Inline

EICS

EICS 2024

Inline Visualization and Manipulation of Real-Time Hardware Log for Supporting Debugging of Embedded Programs

Andrea Bianchi

Department of Industrial Design
& School of Computing,
KAIST

Zhi Lin Yap

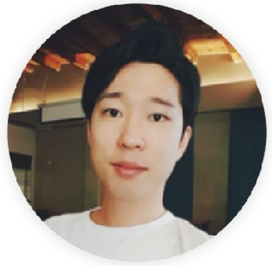
School of Computing,
KAIST

Punn Lertjaturaphat

Department of Industrial Design,
KAIST

Austin Z. Henley

Microsoft



Kongpyung (Justin) Moon

Department of Industrial Design,
KAIST

Yoonji Kim

College of Art & Technology,
Chung-Ang University

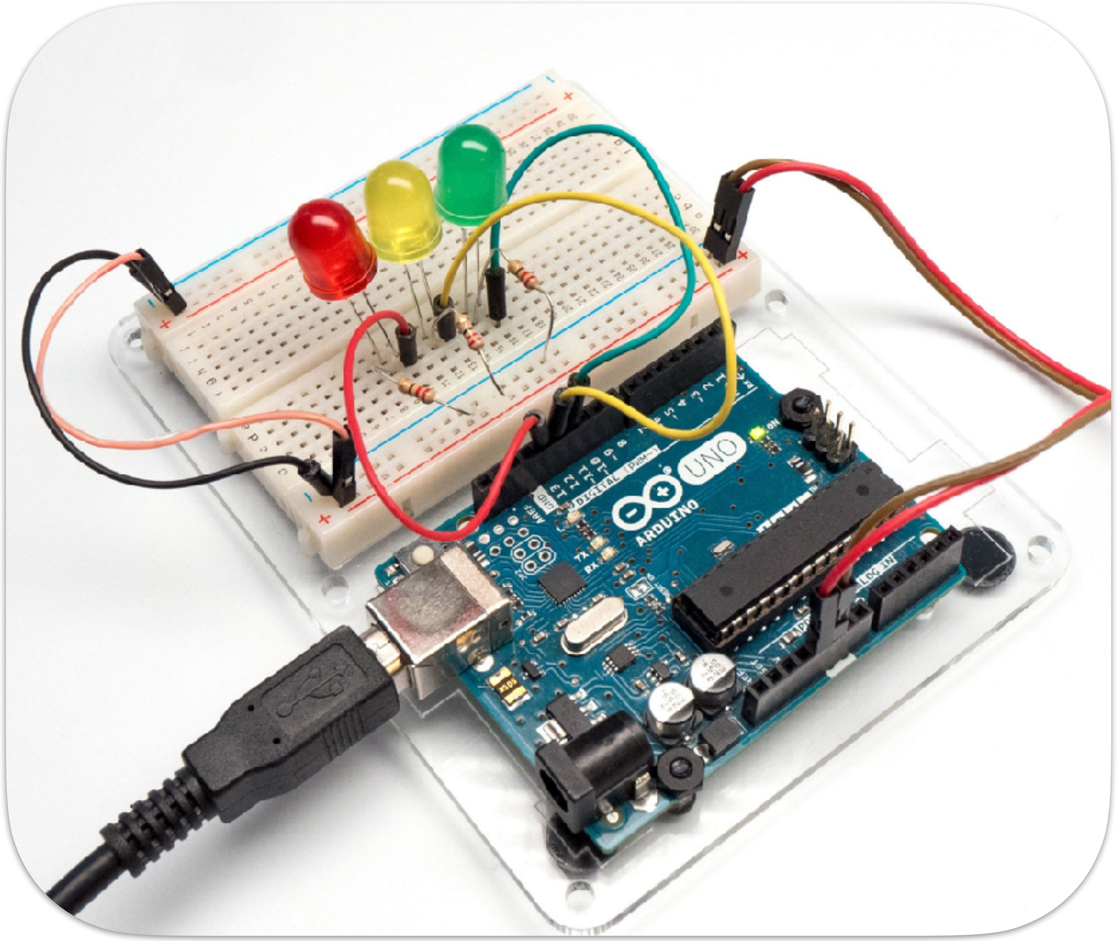
contact
andrea@kaist.ac.kr

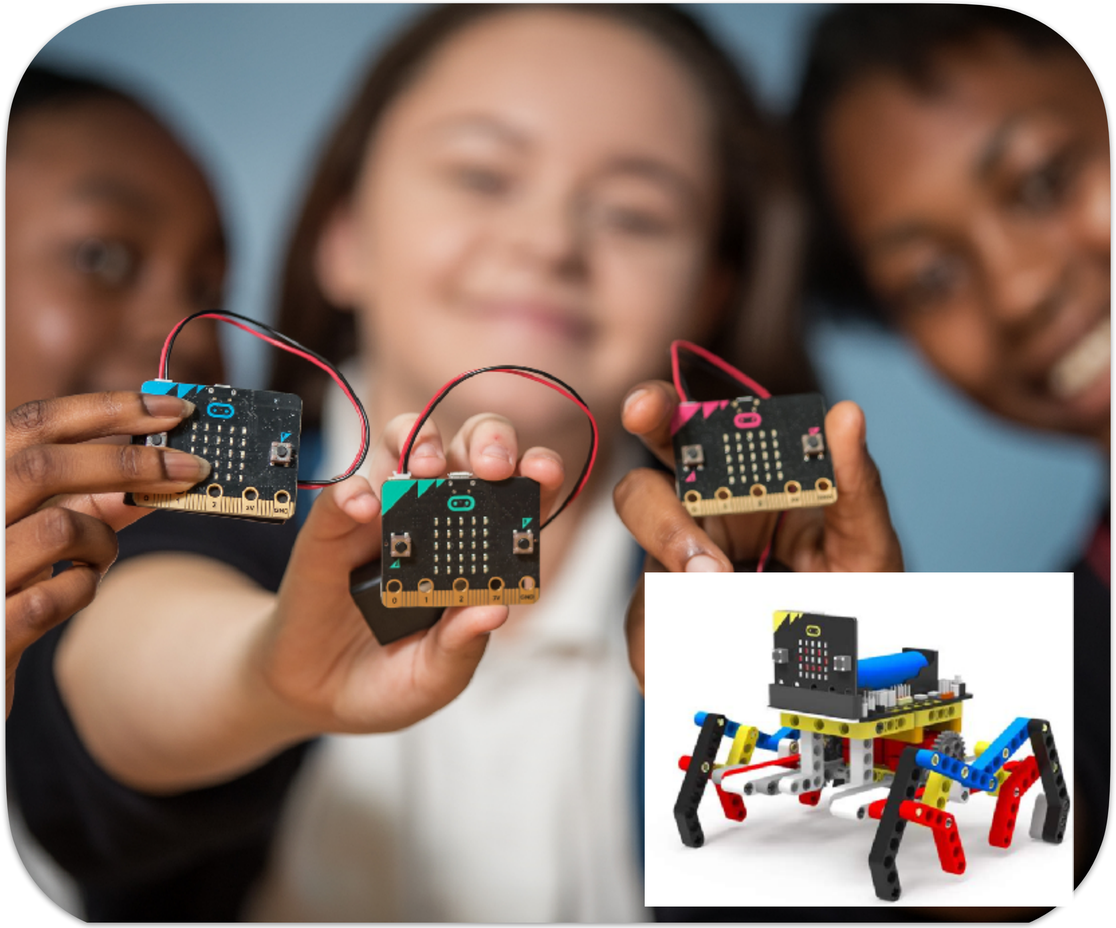
KAIST

CAU 중앙대학교

Microsoft

Physical Computing and STEM education

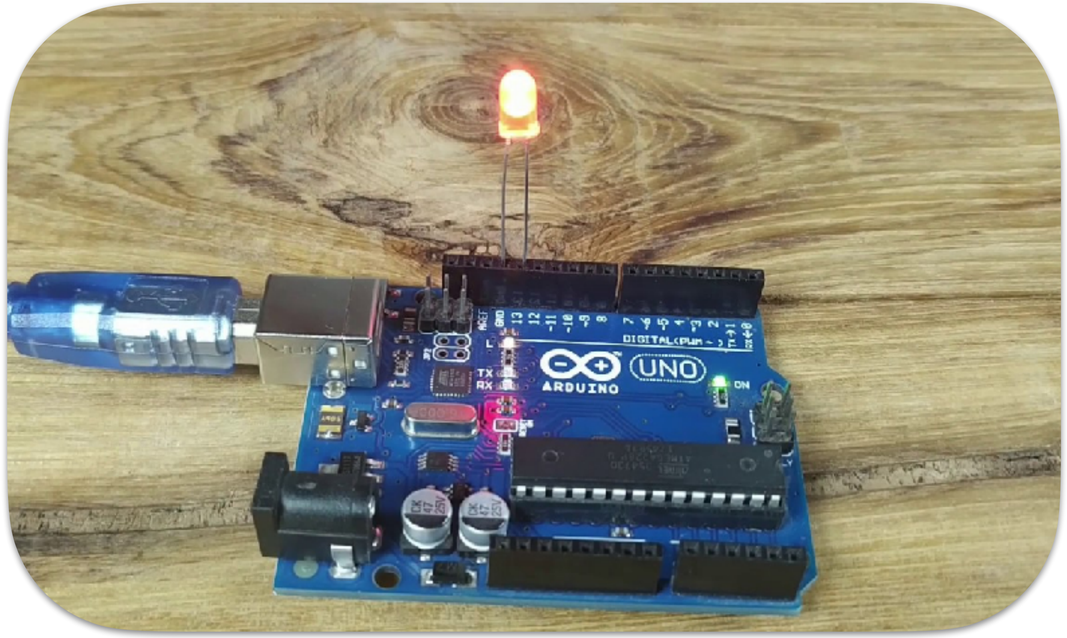




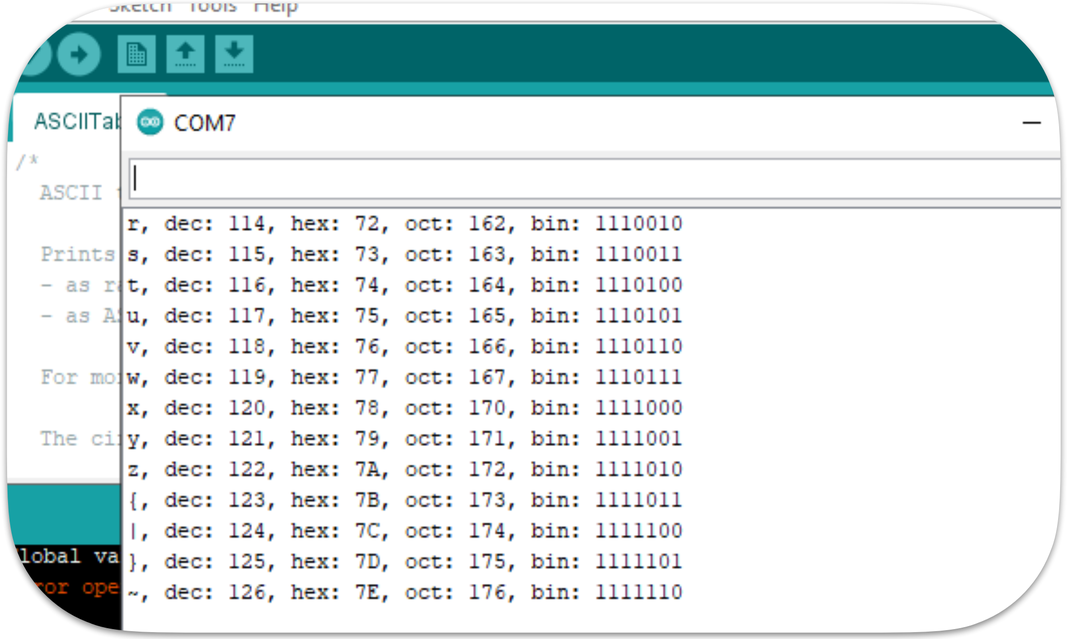
- Widespread popularization of user-friendly platforms like Arduino & co.
- Computational thinking with STEM though embedded programming

Challenges with Physical Computing

- Mistakenly perceived as a *simple* activity
- Previous study [Booth 2016] report 54% of unsolved problems at the interplay between software and hardware
- Information barriers [Ko 2004]: interpreting the program internal state from *externally observable information* (**debugging**)



Blinking LED as ***physical proxy***

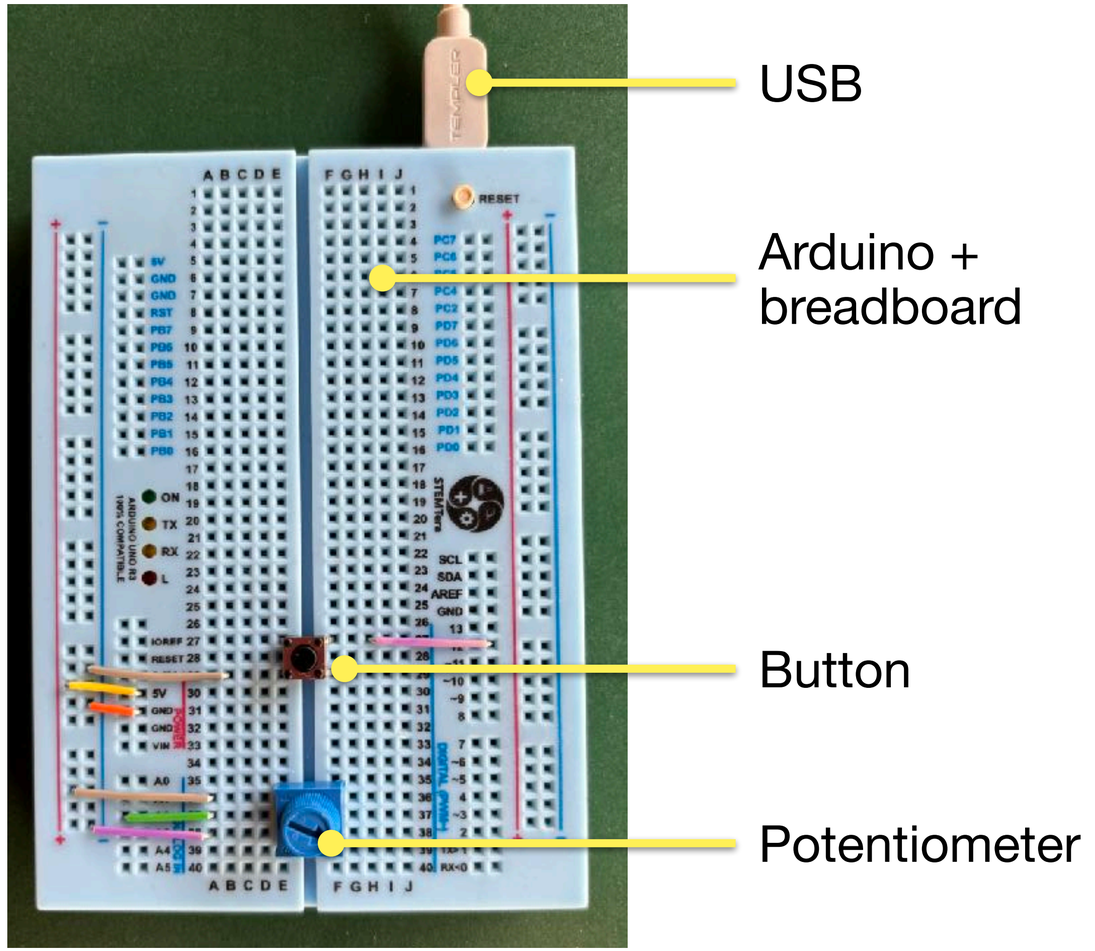


Printing statements for ***console logs***

How to make sense of hardware debug logs?

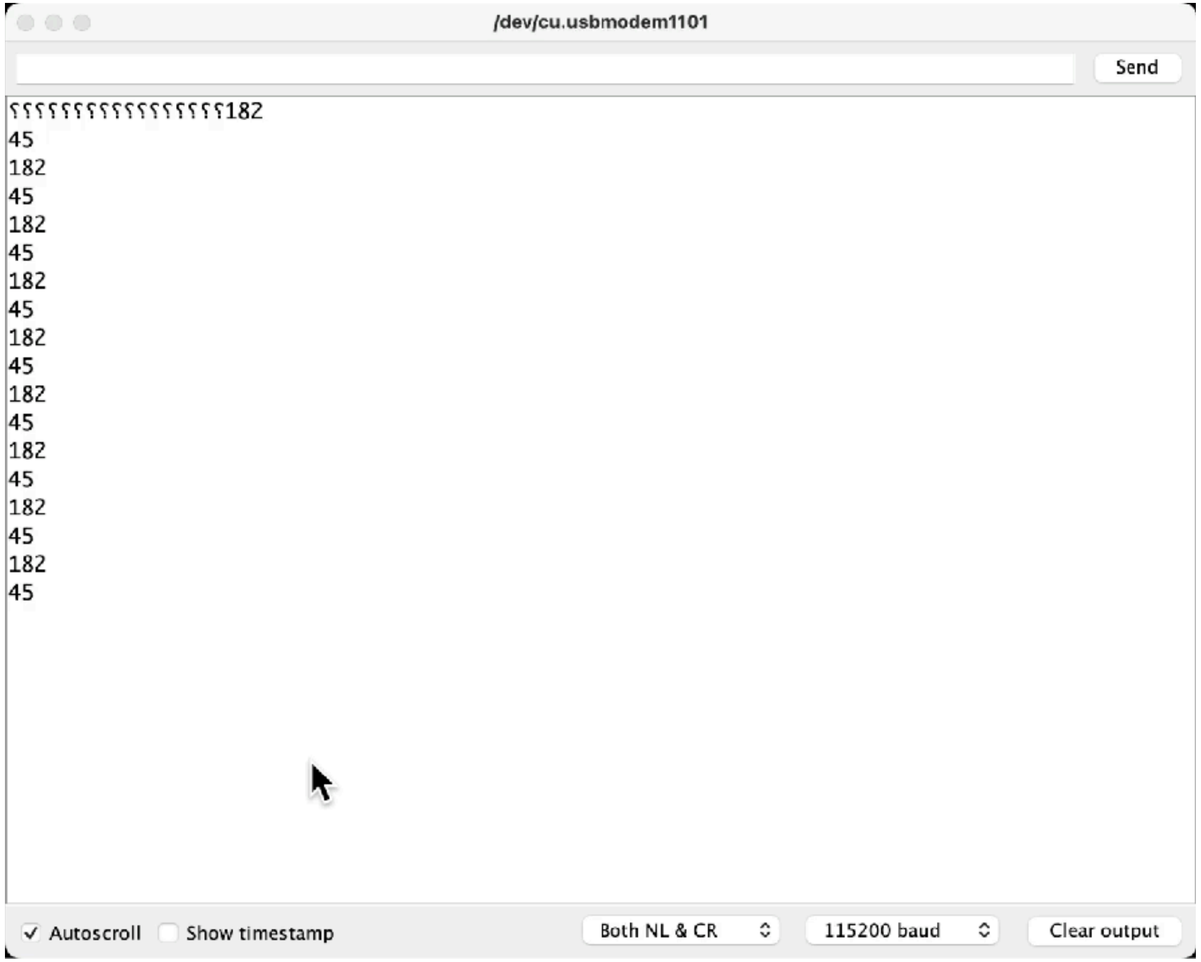
What are debugging logs anyway?

```
void setup(){
    Serial.begin(115200);
}

void loop(){

    // Reading an analog value
    int val = analogRead(A2);
    Serial.println(val);

    // Mapping the value to a suitable
    // duty-cycle for PWM
    int mapped = map(val, 0, 1024, 0, 255);
    Serial.println(mapped);

    analogWrite(3, mapped);

    delay(100);
}
```



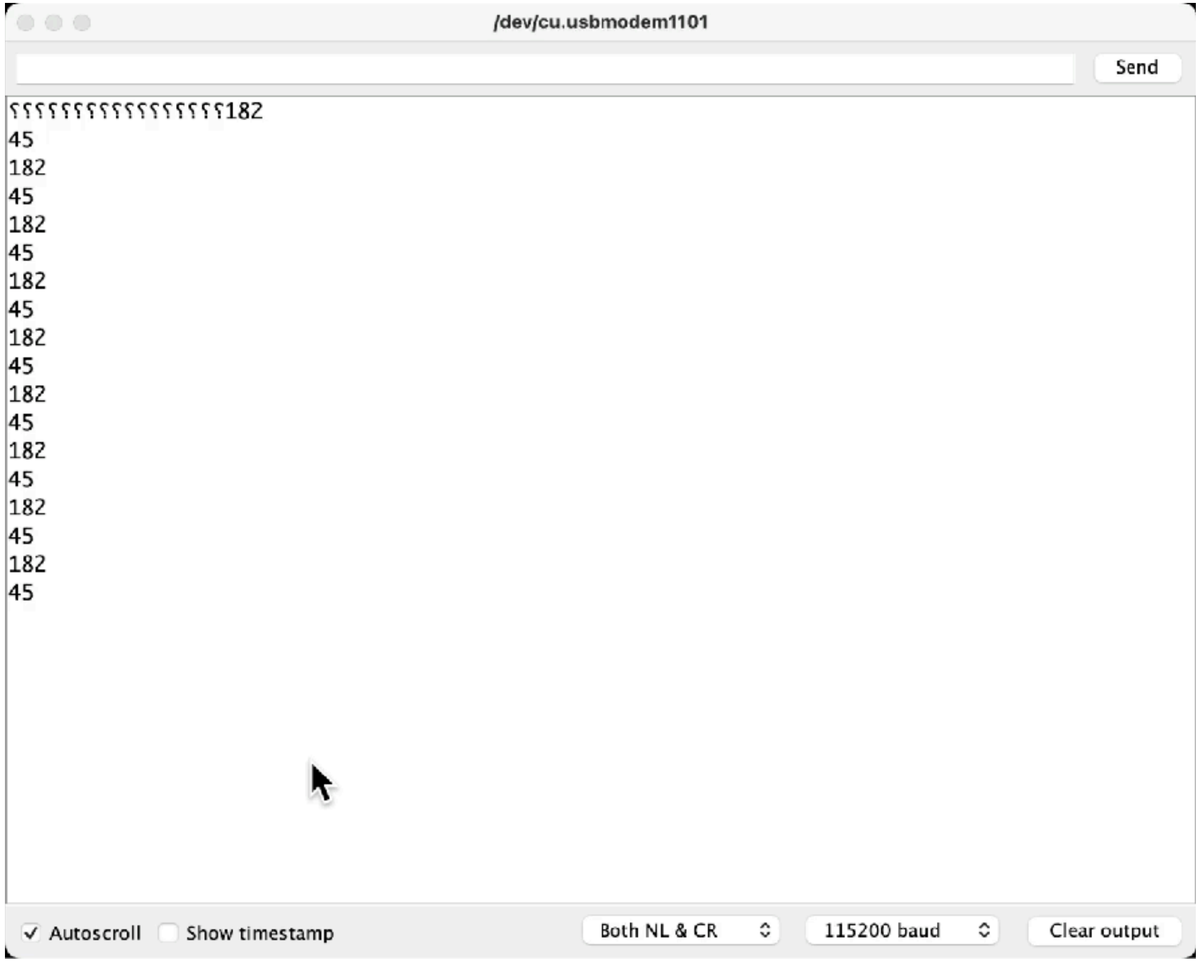
What are debugging logs anyway?

```
void setup(){
    Serial.begin(115200);
}

void loop(){

    // Reading an analog value
    int val = analogRead(A2);
    Serial.println(val);

    // Mapping the value to a suitable
    // duty-cycle for PWM
    int mapped = map(val, 0, 1024, 0, 255);
    Serial.println(mapped);

    analogWrite(3, mapped);

    delay(100);
}
```



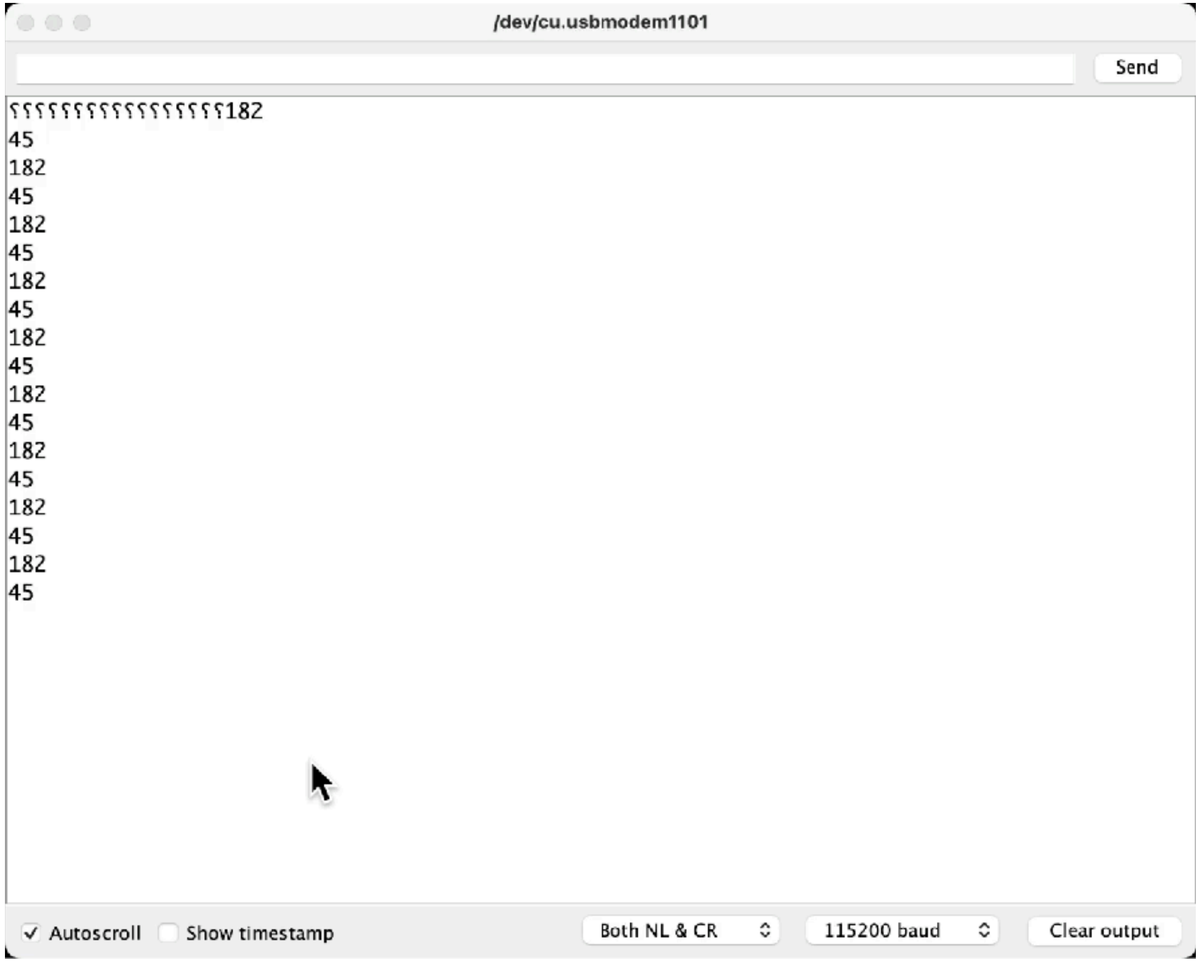
What are debugging logs anyway?

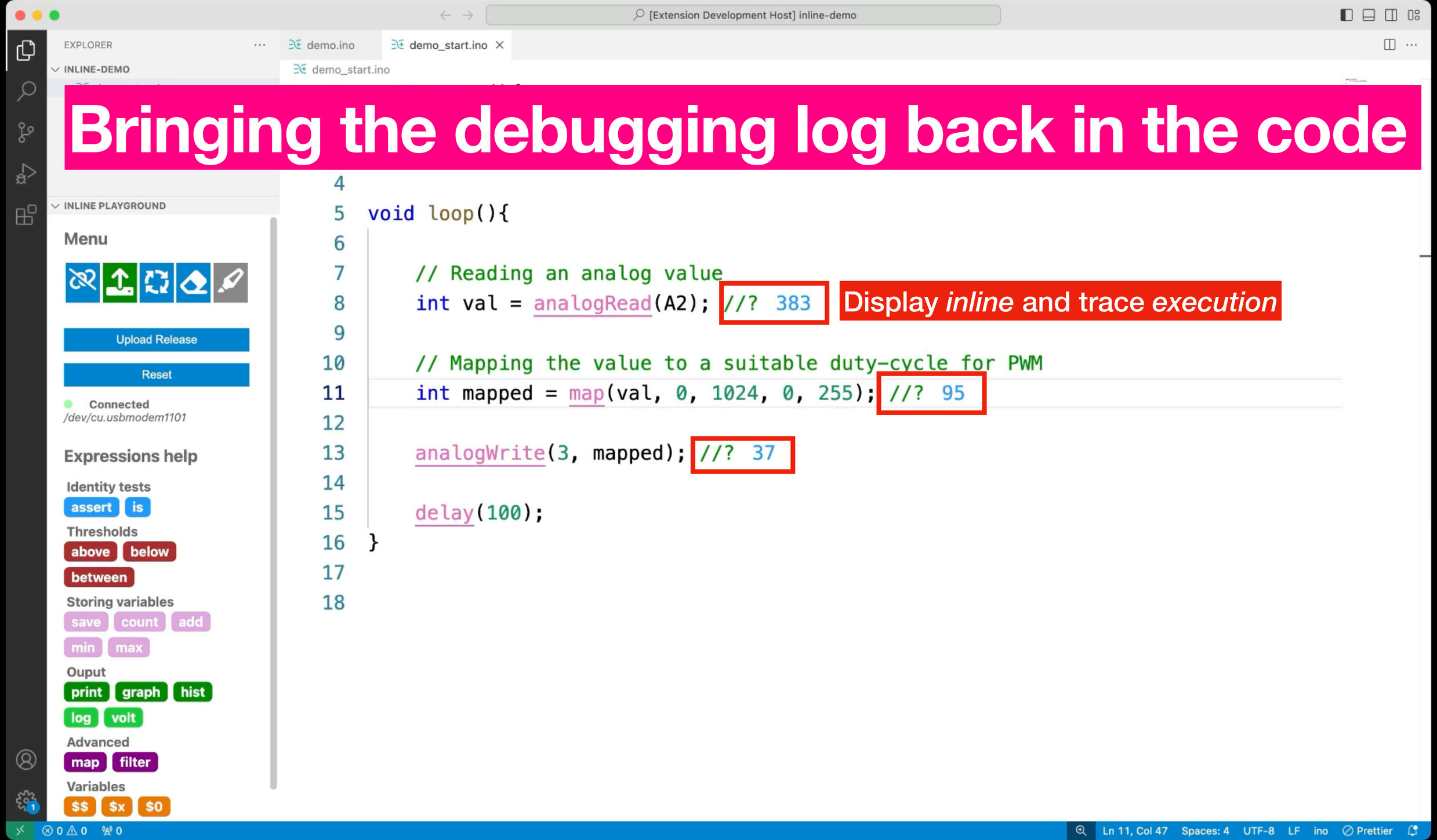
```
void setup(){
    Serial.begin(115200);
}

void loop(){

    // Reading an analog value
    int val = analogRead(A2);
    Serial.println(val);

    // Mapping the value to a suitable
    // duty-cycle for PWM
    int mapped = map(val, 0, 1024, 0, 255);
    Serial.println(mapped);

    analogWrite(3, mapped);

    delay(100);
}
```

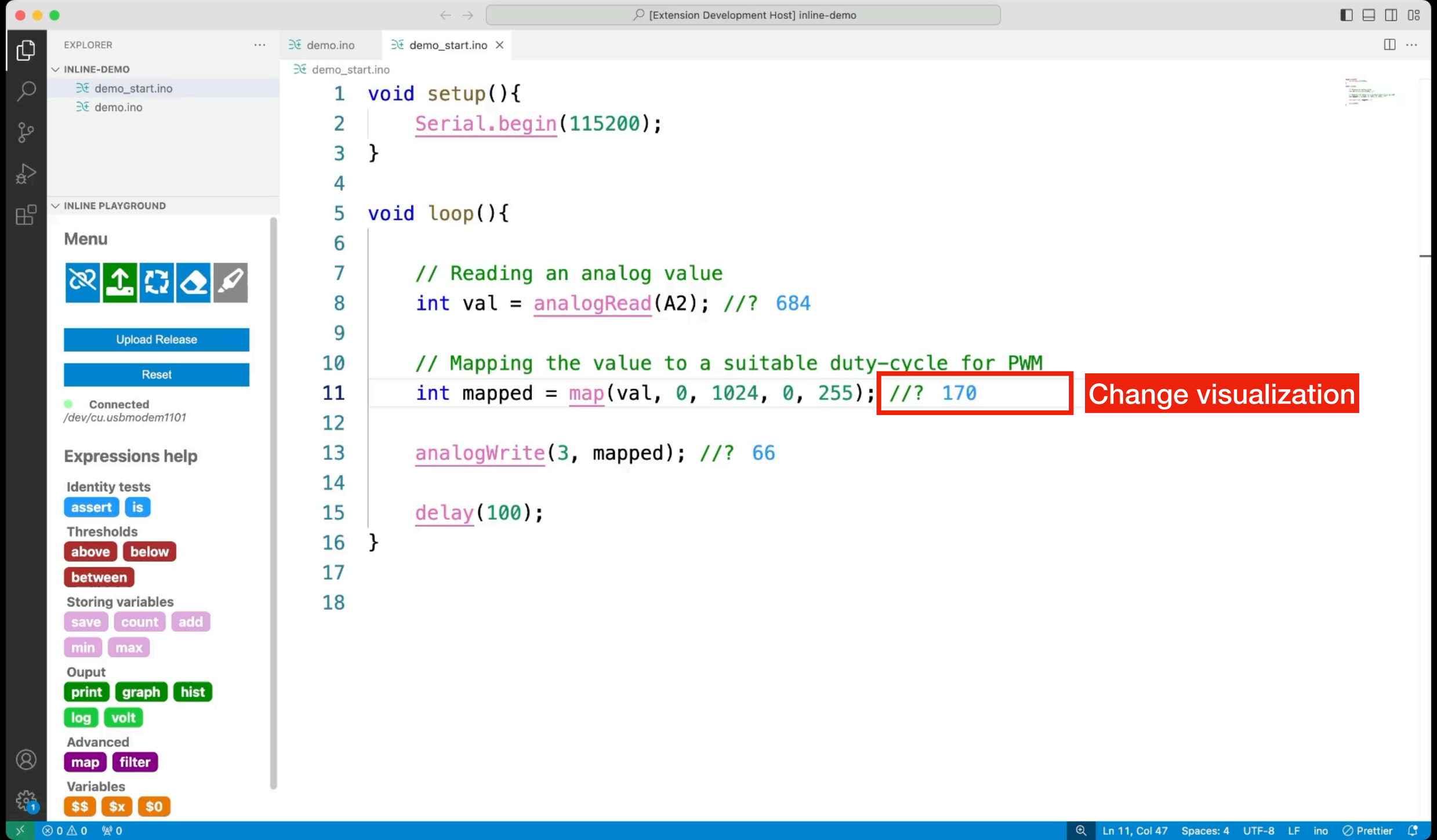


What are debugging logs anyway?

- Where/When/How are **logs** generated?
- Logs are disjointed from the code
- How to store or manipulate them in a simple way?




[Extension Development Host] inline-demo
EXPLORER
INLINE-DEMO
demo.ino
demo_start.ino
demo_start.ino
Bringing the debugging log back in the code
INLINE PLAYGROUND
Menu
Upload Release
Reset
Connected
/dev/cu.usbmodem1101
Expressions help
Identity tests
assert
is
Thresholds
above
below
between
Storing variables
save
count
add
min
max
Ouput
print
graph
hist
log
volt
Advanced
map
filter
Variables
$$
$x
$0
4
5 void loop(){
6
7 // Reading an analog value
8 int val = analogRead(A2); //? 383
9
10 // Mapping the value to a suitable duty-cycle for PWM
11 int mapped = map(val, 0, 1024, 0, 255); //? 95
12
13 analogWrite(3, mapped); //? 37
14
15 delay(100);
16 }
17
18
Display inline and trace execution
Ln 11, Col 47
Spaces: 4
UTF-8
LF
ino
Prettier


[Extension Development Host] inline-demo
EXPLORER
INLINE-DEMO
demo_start.ino
demo.ino
INLINE PLAYGROUND
Menu
Upload Release
Reset
Connected
/dev/cu.usbmodem1101
Expressions help
Identity tests
assert
is
Thresholds
above
below
between
Storing variables
save
count
add
min
max
Ouput
print
graph
hist
log
volt
Advanced
map
filter
Variables
$$
$x
$0
demo.ino
demo_start.ino
demo_start.ino
1 void setup(){
2 Serial.begin(115200);
3 }
4
5 void loop(){
6
7 // Reading an analog value
8 int val = analogRead(A2); //? 684
9
10 // Mapping the value to a suitable duty-cycle for PWM
11 int mapped = map(val, 0, 1024, 0, 255); //? 170
12
13 analogWrite(3, mapped); //? 66
14
15 delay(100);
16 }
17
18
Change visualization
Ln 11, Col 47
Spaces: 4
UTF-8
LF
ino
Prettier

[Extension Development Host] inline-demo

EXPLORER

INLINE-DEMO

demo_start.ino

demo.ino

INLINE PLAYGROUND

Menu

Upload Release

Reset

Connected
/dev/cu.usbmodem1101

Expressions help

Identity tests
assert is

Thresholds
above below
between

Storing variables
save count add
min max

Ouput
print graph hist
log volt

Advanced
map filter

Variables
$$ $x $0

demo.ino | demo_start.ino

demo_start.ino

```
1  void setup(){
2      Serial.begin(115200);
3  }
4
5  void loop(){
6
7      // Reading an analog value
8      int val = analogRead(A2); //?  383
9
10     // Mapping the value to a suitable duty-cycle for PWM
11     int mapped = map(val, 0, 1024, 0, 255); // graph?  95
12
13     analogWrite(3, mapped); //?  37
14
15     delay(100);
16 }
17
18
```

Manipulate values

Ln 11, Col 53 (5 selected) Spaces: 4 UTF-8 LF ino Prettier

[Extension Development Host] inline-demo

EXPLORER

INLINE-DEMO
- demo_start.ino
- demo.ino

INLINE PLAYGROUND

Menu

Upload Release

Reset

Connected
/dev/cu.usbmodem1101

Expressions help

Identity tests
assert is

Thresholds
above below
between

Storing variables
save count add
min max

Ouput
print graph hist
log volt

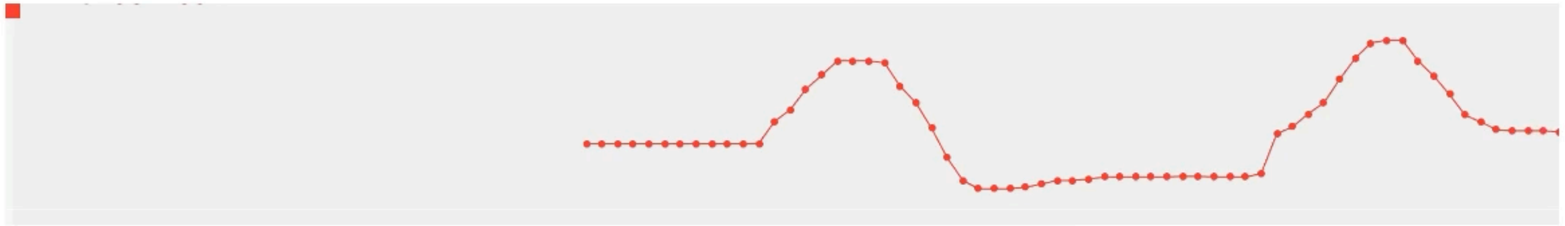
Advanced
map filter

Variables
$$ $x $0

demo.ino | demo_start.ino

demo_start.ino

```
1  void setup(){
2      Serial.begin(115200);
3  }
4
5  void loop(){
6
7      // Reading an analog value
8      int val = analogRead(A2); // min x | max y ?
9
10     // Mapping the value to a suitable duty-cycle for PWM
11     int mapped = map(val, 0, 1024, 0, 255); // graph?  113
```



Plot multiple values

```
12
13     analogWrite(3, mapped); //?  44
14
15     delay(100);
16 }
17
18
```

Ln 8, Col 33 (15 selected) Spaces: 4 UTF-8 LF ino Prettier

What has been done?
Our Contribution

Inspirations: Tools for debugging software

- Lab studies [Ko 2006, Piorkowski 2013] show that 35-50% of programmers' time is spent navigating code and program output (**debugging**).

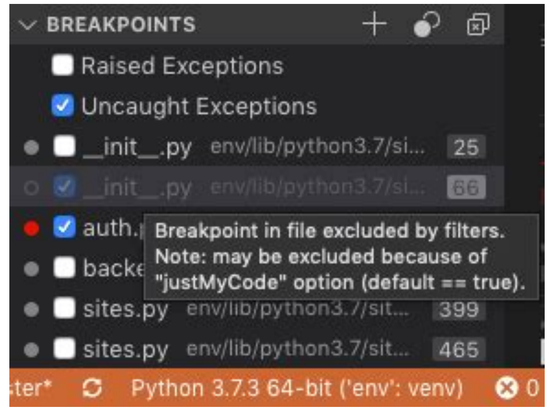
- **Bret Victor**'s notion of *representations of dynamic behavior*

- **Tools** for end user programmer

  A) **Symbolic debugger** and **breakpoints**

  B) **QuokkaJS** / Console Ninja

  C) **Copilot** / ChatGPT and **LLM**-powered tools



GitHub Copilot

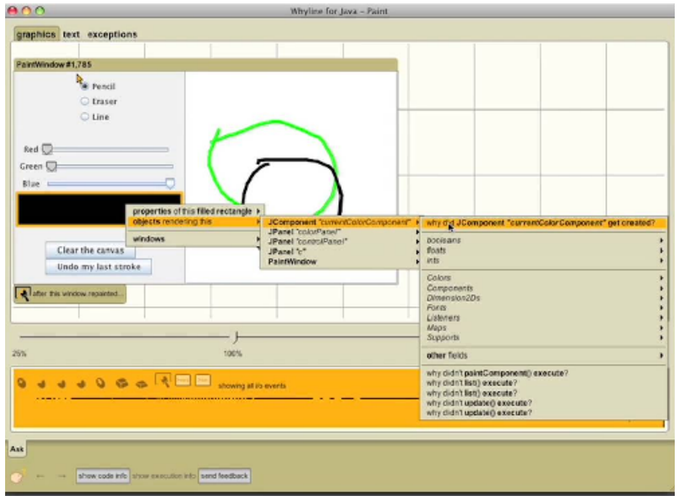
Cursor

Inspirations: Research Tools for debugging software

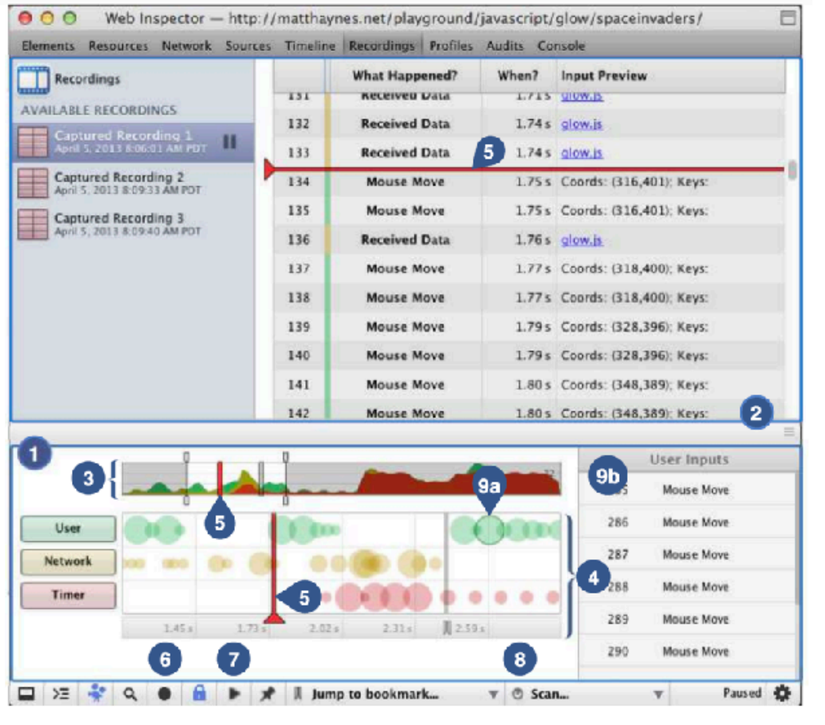
- **Log-it**: instrument logs for better visualization
- **TimeLapse**: replay program execution
- **WhyLine**: ask why and why not

We focus on hardware instead

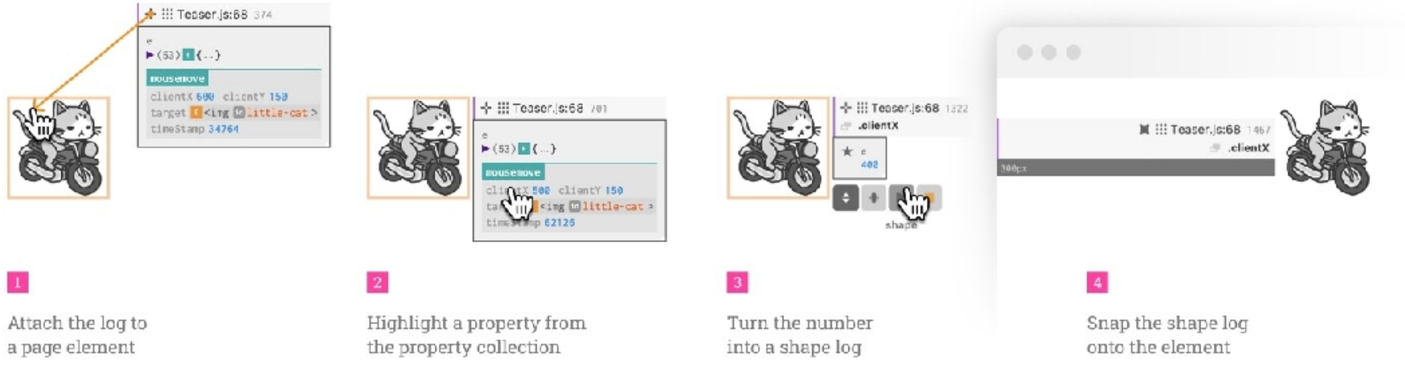
Whyline [CHI 2004]



TimeLapse [UIST 2013]



Log-it [CHI 2023]



1 Attach the log to a page element

2 Highlight a property from the property collection

3 Turn the number into a shape log

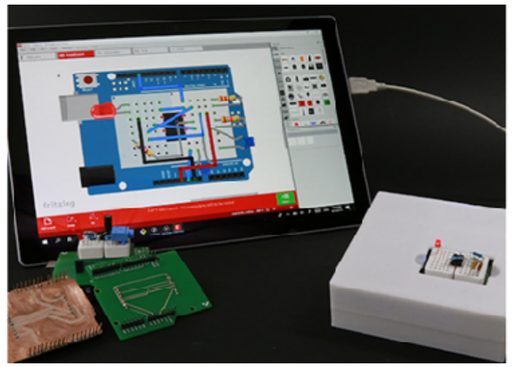
4 Snap the shape log onto the element

Related work: Tools for hardware construction

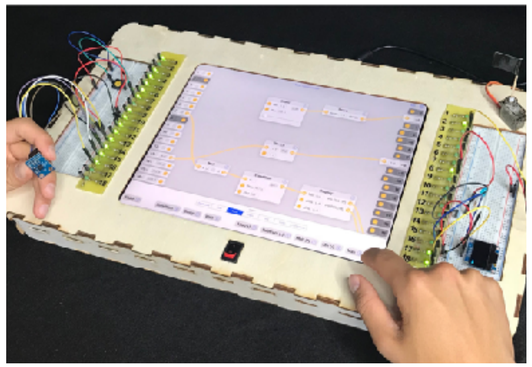
- Hardware tools focused on **facilitating construction** of circuits

A) **Simplifying wiring**:, VirutalWire, FlowBoard, Jacdac

B)**Virtualizing components**: Proxino,VirtualComponent

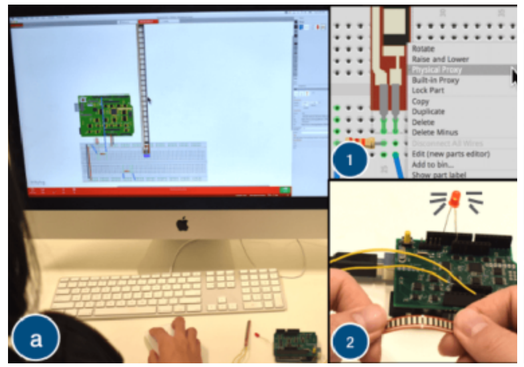


VirtualWire
[TEI 2021]

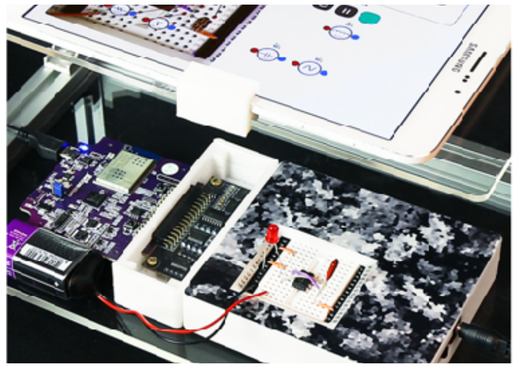


FlowBoard
[TOCHI 2023]

JacDac
[IMWUT 2022]



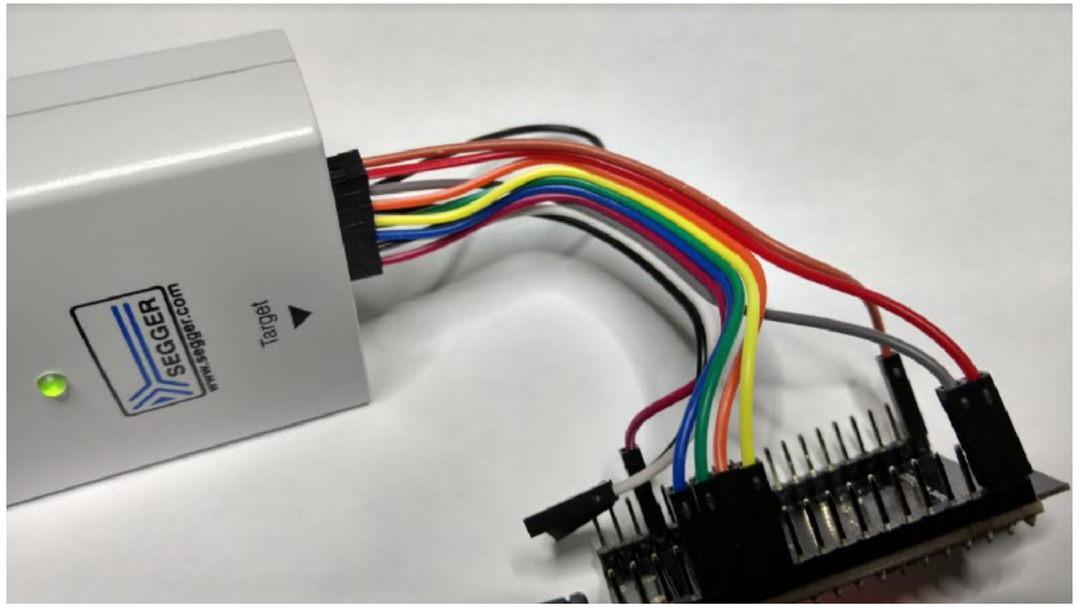
Proxino
[UIST 2019]



VirtualComponent
[CHI 2019]

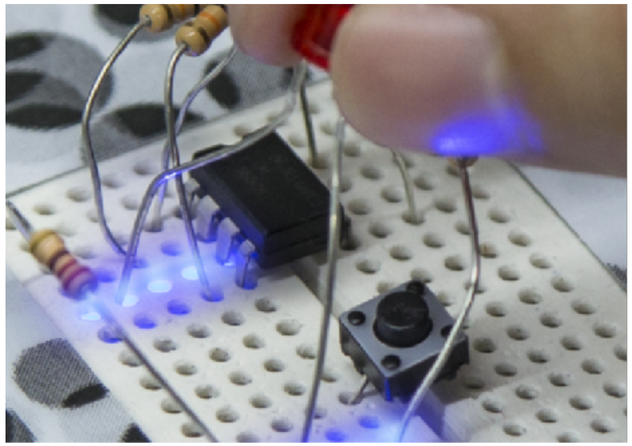
Related work: Tools for debugging hardware

- **JTAG** and boundary scanners: the hard way
- SchemaBoard & ARDW: debugging **schematic**
- CircuitSense & ToastBoard: **current** and **voltage** inspector
- PinPoint & HeyTeddy: **test driven programming** and **simulation**
- Bifröst: non real-time hardware/code inspector

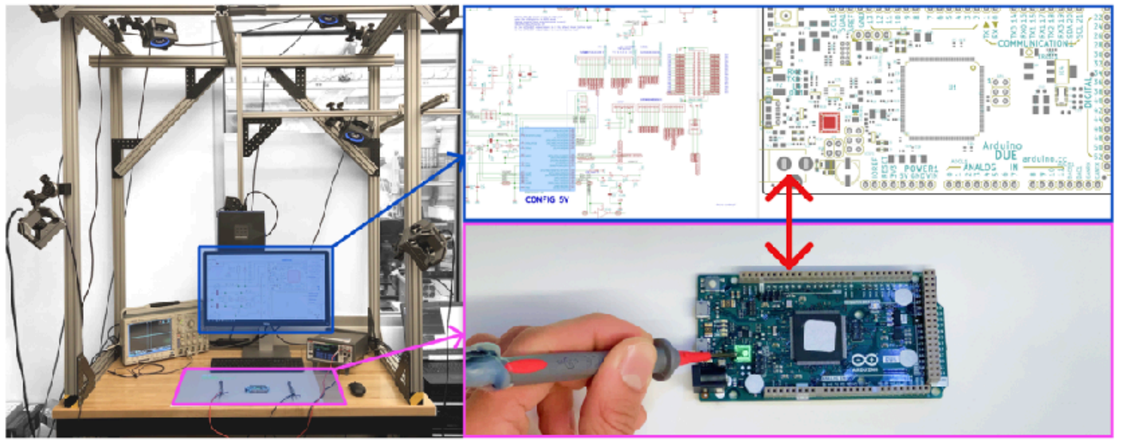


Related work: Tools for debugging hardware

- **JTAG** and boundary scanners: the hard way
- SchemaBoard & ARDW: debugging **schematic**
- CircuitSense & ToastBoard: **current** and **voltage** inspector
- PinPoint & HeyTeddy: **test driven programming** and **simulation**
- Bifröst: non real-time hardware/code inspector



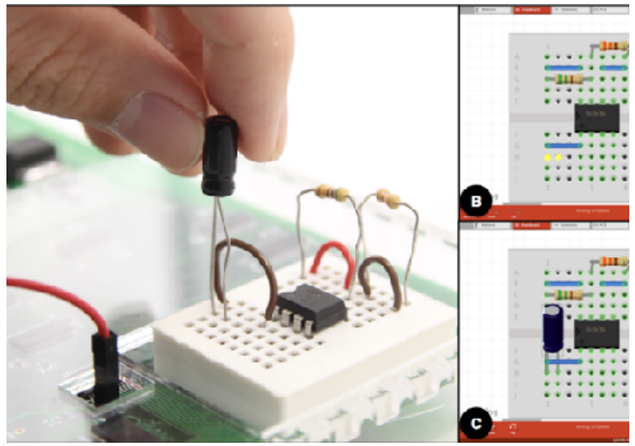
SchemaBoard
[UIST 2021]



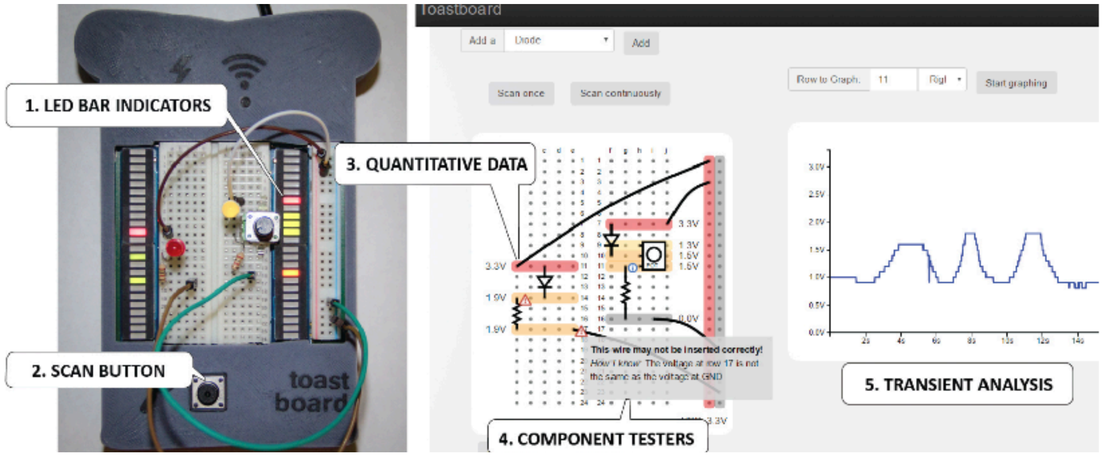
ARDW
[UIST 2022]

Related work: Tools for debugging hardware

- **JTAG** and boundary scanners: the hard way
- SchemaBoard & ARDW: debugging **schematic**
- CircuitSense & ToastBoard: **current** and **voltage** inspector
- PinPoint & HeyTeddy: **test driven programming** and **simulation**
- Bifröst: non real-time hardware/code inspector



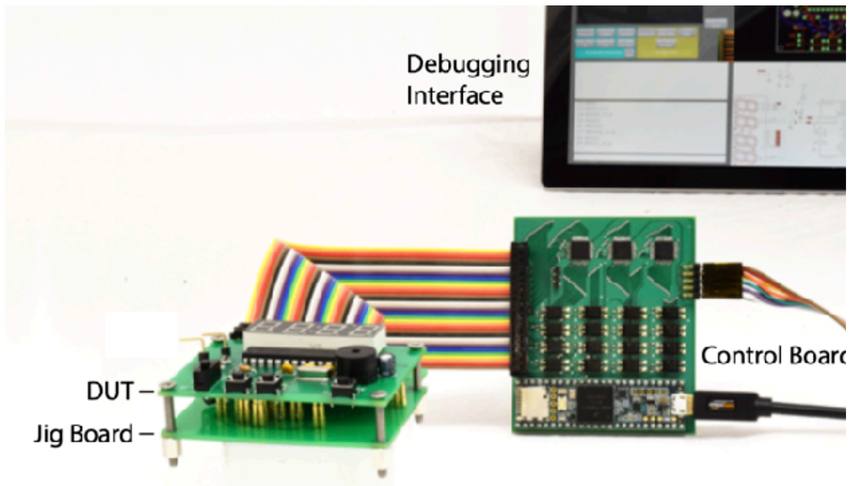
CircuitSense
[UIST 2017]



ToastBoard
[UIST 2016]

Related work: Tools for debugging hardware

- **JTAG** and boundary scanners: the hard way
- SchemaBoard & ARDW: debugging **schematic**
- CircuitSense & ToastBoard: **current** and **voltage** inspector
- PinPoint & HeyTeddy: **test driven programming** and **simulation**
- Bifröst: non real-time hardware/code inspector

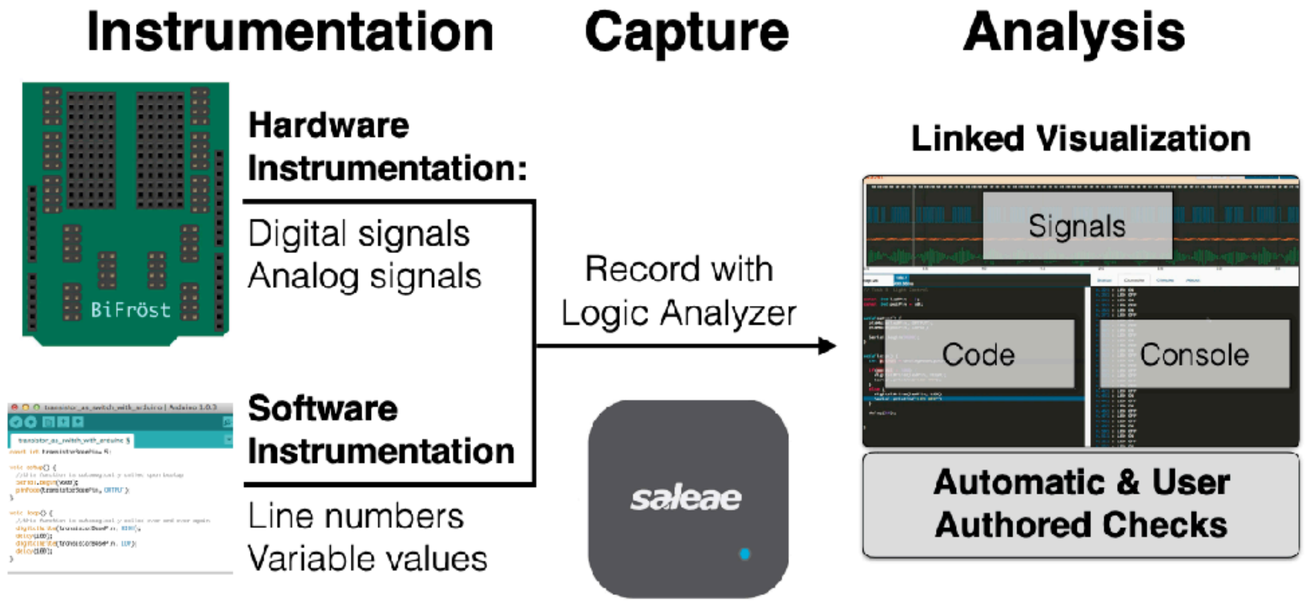


PinPoint
[CHI 2019]

HeyTeddy
[IMWUT 2020]

Related work: Tools for debugging hardware

- **JTAG** and boundary scanners: the hard way
- Schemaboard & ARDW: debugging **schematic**
- CircuitSense & ToastBoard: **current** and **voltage** inspector
- PinPoint & HeyTeddy: **test driven programming** and **simulation**
- Bifröst: **non real-time** hardware/code inspector



Bifröst
[UIST 2017]

Our contributions

1. Hardware debugging via **inline** visulization

```
A0);//?   867
```

```
Serial.print("one");//?  ...
delay(1000);
Serial.print("two");//?  ...
delay(1000);
Serial.print("three");//?  ...
delay(1000);
```

```
//assert?  ✅
//is 1?  true
//above 0?  1
//below 5?  1
//between 0, 5?  1
//save x?  1
//count y?  18
//add z?  18
//print $x?  1, 1
//log?  1
//volt?  0
//map x=>x/2?  0.5
//filter x=>x<2?  1
```

Our contributions

2. Real-time visitation and directly within the editor

```
A0);//?  176
```

```
Serial.print("one");//?  one
delay(1000);
Serial.print("two");//?  two
delay(1000);
Serial.print("three");//?  three
delay(1000);
```

```
//assert?  ✅
//is 1?  true
//above 0?  1
//below 5?  1
//between 0, 5?  1
//save x?  1
//count y?  20
//add z?  20
//print $x?  1, 1
//log?  1
//volt?  0
//map x=>x/2?  0.5
//filter x=>x<2?  1
```

Our contributions

3. Manipulation of logs for exploring the data

```
A0);//?   0
```

```
Serial.print("one");//?  one
delay(1000);
Serial.print("two");//?  two
delay(1000);
Serial.print("three");//?  three
delay(1000);
```

```
//assert?  ✅
//is 1?  true
//above 0?  1
//below 5?  1
//between 0, 5?  1
//save x?  1
//count y?  21
//add z?  21
//print $x?  1, 1
//log?  1
//volt?  0
//map x=>x/2?  0.5
//filter x=>x<2?  1
```

Collecting requirements via formative study

Formative Study

- **10 participants** (7 male, 1 female) aged 26-31 + **2 university professors** (2-15 years teaching experience)

Participants	Occupation	Department	Age	Physical Computing Experience (years)	Software Development Experience (years)	Physical Computing Self-Rank (scale:1-7)
P1	maker	Design (M.S)	26	9	6	6
P2	maker	Design (Ph.D)	28	7	7	5
P3	maker	Design (M.S)	26	6	6	6
P4	maker	CS (Ph.D)	26	2	8	3
P5	maker	Design (Ph.D)	29	10	7	6
P6	maker	CS (Ph.D)	31	7	7	6
P7	maker	CS (Ph.D)	32	6	14	4
P8	maker	Design (Ph.D)	28	8	6	6
P9	instructor	Electronics	50	15 (physical computing) + 15 (teaching)	-	-
P10	instructor	Design	35	12 (physical computing) + 2 (teaching)	-	-
Avg (SD)	-	-	28.3 (2.3)	9.9 (7.7)	7.6 (2.7)	5.3 (1.2)

- Semi-structure interviews (50 minutes)
- Translation + open-axial coding
- Three main findings

Challenge 1: Printing and managing logs

Students **use** the serial monitor and are ***formally taught*** to use it in class.

Problem arise when there are **multiple print statements**

> "it was almost impossible to follow all different data in the text logs. [...] Even when I wanted to trace them, the scroll bar kept moving on, and I lost them."

Solution 1 is to log in data files and exploring the patterns later (**not real time**)

Solution 2 is to print statements a few at a time, but this requires continuously changing the code, and recompile/upload (possibly **introducing new bugs**)

> "it would be really nice if I could simply **toggle the output of the logs** I want to see in real-time, on and off."

Challenge 2: Tracking code execution in real time

Tracking code behavior: e.g. how the value from sensors change over time.

Require spending time tracking for errors.

> "I needed to **scroll back and forth** in both windows to locate the code that needed fixing and keep track of the logs, which confused me."

One way is to use known events and **physical proxies**, like LED and actuators.

This is fast reaction time but limited to simple binary checks (e.g., over threshold or condition happens) with limited bandwidth.

> "For instance, if you expected it to be off but unexpectedly turned on, you could recognize that something was wrong."

Challenge 3: Making sense of logs

Filtering and making sense of the data is also complex

"They might look at this data, but *it's not helpful for them*." (instructor)

> "students try to print and read text logs to debug their prototypes, but they **cannot explain the meaning of those texts** even though they wrote the code."

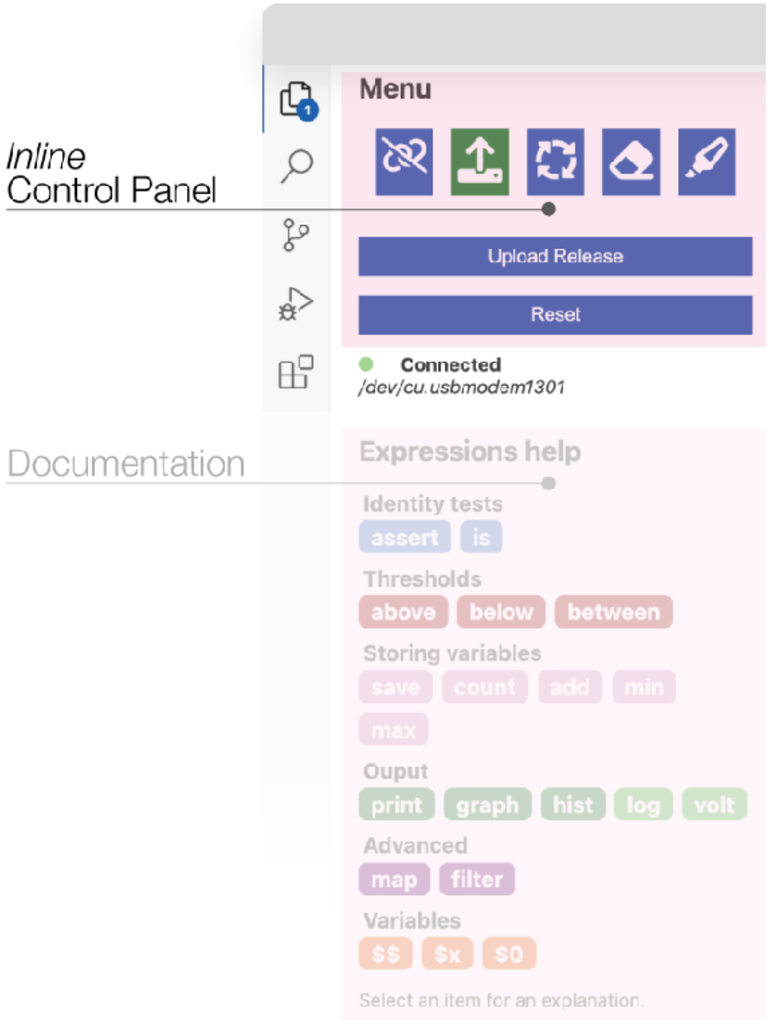
Introducing condition or code to **manipulate the code** might introduce new bugs

> "I adjusted the cutoff value to make the data smoother, then it started to be too slow and did not match the real-time movement."

The system: overview and demo

The VSCode Extension

The Menu Panel



Menu

Connect
Disconnect

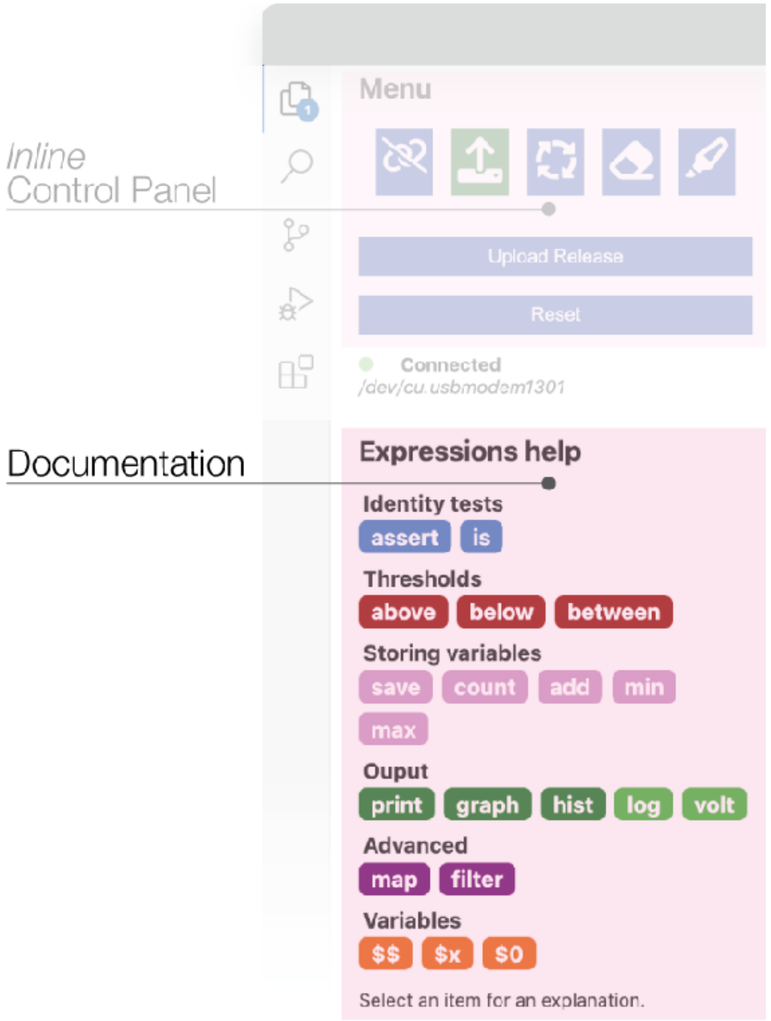
Upload
Is dirty?

Refresh

Delete
annotations

Highlight

Documentation



Expressions help

Identity tests

assert | is

Thresholds

above | below | between

Storing variables

save | count | add | min | max

Ouput

print | graph | hist | log | volt

Advanced

map | filter

Variables

$$ | $x | $0

```
// input | assert ? ✅ or ❌
```

`assert` returns ✅ if the **input** expression evaluates to **true**, otherwise ❌.

Examples

```
// 2 == 2 | assert ? ✅
```
copy

```
// 0 | assert ? ❌
```
copy

```
// 15 | above 10 | assert ? ✅
```
copy

Feature 1

1. Trace and view the execution flow of the program

[Extension Development Host] inline-demo

EXPLORER

INLINE-DEMO
- 2_trace.ino
- demo_start.ino
- demo.ino

INLINE PLAYGROUND

Menu

Upload Release

Reset

Connected
/dev/cu.usbmodem21101

Expressions help

Identity tests
assert is

Thresholds
above below between

Storing variables
save count add min max

Ouput
print graph hist log volt

Advanced
map filter

Variables
$$ $x $0

Select an item for an explanation.

demo.ino | 2_trace.ino | demo_start.ino

2_trace.ino

```ino
void setup(){
    Serial.begin(115200);
    pinMode(12, INPUT_PULLUP);
}

void loop(){

    if (digitalRead(12) == LOW){
        int val = analogRead(A2); //? ...
        int mapped = map(val, 0,1024,0,255);//? ...
        analogWrite(5, val);//? ...
    }else{
        Serial.println("Do other"); //? Do other
    }
    delay(1000);
}
```

Ln 15, Col 17 Spaces: 4 UTF-8 LF ino Prettier

Feature 2

2. Display logs inline with the code

//?

```
int value = analogRead(potentiometer); //?  555

int randomValue = random(5); //?  4

sqrt(4); //?  2
```

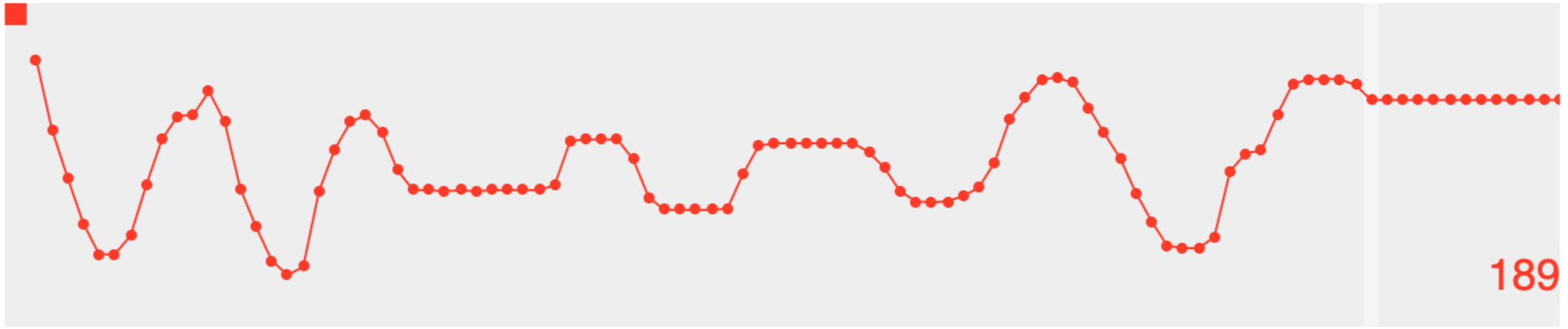
Works with any Arduino function

```
int randomValue = random(5); analogRead(potentiometer); //$1?  556
```

Chaining with pipe

```
value = analogRead(potentiometer); //below 100 | assert?   ❌
analogWrite(PWM_PIN, value/4); //below 100 | assert?  ✅
```

```
int value = analogRead(pot1); //above 0 | below 255 | graph?  189
```



Feature 3

3. Multiple visualizations along the code

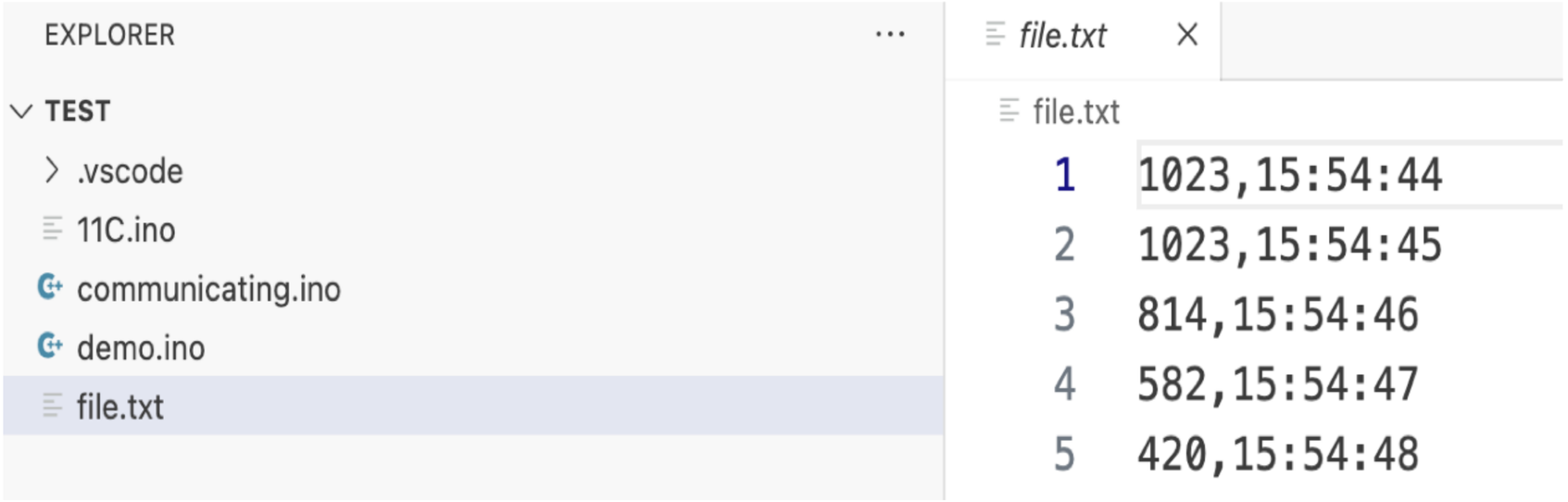
Text, Graphs or Histograms

Textual view

```
value = analogRead(potentiometer); // volt?   2.71
value = analogRead(potentiometer); // volt 3?   1.63
```

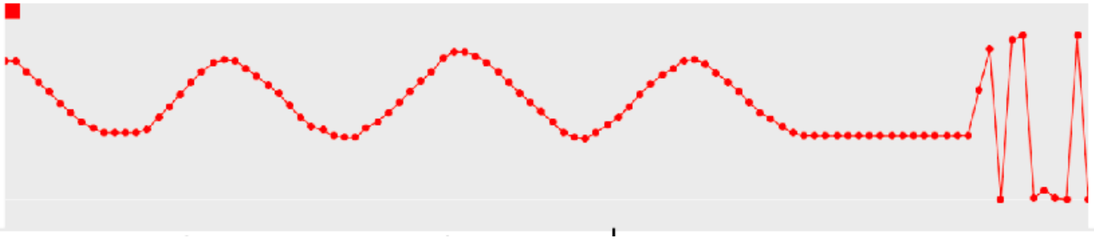
```
value = analogRead(potentiometer); // log file.txt?   556
```

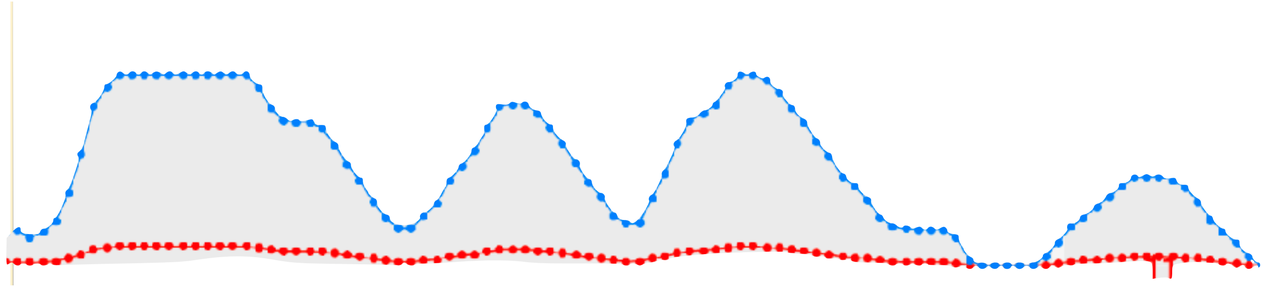
File logs



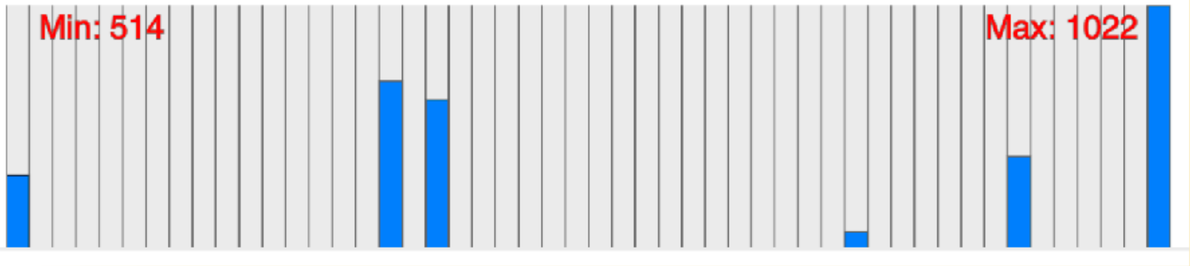
Text, Graphs or Histograms

```
int value = analogRead(pot1); //graph?
```





```
int value = analogRead(pot1); //hist?   703
```




```
int value = analogRead(pot1); //save x ?  1023
analogWrite(PWM_PIN, value/4); //hist $x?  100, 1023
```

One variable

Multiple variables

Feature 4

4. Use runtime expressions to explore and manipulate logs

Identity / Assertion

assert returns ✅ if the **input** expression evaluates to **true**, otherwise ❌.

```
// 2 == 2 | assert? ✅
// 0 | assert? ❌
// 15 | above 10 | assert? ✅
```

is checks whether **input** is equal or not to a specified value.

```
// 1 | is 1 ? true
// 0 | is 1 ? false
// "string" | is "string"? "string"
```

Visual Output

print takes an **input** and maps it with a function to an output.

graph takes an **input** and return is the filter function returns true, otherwise *None*

```
// 12 | filter x ⇒ x%2==0? 12
// 5 | filter x ⇒ x > 10? None
```

hist takes an **input** and maps it with a function to an output.

```
// 10 | map x ⇒ x*2 ? 20
```

log takes an **input** and return is the filter function returns true, otherwise *None*

```
// 12 | filter x ⇒ x%2==0? 12
// 5 | filter x ⇒ x > 10? None
```

volt takes an **input** and maps it with a function to an output.

```
// 10 | map x ⇒ x*2 ? 20
```

Using Variables

$$ / $0 takes an **input** and maps it with a function to an output.

```
// 10 | map x ⇒ x*2 ? 20
```

$name takes an **input** and return is the filter function returns

```
// 12 | filter x ⇒ x%2==0? 12
// 5 | filter x ⇒ x > 10? None
```

Thresholds

above takes an **input** and maps it with a function to an output.

```
// 10 | map x ⇒ x*2 ? 20
```

below takes an **input** and return is the filter function returns true, otherwise *None*

```
// 12 | filter x ⇒ x%2==0? 12
// 5 | filter x ⇒ x > 10? None
```

between takes an **input** and maps it with a function to an output.

```
// 10 | map x ⇒ x*2 ? 20
```

Storing Variables

save takes an **input** and maps it with a function to an output.

```
// 10 | map x ⇒ x*2 ? 20
```

count takes an **input** and return is the filter function returns true, otherwise *None*

```
// 12 | filter x ⇒ x%2==0? 12
// 5 | filter x ⇒ x > 10? None
```

add takes an **input** and maps it with a function to an output.

```
// 10 | map x ⇒ x*2 ? 20
```

min takes an **input** and return is the filter function returns true, otherwise *None*

```
// 12 | filter x ⇒ x%2==0? 12
// 5 | filter x ⇒ x > 10? None
```

max takes an **input** and maps it with a function to an output.

```
// 10 | map x ⇒ x*2 ? 20
```

Advanced

map takes an **input** and maps it with a function to an output.

```
// 10 | map x ⇒ x*2 ? 20
```

filter takes an **input** and return is the filter function returns true, otherwise *None*

```
// 12 | filter x ⇒ x%2==0? 12
// 5 | filter x ⇒ x > 10? None
```

Thresholds

```
int randomValue = random(5); //between 0,5?  1
int randomValue = random(5); //above 5?  None
int randomValue = random(5); //below 0?  None
```

Variables

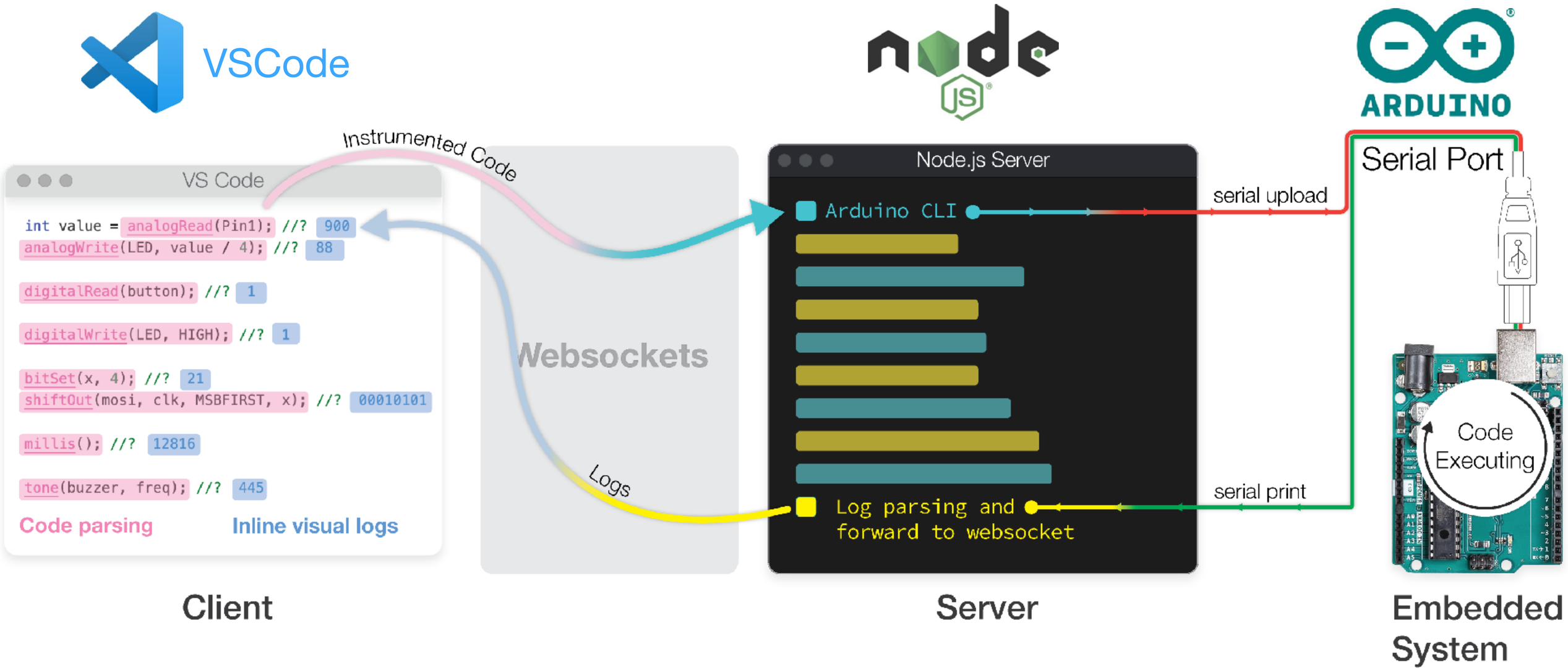
```
int randomValue = random(5); //save x?  3
analogRead(potentiometer); //$x?  3
```

map and filter

```
int value = analogRead(pot1); //map x => x/4?  189.75
int value = analogRead(pot1); //filter x => (x>=0) && (x<= 255) ?  None
int value = analogRead(pot1); //filter x => (x>=0) && (x<= 255) ?  129
```

Implementation details

System Architecture



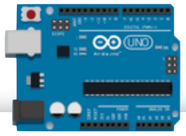
Implementation details

1. User code

```
1 void loop(){
2     shiftOut(dataPin, clockPin, MSBFIRST, 10);
3 }
```

Upload

2. Instrumented code



position
line
logs on line
identifier

```
void loop(){
    _shiftOut(dataPin, clockPin, MSBFIRST, 10, "a8a4ac", 2, 1, 1);
}

void _shiftOut(uint8_t dataPin,  // user's argument
               uint8_t clockPin, // user's argument
               uint8_t bitOrder, // user's argument
               uint8_t val,      // user's argument
               PARAMS){          // Inline METADATA via MACRO

  shiftOut(dataPin, clockPin, bitOrder, val);

  printValueFormatted(SerialWrapper(int2bin(val, bitOrder)),
                      id, line, index, items);
}
```

Serial output

```
$a8a4ac\t2\t"00001010"
```

Start tag — identifier — line — payload

Serializer

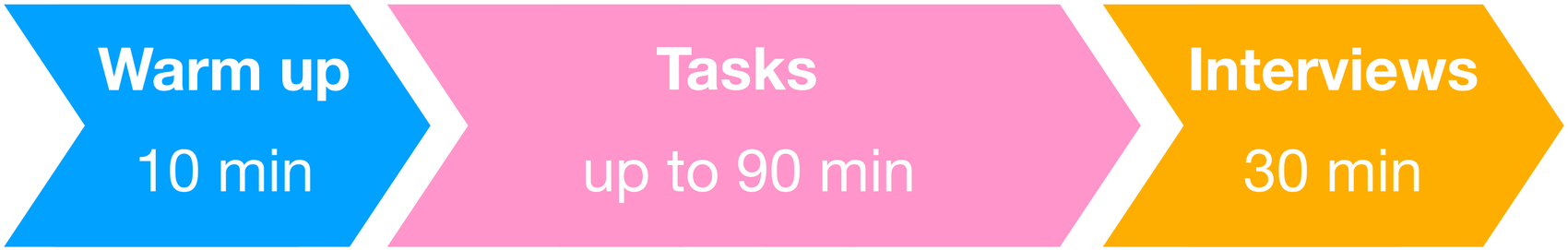
3. *Inline* Log Decorators

```
1 void loop(){
2     shiftOut(dataPin, clockPin, MSBFIRST, 10);//? 00001010
3 }
```

Evaluation with users

User Study

- **12 makers** (8 male, 4 female) aged 20-34 (avg=25.7, sd=4) from our institution
- Writing code and **findings software/hardware bugs** in 5 debugging exercises

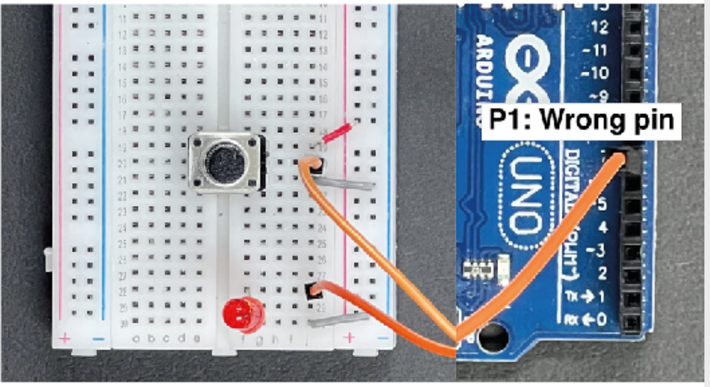


- Data **analyzed by three researchers** (axial coding)
- **Ecological validity**: open ended solutions

The tasks

Tasks	Goal	Problems and Solutions
1	Adjust LED brightness using a potentiometer	(HW/SW) Reconnect the LED to a PWM pin, not a digital one
		(HW/SW) Replace digitalRead with analogRead
		(HW/SW) Map analogRead values to a 0-255 range
2	Toggle the LED with a button	(HW/SW) Set button pinMode as "INPUT_PULLUP"
		(SW) Use "==" for equality, not "="
		(HW/SW) Replace analogRead with digitalRead
3	Alternate two LEDs using random values	(HW/SW) Set pinMode for LEDs as "OUTPUT"
		(SW) Adjust if-else condition values to 0 and 1, not 1 and 2
4	Play a tune and silence the buzzer using a bend sensor	(HW) Ground the buzzer
		(SW) Correct the minimum and maximum value ranges
5	Update LED brightness using a button and a photoresistor	(HW/SW) Modify the upper bound of the value's current range to 9 in the map function

T1: Adjusting brightness of LED with a potentiometer



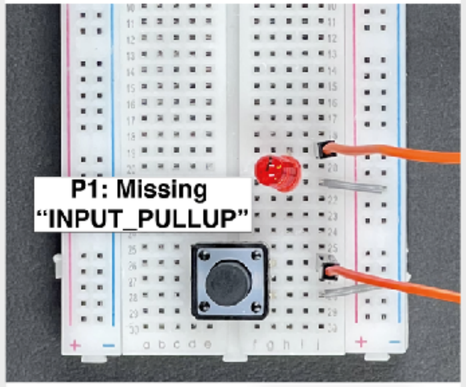

```
int value = digitalRead(potentiometer);
```

P2: Incorrect read function

P3: Incorrect mapping

```
analogWrite(PWM_PIN, value);
```

T2: Turn LED on/off with a button

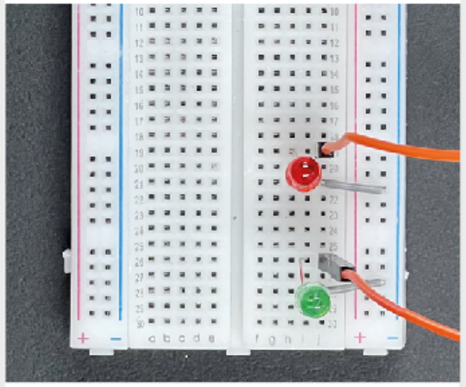



```
if (but = HIGH)
{
    analogWrite(LED_PIN, HIGH);
}
else
{
    analogWrite(LED_PIN, LOW);
}
```

P2: Wrong comparison operator

P3: Incorrect read function

T3: LEDs randomly light up in turns.

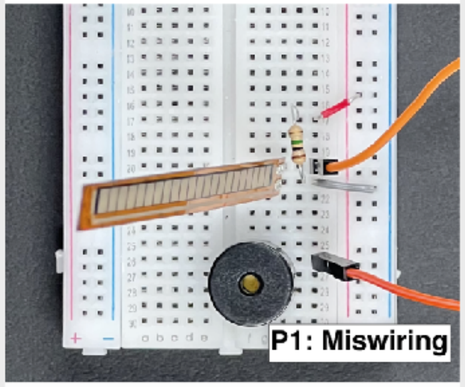


```
pinMode(LED_GREEN, INPUT);
pinMode(LED_RED, INPUT);
int randomValue = random(2);
if (randomValue == 1)
{
  digitalWrite(LED_RED, HIGH);
}
else if (randomValue == 2)
{
  digitalWrite(LED_GREEN, HIGH);
}
```

P1: Wrong pin setup

P2: Unreachable code

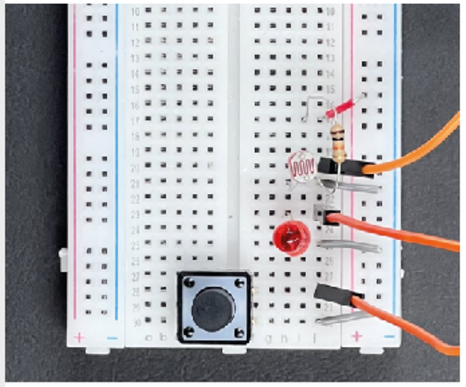
T4: Tune the buzzer with a bend sensor




```
int val = analogRead(bendSensor);

int minV = 0;
int maxV = 1023;

val = map(val, minV, maxV, 0, 1000);
```

P2: Incorrect mapping

T5: Adjust LED brightness with a photoresistor

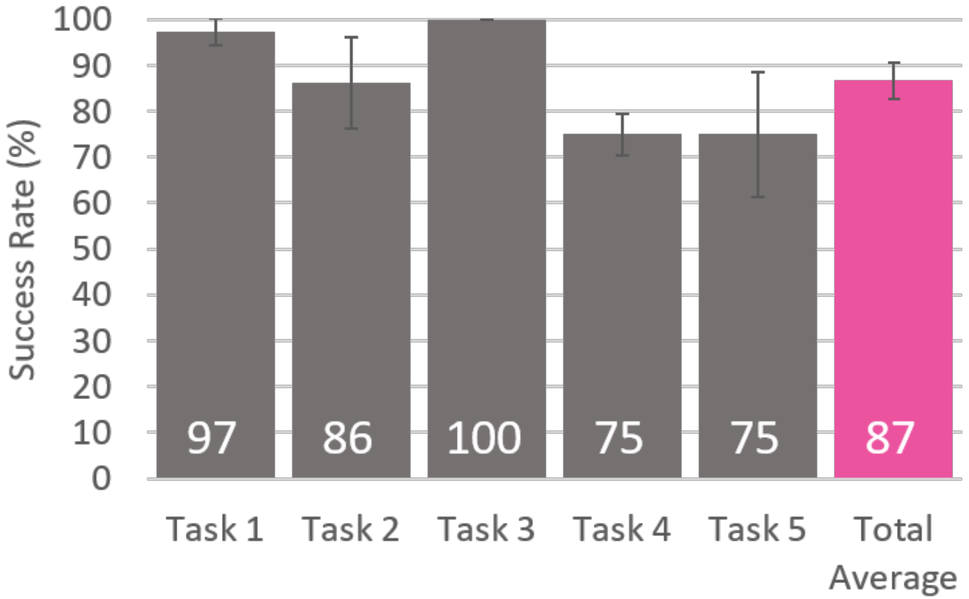


```
val = map(val, 0, 1023, 0, 255);
```

P1: Incorrect mapping

Overall sentiment

- Ten participants did well. 8 participants identified all bugs and 2 made a single mistake.
- Two users unable to identify 3-4 bugs



- **Features used**: `//?` 35% (n=51), output expressions 29% (42), identity tests 16% (24), variables 11% (16), and thresholds 9% (13).

- Overall **appreciation** of conveniency ("don't have to change the code every time"). **Histograms** used for identifying operational ranges, line **graphs** to see sensors.

Patterns of usage

- Start with `//?` and then more specific intent (e.g. `volt`) and then `graph / hist`.
- `assert` used to check whether lines were **executed**
- `above 0` to check for **nullish** value
- inline markings **left there** as annotations (can help other understand"
- Typically users were able to **identify the source of a problem**, but not necessarily how to fix it

In conclusions

1

Print value *Inline*

printing functions value next to the line of code

input from hardware *output Inline*

```
analogWrite(LED, value / 4); //?  88
```

```
int value = analogRead(Pin1); //?  900
analogWrite(LED, value / 4); //?  88

digitalRead(button); //?  1

digitalWrite(LED, HIGH); //?  1

bitSet(x, 4); //?  21
shiftOut(mosi, clk, MSBFIRST, x); //?  00010101

millis(); //?  12816

tone(buzzer, freq); //?  445
```

2

Highlight

Tracking code's execution flow

```
if (A && B) {
    quadrature = 0;
    digitalWrite(LED1, HIGH); //?  1
} else if (!A && B) {
    quadrature = 1;
    digitalWrite(LED2, HIGH); //?  1
} else if (!A && !B) {
    quadrature = 2;
    digitalWrite(LED3, HIGH); //?  1
} else if (A && !B) {
    quadrature = 3;
    digitalWrite(LED4, HIGH); //?
}
```

3

Analyze / Visualize data *Inline*

```
// assert?  ✅
// is 1?  true
```

Identity Test

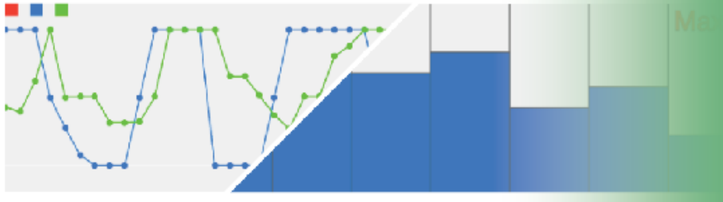
```
// above 100?  960
// below 100?  None
// between 100, 200?  None
```

Threshold

```
// save x?  1023
// count y?  831
// add z?  465522
// min a?  0
// max b?  1023
```

Storing Variables

```
// print $x, $y?   , 1023, 831
// graph $$, $x?  960, 960, 1023
// hist?  826
```



```
// log myLog.txt ?  960
// volt?  4.69
```

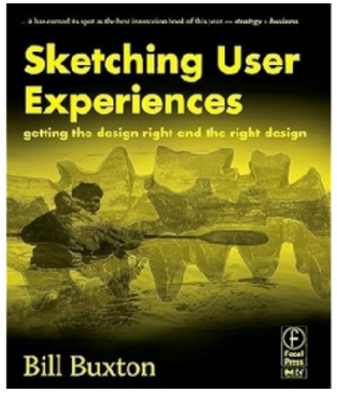
Output

```
// map x => x / 2?  480
// filter x=> x%2 == 0?  960
```

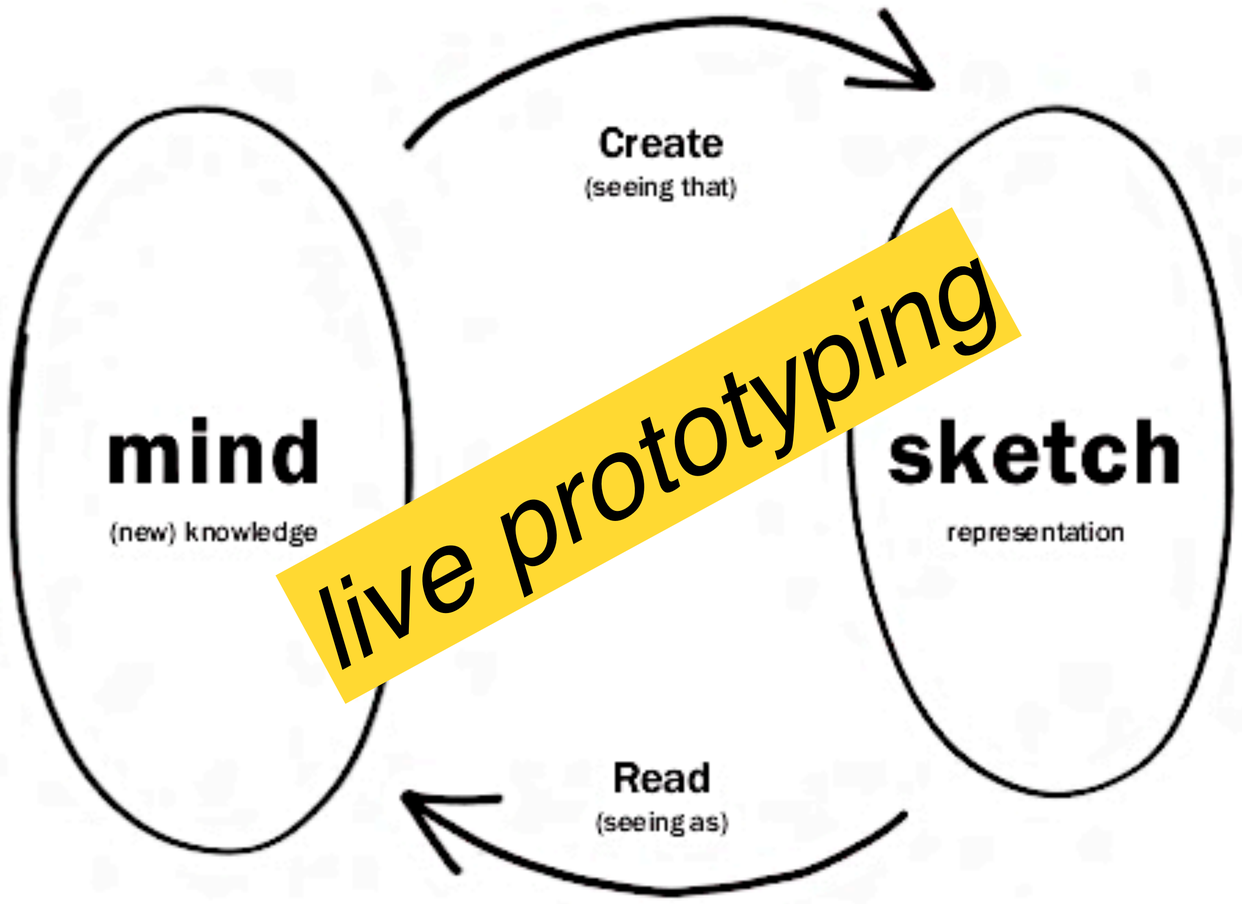
Advanced

“Sketching is about the activity not the result”

Bill Buxton



Microsoft Research



Inline

EICS

EICS 2024

Inline Visualization and Manipulation of Real-Time Hardware Log for Supporting Debugging of Embedded Programs

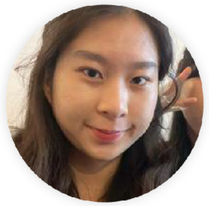
Andrea Bianchi

Department of Industrial Design
& School of Computing,
KAIST

A tool for

- **Inline visualization** of debugging logs for hardware
- **Real-time** tracing
- **Manipulation** of logs

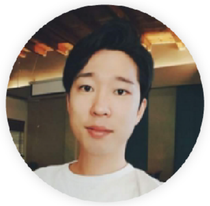
Another step toward *live prototyping* with hardware



Zhi Lin Yap
School of Computing,
KAIST

Punn Lertjaturaphat
Department of Industrial Design,
KAIST

Austin Z. Henley
Microsoft



Kongpyung (Justin) Moon
Department of Industrial Design,
KAIST

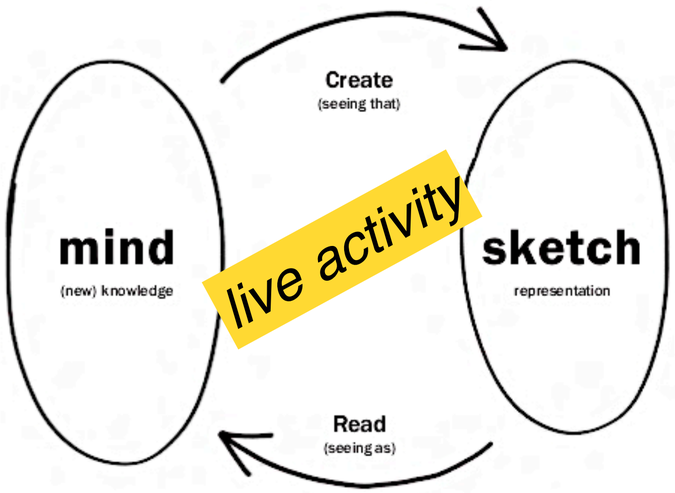
Yoonji Kim
College of Art & Technology,
Chung-Ang University

KAIST

CAU 중앙대학교

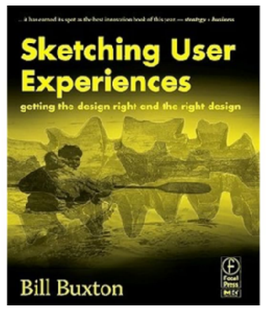
Microsoft

Appendix



"Sketching is about the activity not the result"

Bill Buxton



Microsoft Research

```
152
153 void loop(){
154   potVal = analogRead(potPin); //? 0
155   int length = map(potVal, 0, 1023, 0, 11); //? 0
156   int brightness = map(potVal, 0, 1023, 10, 50); //? 10
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
```

Video