Creating Chaos and Hard Faults

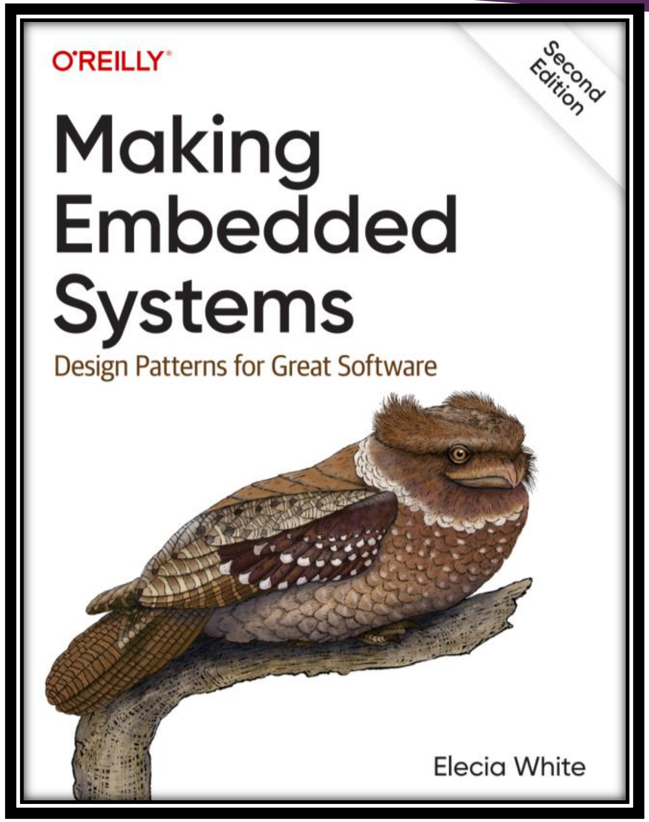
...because you can

Elecia White

Agenda

1. Smash the stack!
2. Wait, what is the stack?
3. Debugging
4. Creating hard faults
5. Debugging hard faults
6. Debugging impossible bugs

Speaker:
Elecia White



Embedded software engineer

Goal: Build interesting gadgets that make the world a better place

If you like this talk, check out my book and podcast:

- **Making Embedded Systems: Design Patterns for Great Software**, 2nd edition, 2024.
- **Embedded.fm**, a show about engineering and engineers, wherever you get your podcasts from.

1

Smash the stack!

One of the best and most effective ways to cause chaos on your embedded system.

Stack Chaos!

Overflow a buffer

This type of error allows anyone to run code on your processor.

```c
#define MAX_NAME_LENGTH 10
// enter more than 10 characters to cause problems


void unbounded_fill_from_input(char* name)
{
    int i = 0;
    char ch;
    ch = getchar();
    while (ch != '\n') { // wait for new line
        name[i] = ch; i++;
        ch = getchar();
    }
    name[i] = NULL; // NULL terminate string
}
void trash_the_stack(void)
{
    char name[MAX_NAME_LENGTH];
    printf("Hello! Tell me your name: ");
    unbounded_fill_from_input(name);
    printf("Hello %s\r\n", name);
}
```

2

Wait, what is the stack?

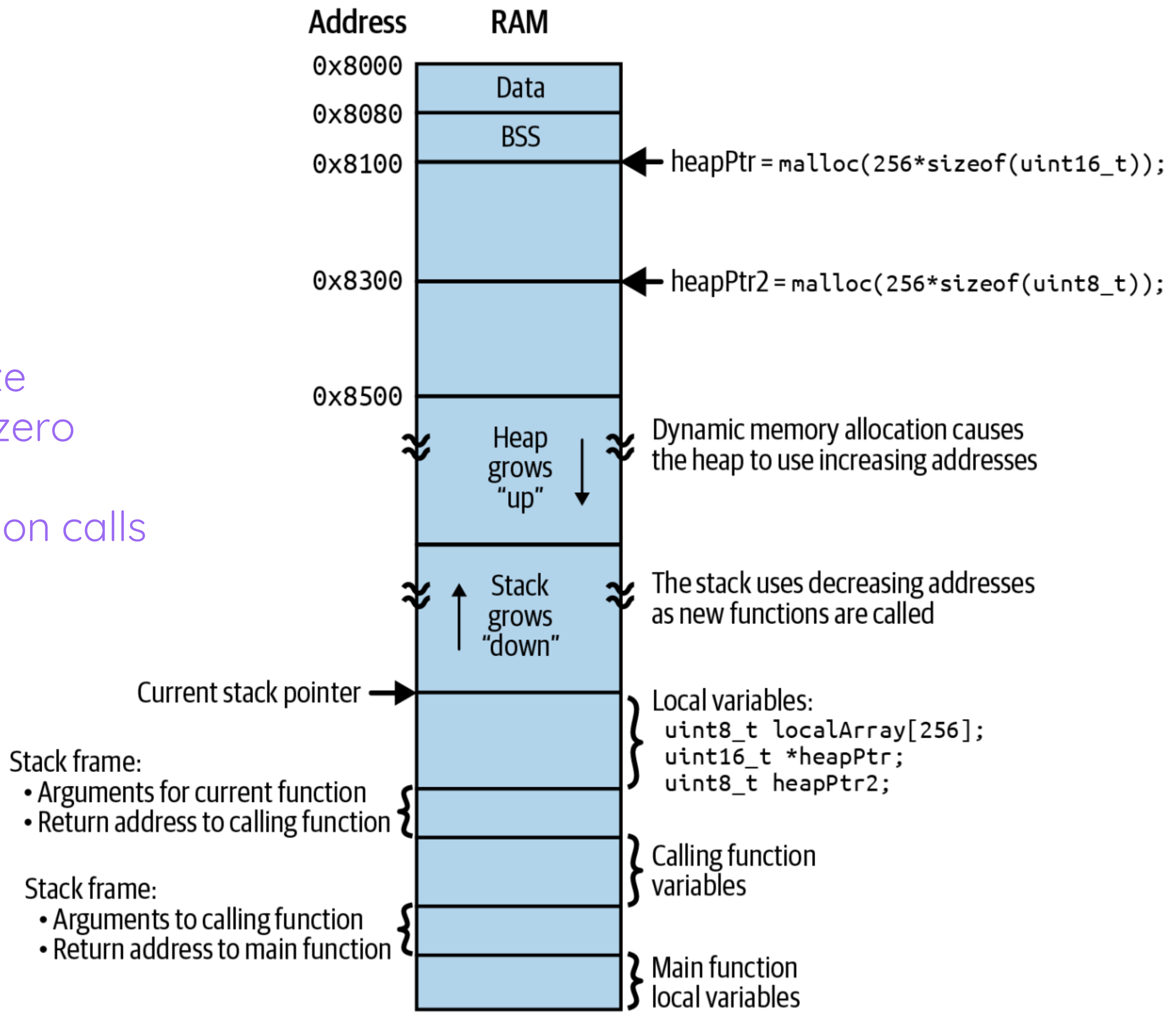
Chaos is more fun when you understand what you are doing.

Memory Map

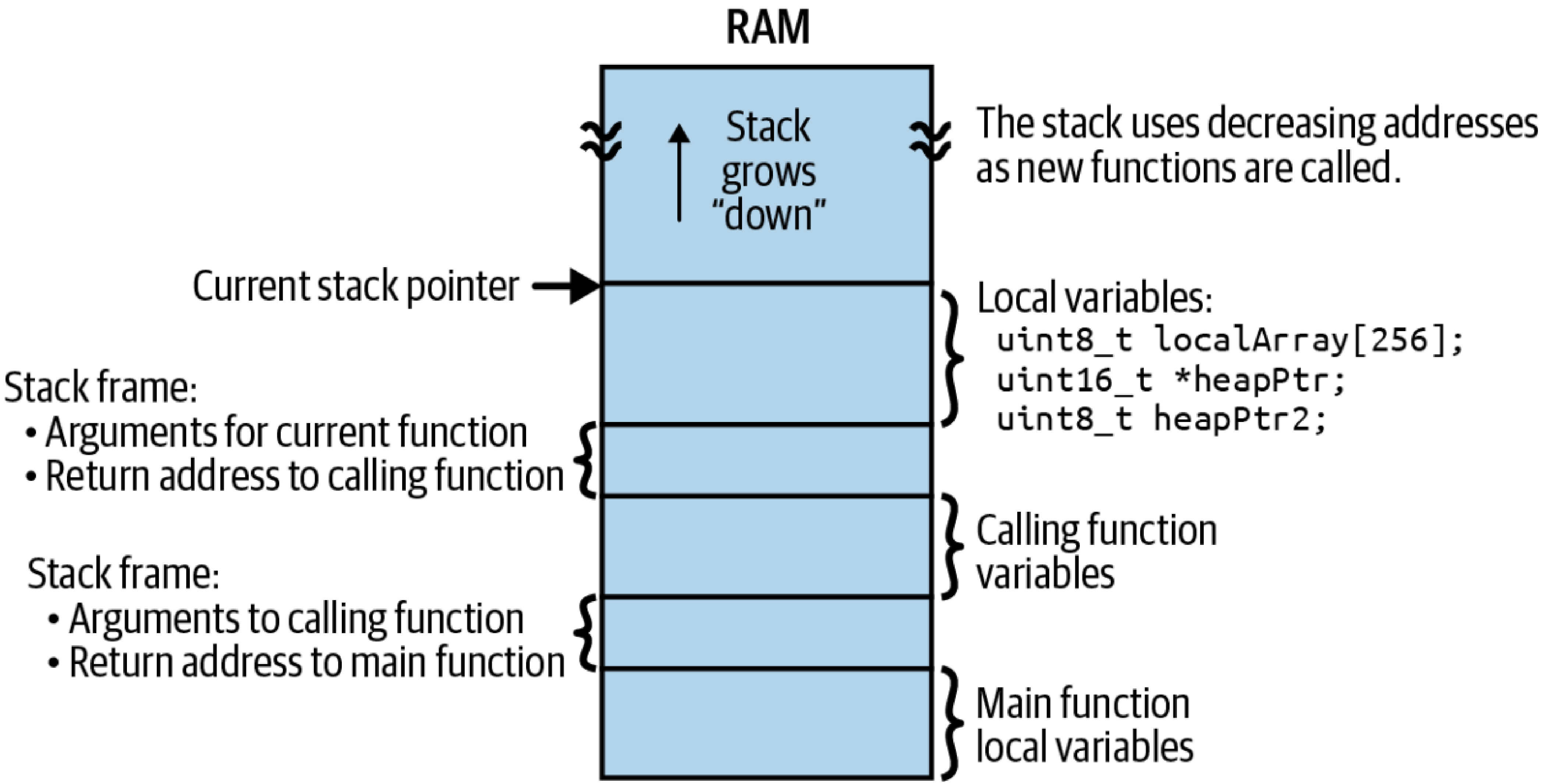
embedded.fm/blog/mapfiles

Memory Map: RAM

Data: global variables you initialize
BSS: global variables auto-init to zero
Heap: malloc space
Stack: working memory for function calls



Looking More Closely at the Stack



Stack Chaos!

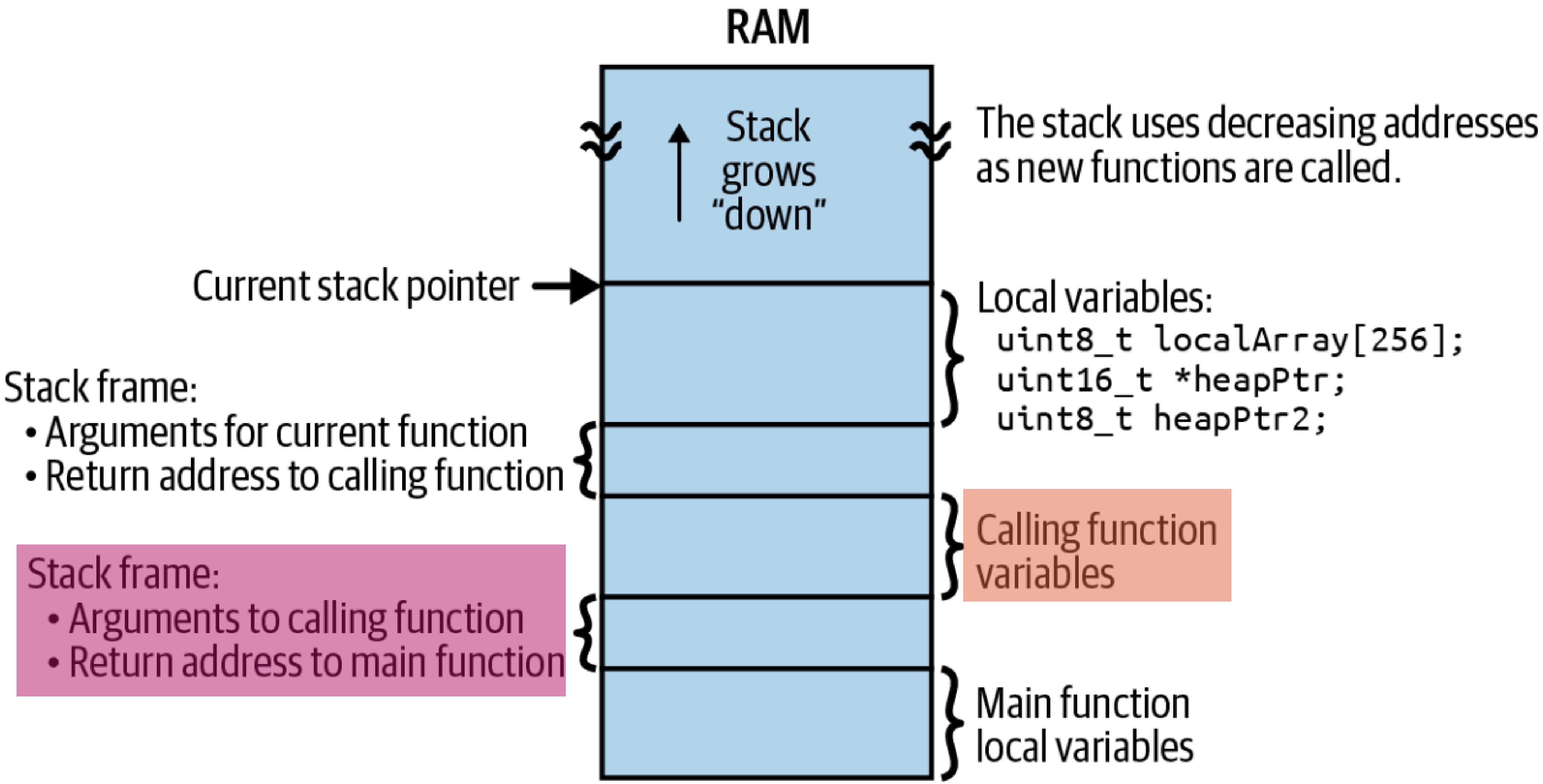
Overflow a buffer

This type of error allows anyone to run code on your processor.

```c
#define MAX_NAME_LENGTH 10
// enter more than 10 characters to cause problems


void unbounded_fill_from_input(char* name)
{
    int i = 0;
    char ch;
    ch = getchar();
    while (ch != '\n') { // wait for new line
        name[i] = ch; i++;
        ch = getchar();
    }
    name[i] = NULL; // NULL terminate string
}
void trash_the_stack(void)
{
    char name[MAX_NAME_LENGTH];
    printf("Hello! Tell me your name: ");
    unbounded_fill_from_input(name);
    printf("Hello %s\r\n", name);
}
```

Looking More Closely at the Stack



Stack Chaos!

Overflow a buffer

This type of error allows anyone to run code on your processor.

```
#define MAX_NAME_LENGTH 10
// enter more than 10 characters to cause problems


void unbounded_fill_from_input(char* name)
{
    int i = 0;
    char ch;
    ch = getchar();
    while (ch != '\n') { // wait for new line
        name[i] = ch; i++;
        ch = getchar();
    }
    name[i] = NULL; // NULL terminate string
}
void trash_the_stack(void)
{
    char name[MAX_NAME_LENGTH];
    printf("Hello! Tell me your name: ");
    unbounded_fill_from_input(name);
    printf("Hello %s\r\n", name);
}
```

3

Debugging

Let’s start with normal debugging.

Six Stages of Debugging

- That can't happen.
- That doesn't happen on my machine.
- That shouldn't happen.
- Why does that happen?
- Oh! I see…
- How did that ever work?

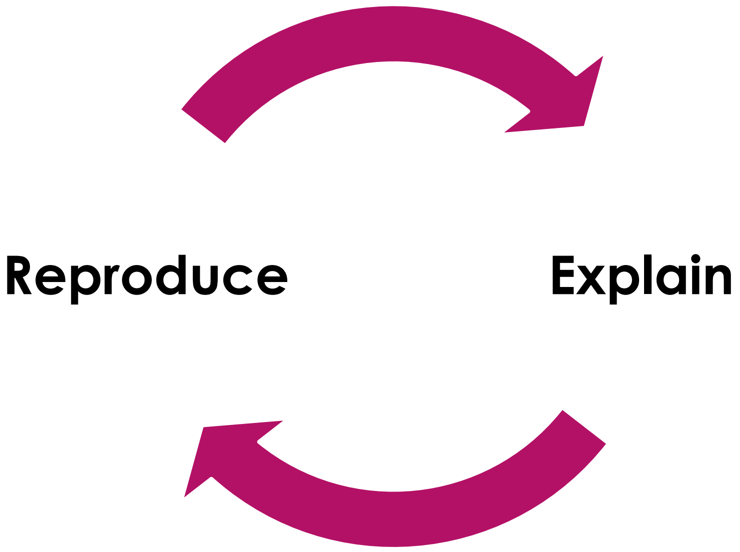
Debugging

Reproduce the bug

Make it happen on command with a minimal code set.

Explain the bug

Rubber duck debugging is amazing.



4

Creating hard faults

Intentionally causing crashes is way more fun than when it happens in real life.

Hard faults

- Divide by zero
- Incorrect memory access (unaligned)
- Talking to things that aren’t there
- Weird return code from interrupt/RTOS
- Running from non-code memory (undefined instruction, invalid)
- Stack error
- Data access disallowed memory
- Instruction access disallowed memory
- Debug breakpoint (asserts in production code?)
- Invalid address in exception

Demo!

I WROTE SOME TERRIBLE CODE FOR YOU.
I HOPE YOU APPRECIATE IT.

You can find the code at
https://github.com/eleciawhite/making-embedded-systems/blob/main/Ch09_Debugging

Demo!

Fault 1: Divide by zero

```
int divide_by_zero(void)
{
    int a = 1;
    int c = 0;
    int b = a/c;
    return b; // forces compiler to actually run this
}
```

Demo!

FAULT 1: DIVIDE BY ZERO

```c
int divide_by_zero(void)
{
  int a = 1;
  int c = 0;
  int b = a/c;
  return b; // forces compiler to actually run this
}
```

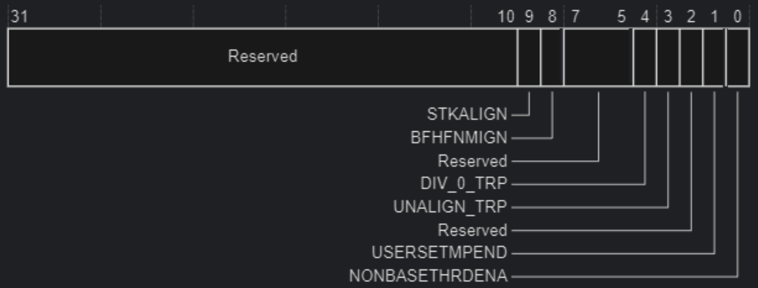
Configuration and Control Register

The CCR controls entry to Thread mode and enables:

- the handlers for NMI, hard fault and faults escalated by FAULTMASK to ignore BusFaults
- trapping of divide by zero and unaligned accesses
- access to the STIR by unprivileged software, see *Software Trigger Interrupt Register*.

See the register summary in Table 4.12 for the CCR attributes.

The bit assignments are:



Divide by zero causes a hard fault

only if DIV_0_TRP is set in SCB->CCR

System Control Block ->

Configuration and Control Register

Cortex-M3 **Devices Generic User Guide** describes these processor registers

Demo!

Fault 2: Write to NULL

Works fine.

But should it?

```c
// uninitalized global ends up initialized to 0
int* global_ptr_to_null = NULL;
int* global_ptr_unitialized;
int write_to_null(void) {
  int* ptr_to_null = NULL;
  int* ptr_unitialized;

  *global_ptr_to_null  = 10;    /* tries to write to address zero */
  *global_ptr_unitialized = 10; /* tries to write to address zero */
  *ptr_to_null  = 10;           /* tries to write to address zero */
  *ptr_unitialized = 10;        /* tries to write ?? somewhere ?? */

  return *global_ptr_to_null + *global_ptr_unitialized +
          *ptr_to_null + *ptr_unitialized;
}
```

Demo!

Fault 3: Not every set of bytes is a valid instruction

Touch your nose to your elbow!

```c
// Assuming 0xE0000000 is a bad instruction
int illegal_instruction_execution(void) {
  int instruction = 0xE0000000;
  int (*bad_instruction)(void) = (int(*)())(&instruction);
  return bad_instruction();
}
```

Demo!

Hard Fault Handler 1

Finally, something fails!

```c
void HardFault_Handler(void)
{
    __asm volatile
    (
        " movs r0,#4     \n"
        " movs r1, lr    \n"
        " tst r0, r1     \n"
        " beq _MSP      \n"
        " mrs r0, psp \n"
        " b _HALT    \n"
        "_MSP:       \n"
        " mrs r0, msp \n"
        "_HALT:      \n"
        " ldr r1,[r0,#20] \n"
        "  b hard_fault_handler_c \n"
        "  bkpt #0 \n"
    );
}
```

```c
void hard_fault_handler_c(unsigned long *hardfault_args){
  volatile unsigned long stacked_r0 ;
  volatile unsigned long stacked_r1 ;
```

```c
  volatile unsigned long _BFAR ;
  volatile unsigned long _MMAR ;

  stacked_r0 = ((unsigned long)hardfault_args[0]) ;
  stacked_r1 = ((unsigned long)hardfault_args[1]) ;
```

```c
  // Read the Fault Address Registers. These may not contain
  // valid values.
  // Check BFARVALID/MMARVALID to see if they are valid values
  // MemManage Fault Address Register
  _MMAR = (*((volatile unsigned long *)(0xE000ED34))) ;
  // Bus Fault Address Register
  _BFAR = (*((volatile unsigned long *)(0xE000ED38))) ;

  __asm("BKPT #0\n") ; // Break into the debugger
}
```

Vector table calls
`HardFault_Handler`
which sets up registers then calls
`hard_fault_handler_c`
which makes things more readable then breakpoints for debugging

Demo!

Hard Fault Handler 2

Stash it in RAM…
Check it on boot

```c
typedef struct __attribute__((packed)) ContextStateFrame {
  uint32_t r0;
  uint32_t r1;
  uint32_t r2;
  uint32_t r3;
  uint32_t r12;
  uint32_t lr;
  uint32_t return_address;
  uint32_t xpsr;
} sContextStateFrame;
```

```c
uint32_t r3;
uint32_t returnAddress;
uint32_t stackPointer;
} __attribute__((section(".CoreDump"))) coreDump;

void my_fault_handler_c(sContextStateFrame *frame)
{
    coreDump.key = COREDUMP_KEY;
    coreDump.r0 = frame->r0;
    coreDump.r1 = frame->r1;
    coreDump.r2 = frame->r2;
    coreDump.r3 = frame->r3;
    coreDump.return_address = frame->return_address;
    coreDump.stackPointer = frame->xpsr;
```

```
._user_heap_stack :
{
  . = ALIGN(8);
  PROVIDE ( end = . );
  PROVIDE ( _end = . );
  . = . + _Min_Heap_Size;
  . = . + _Min_Stack_Size;
  . = ALIGN(8);
} >RAM

.CoreDump :
{
} > RAM2
```

Demo!

I WROTE SOME TERRIBLE CODE FOR YOU.
I HOPE YOU APPRECIATE IT.

You can find the code at
https://github.com/eleciawhite/making-embedded-systems/blob/main/Ch09_Debugging

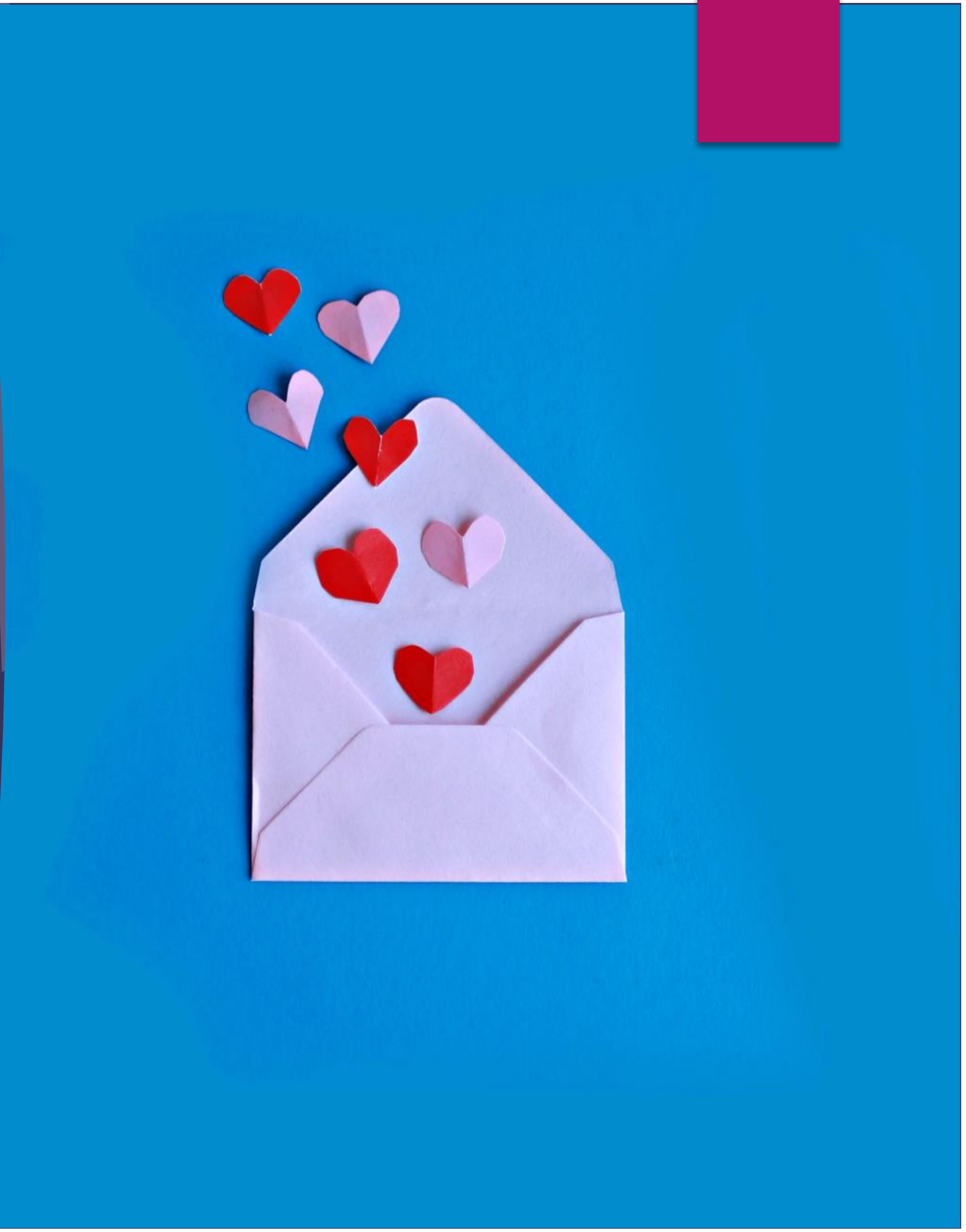
5

Debugging hard faults

Wizardry at its finest.

Hard faults

- Divide by zero
- Incorrect memory access (unaligned)
- Talking to things that aren't there
- Weird return code from interrupt/RTOS
- Running from non-code memory (undefined instruction, invalid)
- Stack error
- Data access disallowed memory
- Instruction access disallowed memory
- Debug breakpoint (asserts in production code?)
- Invalid address in exception



Don't create hard faults

Don't do bad things

- Buffer overflows (and off-by-one errors)
- Stack corruption (returning memory from a function)
- Stack overflows (recursion)
- Uninitialized variables

Don't do useful things

- Avoid heaps
- Avoid recursion
- Avoid pointers of all kinds
- Avoid function pointers most of all
- Avoid interrupts and RTOSs
- Avoid compiler and linker optimizations

Don’t create hard faults

Don’t do bad things

- ✓ Buffer overflows (and off-by-one errors)
- ✓ Stack corruption (returning memory from a function)
- ✓ Stack overflows (recursion)
- ✓ Uninitialized variables

Don’t do useful things

- ➢ Avoid heaps
- ➢ Avoid recursion
- ➢ Avoid pointers of all kinds
- ➢ Avoid function pointers most of all
- ➢ Avoid interrupts and RTOSs
- ➢ Avoid compiler and linker optimizations

```
../Core/Src/hardfaults.c: In function 'dont_return_stack_memory':
../Core/Src/hardfaults.c:83:16: warning: function returns address of local variable [-Wreturn-local-addr]
   83 |         return stack_memory;
      |                ^~~~~~~~~~~~
../Core/Src/hardfaults.c: In function 'write_to_null':
../Core/Src/hardfaults.c:50:20: warning: 'ptr_unitialized' is used uninitialized [-Wuninitialized]
   50 |   *ptr_unitialized = 10;                    /* tries to write ?? somewhere ?? */
      |   ~~~~~~~~~~~~~~~~~^~~~
```

Don’t create hard faults

Don’t do bad things

- ✓ Buffer overflows (and off-by-one errors)
- ✓ Stack corruption (returning memory from a function)
- ✓ Stack overflows (recursion)
- ✓ Uninitialized variables

Don’t do useful things

- ➢ Avoid heaps
- ➢ Avoid recursion
- ➢ Avoid pointers of all kinds
- ➢ Avoid function pointers most of all
- ➢ Avoid interrupts and RTOSs
- ➢ Avoid compiler and linker optimizations

When they inevitably happen use Interrupt Blog’s **How to Debug a Hard Fault on an ARM Cortex-M MCU** and:

- ❑ Reproduce the problem
- ❑ Explain the problem
- ❑ Create a core dump
- ❑ Fix compiler warnings

6

Debugging impossible bugs

To fight the unbeatable foe

To bear with unbearable sorrow

To run where the brave dare not go

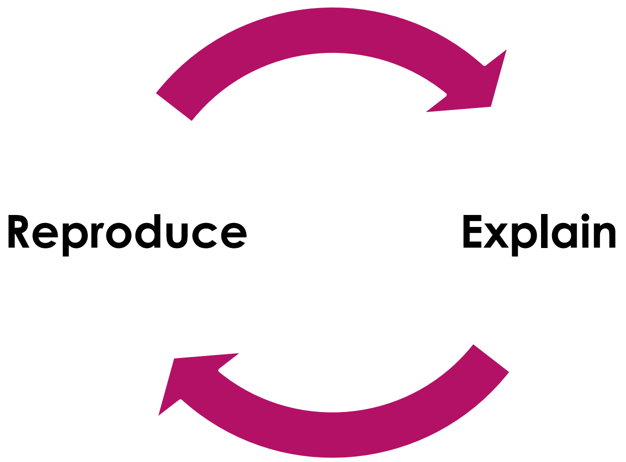
Debugging

Reproduce the bug

Make it happen on command with a minimal code set.

Explain the bug

Rubber duck debugging is amazing.



Debugging: Reproduce the bug

1. Reproduce the bug.
2. Add instrumentation.
3. Reproduce the bug with the minimal amount of code.
4. Go back to the last working version, incrementally add changes.
5. Reproduce the bug on command (or on boot) so you don't have to do ten steps to reproduce it. Remove steps if possible.
6. After making exploratory tweaks, are you sure you are running the code you think you are?

Debugging:
Explain the bug

1. Describe the bug's symptoms.
2. Explain how the bug could possibly do what it does. List all possible causes (without using ground loops).
3. Determine how to eliminate each cause, starting with the simplest.
4. Explain the bug and what you've tried to a duck. If you realize you forgot to try something, note it down.
5. Take notes on what changes have what effect. You will not remember after the third tweak.
6. Get a code review.

Explain the bug

The goal is to figure out all the places the bug might be.
This is not a complete list. You should make your own version.

Code	Hardware
Memory	Jumper wires
Timing	Cables and connectors
Resource collision	Crosstalk
Operating system	Brownouts
Too much time in critical sections	Boot order
Too long in an interrupt	Temperature dependence
Unhandled interrupt	Clock weirdness
Hard fault	Percussive maintenance
	Try a simulator

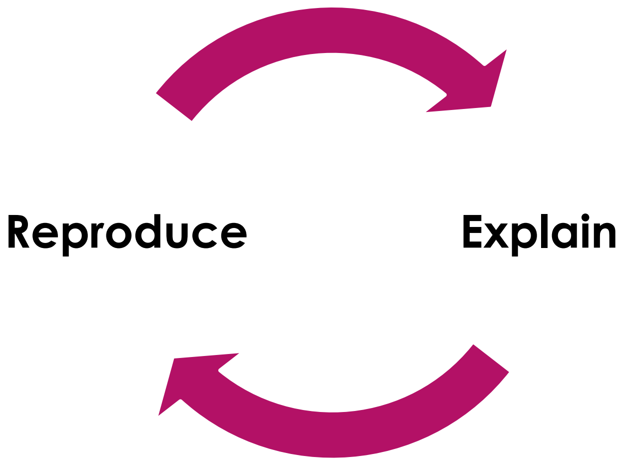
Debugging

Reproduce the bug

1. Reproduce the bug.
2. Add instrumentation.
3. Reproduce the bug with the minimal amount of code.
4. Go back to the last working version.
5. Reproduce the bug on command (or on boot) so you don't have to do ten steps to reproduce it. Remove steps if possible.
6. After making exploratory tweaks, are you sure you are running the code you think you are?

Explain the bug

1. Describe the bug's symptoms.
2. Explain how the bug could possibly do what it does. List all possible causes (without using ground loops).
3. Determine how to eliminate each cause, starting with the simplest.
4. Explain the bug and what you've tried to a duck. If you realize you forgot to try something, note it down.
5. Take notes on what changes have what effect. You will not remember after the third tweak.
6. Get a code review.

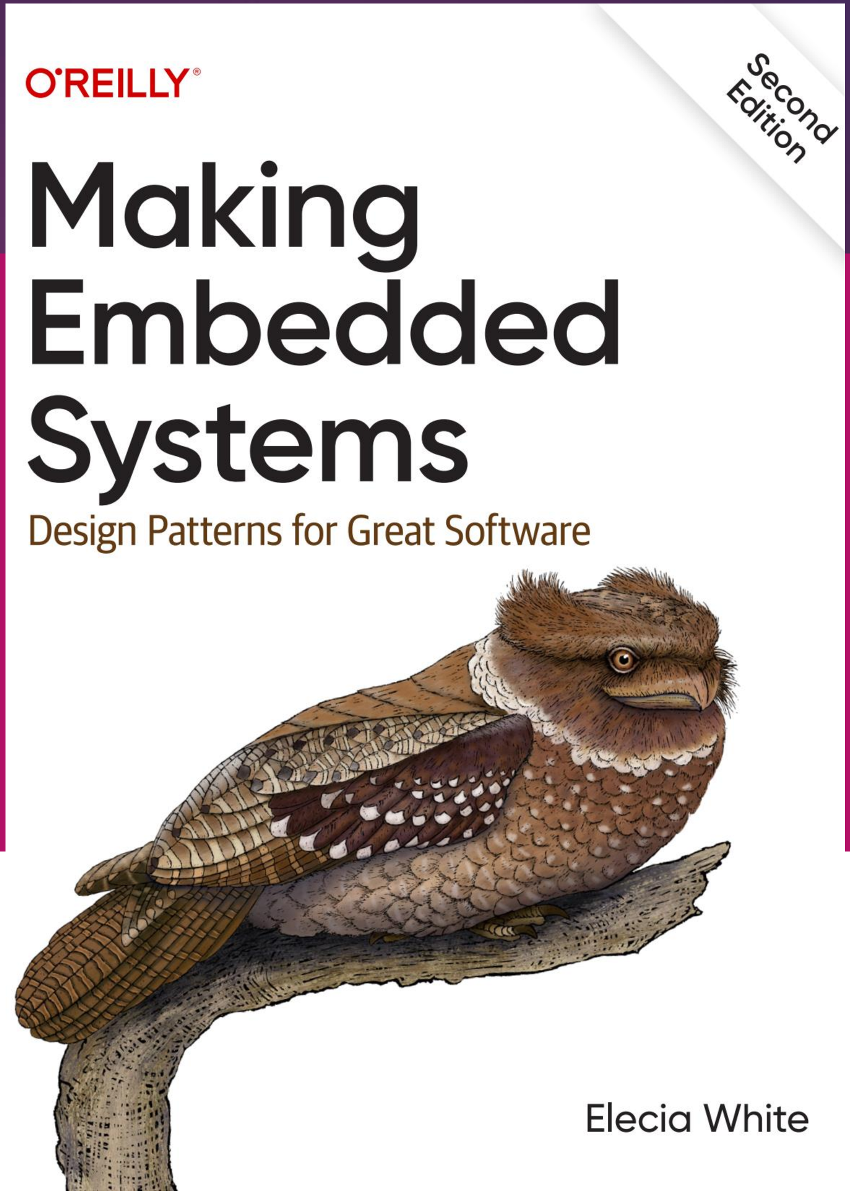


Resources

- Introduction to Hard Faults:
  - Debugging Hard Faults on ARM Cortex-M | MCU on Eclipse
  - STM32 Hard Fault debugging
- First handler shown from FreeRTOS: Debugging and diagnosing hard faults on ARM Cortex-M CPUs
- ❤️ Second handler shown from Memfault: How to debug a HardFault on an ARM Cortex-M MCU | Interrupt (this is the most in-depth resource)
- 💻 Code used in the demo: https://github.com/eleciawhite/making-embedded-systems/blob/main/Ch09_Debugging/

- Arm Documentation: Configurable Fault Status Register - Cortex-M3 and Cortex -M4 Devices Generic User Guide
- Adding NULL identification: Setting up the Cortex-M3/4 (ARMv7-M) Memory Protection Unit (MPU) - Sticky Bits
- 💲 Smashing the Stack for Fun and Profit describes how to manipulate the stack
- 🗺️ Buried Treasure and Map Files (and Linker Files) talk and map: https://embedded.fm/blog/mapfiles

Thank you!



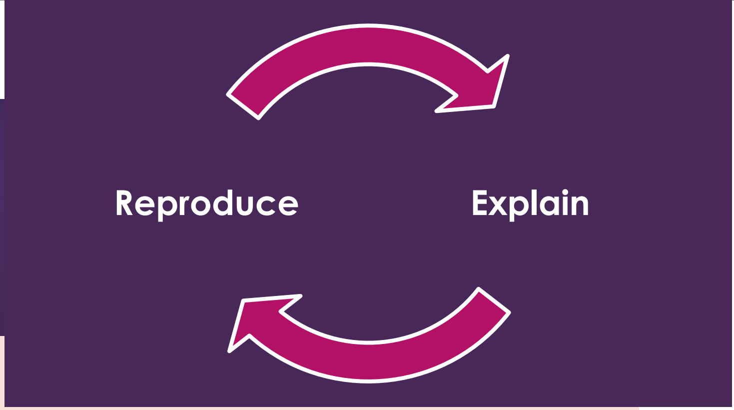
Elecia White
Logical Elegance, Inc.

THANK YOU

Embedded
Online
Conference

www.embeddedonlineconference.com

Debugging Impossible Bugs



Reproduce the bug

1. Reproduce the bug.
2. Add instrumentation.
3. Reproduce the bug with the minimal amount of code.
4. Go back to the last working version, incrementally add changes.
5. Reproduce the bug on command (or on boot) so you don't have to do ten steps to reproduce it. Remove steps if possible.
6. After making exploratory tweaks, are you sure you are running the code you think you are?

Explain the bug

1. Describe the bug's symptoms.
2. Explain how the bug could possibly do what it does. List all possible causes (without using ground loops).
3. Determine how to eliminate each cause, starting with the simplest.
4. Explain the bug and what you've tried to a duck. If you realize you forgot to try something, note it down.
5. Take notes on what changes have what effect. You will not remember after the third tweak.
6. Get a code review.