them inside EVERY code block, or do they start to use the breakpoints in new ways that we hadn't intended or thought of?

We hope our observations of students will help us better understand:

- How does student use of the Arduino Debugger differ from our intended vision?
- Does use of the Arduino Debugger and the Debugger Code Templates change over time?
- Is there a common type of bug(s) that the students use the Arduino Debugger to find?
- How can the Arduino Debugger and Debugger Code Templates be modified to improve students debugging techniques?

5 ACKNOWLEDGEMENTS

This work was supported by grant #1742081 from the National Science Foundation. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation or the University of Colorado, Boulder. Special thanks to the PI's Ann Eisenberg and Mark Gross, my adviser Tamara Sumner, as well as the other members of the Craft Tech Lab - Christian Hill, Arielle Blum, and Rona Sadan.

REFERENCES

[1] Ryan Chmiel and Michael C Loui. 2004. Debugging: from novice to expert. In *ACM SIGCSE Bulletin*, Vol. 36. ACM, 17–21.
[2] Jan Dolinay, Petr Dostálek, and Vladimír Vašek. 2016. Arduino Debugger. *IEEE Embedded Systems Letters* 8, 4 (2016), 85–88.
[3] Deborah A Fields, Kristin A Searle, Yasmin B Kafai, and Hannah S Min. 2012. Debuggems to assess student learning in e-textiles. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. 699–699.
[4] Sue Fitzgerald, Renée McCauley, Brian Hanks, Laurie Murphy, Beth Simon, and Carol Zander. 2009. Debugging from the student perspective. *IEEE Transactions on Education* 53, 3 (2009), 390–396.
[5] John D Gould. 1975. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies* 7, 2 (1975), 151–182.
[6] Irvin R Katz and John R Anderson. 1987. Debugging: An analysis of bug-location strategies. *Human-Computer Interaction* 3, 4 (1987), 351–399.
[7] Debora Lui, Emma Anderson, Yasmin B Kafai, and Gayithri Jayathirtha. 2017. Learning by fixing and designing problems: A reconstruction kit for debugging e-textiles. In *Proceedings of the 7th Annual Conference on Creativity and Fabrication in Education*. 1–8.
[8] Visual Micro. 2019. *Arduino for Visual Studio*. Retrieved January 5, 2020 from https://www.visualmicro.com/
[9] Microchip. 2020. *Atmel-ICE*. Retrieved January 5, 2020 from https://www.microchip.com/DevelopmentTools/ProductDetails/ATATMEL-ICE
[10] Microchip. 2020. *Atmel Studio 7*. Retrieved January 5, 2020 from https://www.microchip.com/mplab/avr-support/atmel-studio-7
[11] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: the good, the bad, and the quirky–a qualitative analysis of novices' strategies. In *ACM SIGCSE Bulletin*, Vol. 40. ACM, 163–167.
[12] Murthi Nanja and Curtis R Cook. 1987. An analysis of the on-line debugging process. In *Empirical studies of programmers: second workshop*. Ablex Publishing Corp., 172–184.
[13] Segger. 2020. *J-Link Debug Probes*. Retrieved January 5, 2020 from https://www.segger.com/products/debug-probes/j-link/
[14] SysProgs. 2020. *VisualGDB*. Retrieved January 5, 2020 from https://visualgdb.com/
[15] Yago Torroja, Alejandro López, Jorge Portilla, and Teresa Riesgo. 2015. A serial port based debugging tool to improve learning with arduino. In *2015 Conference on Design of Circuits and Integrated Systems (DCIS)*. IEEE, 1–4.
[16] Iris Vessey. 1985. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies* 23, 5 (1985), 459–494.