

Interrupts (resumed)

Last time

Exceptional control flow
(low-level mechanisms)

Today

Setup/enable interrupts as client

Design of interrupt module

Encapsulate details inside

Client interface that is safe, convenient, flexible

Coordination of activity

Exceptional and non-exceptional code, dispatch to multiple handlers

Data sharing, writing code that can be safely interrupted





Looking ahead

Next week (last "regular" week) Lab 7, Assign 7

Polishing touches and crossing the finish line

Nab that complete system bonus!

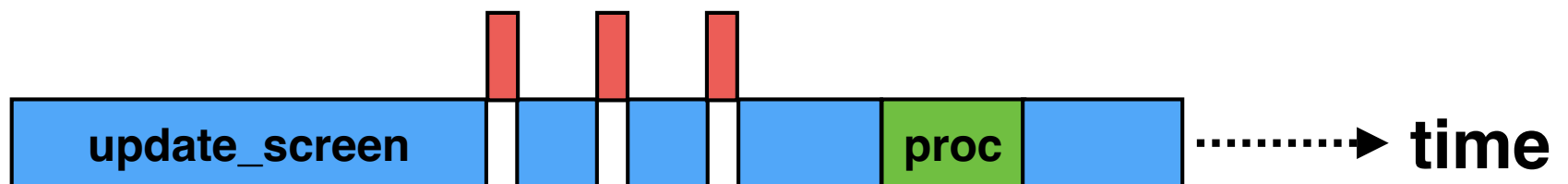
Brainstorm project ideas and begin form teams

Labs 8 & 9 are project team meetings

Interrupts Motivation: Concurrency

```
when a scan code arrives {  
    add_scan_code_to_buffer();  
}
```

```
while (1) {  
    while (read_chars_from_buffer()) {}  
    update_screen();  
}
```



Interrupts, Redux

Cause processor to pause what it's doing and immediately execute interrupt code, returning to original code when done

Causes for interrupts

- External events (reset, timer, GPIO)
- Internal events (bad memory access, bad instruction)
 - *Sometimes called "exceptions;" different in that they imply code has to do something about the instruction that was interrupted*

Interrupts are extremely valuable but getting them right requires using everything you've learned: assembly, linking, C, memory

What Happens on an Interrupt

Processor switches to interrupt mode

- A mode where you get a few private registers (sp, lr, and CSPR)
- Processor remembers where your program was interrupted (lr)
- The processor starts executing from one of 8 predefined addresses (vectors)
 - *8 different interrupt cases (we only care about one)*
 - *Processor expects table of instruction (one per interrupt) starting at address 0x0*

What your software needs to do

- Define that table of instructions (one per interrupt) at address 0x0
 - *Make sure the can correctly call functions anywhere*
- Initialize an interrupt stack
- Save r0 - r12, lr registers on new stack if you use them for interrupt processing
- The rest is regular software (call functions, etc)
- When done, restore registers, return to lr, restore supervisor mode

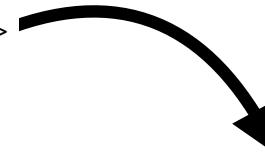
C Code

```
#define RPI_VECTOR_START 0x0
```

```
int* vectorsdst = (int*)RPI_VECTOR_START;
int* vectors = &_vectors;
int* vectors_end = &_vectors_end;
while (vectors < vectors_end)
    *vectorsdst++ = *vectors++;
```

0000807c <_vectors>:

807c:	e59ff018	ldr	pc, [pc, #24]	; 809c <abort_addr>
8080:	e59ff014	ldr	pc, [pc, #20]	; 809c <abort_addr>
8084:	e59ff010	ldr	pc, [pc, #16]	; 809c <abort_addr>
8088:	e59ff00c	ldr	pc, [pc, #12]	; 809c <abort_addr>
808c:	e59ff008	ldr	pc, [pc, #8]	; 809c <abort_addr>
8090:	e59ff004	ldr	pc, [pc, #4]	; 809c <abort_addr>
8094:	e59ff004	ldr	pc, [pc, #4]	; 80a0 <interrupt_addr>
8098:	e51ff004	ldr	pc, [pc, #-4]	; 809c <abort_addr>
809c:	000080a4	.word	0x000080a4	
80a0:	000080a8	.word	0x000080a8	



00000000 <_vectors>:

0000:	e59ff018	ldr	pc, [pc, #24]	; 809c <abort_addr>
0004:	e59ff014	ldr	pc, [pc, #20]	; 809c <abort_addr>
0008:	e59ff010	ldr	pc, [pc, #16]	; 809c <abort_addr>
000c:	e59ff00c	ldr	pc, [pc, #12]	; 809c <abort_addr>
0010:	e59ff008	ldr	pc, [pc, #8]	; 809c <abort_addr>
0014:	e59ff004	ldr	pc, [pc, #4]	; 809c <abort_addr>
0018:	e59ff004	ldr	pc, [pc, #4]	; 80a0 <interrupt_addr>
001c:	e51ff004	ldr	pc, [pc, #-4]	; 809c <abort_addr>
0020:	000080a4	.word	0x000080a4	
0024:	000080a8	.word	0x000080a8	

interrupts_asm.s

```
interrupt_asm:
    mov     sp, #0x8000
    sub     lr, lr, #4
    push    {r0-r12, lr}
    mov     r0, lr
    bl      interrupt_dispatch
    ldm     sp!, {r0-r12, pc}^
```

Today

1. Set up the interrupt stack.
2. Install interrupt handler code.
3. Tell CPU when to trigger interrupts.
 - When PS/2 clock line has a falling edge
4. Enable interrupts!
5. Writing safe interrupt handlers.
 - How do you share state that can be modified at any time?

Three Layers

1. Enable/disable a specific interrupt source

- For example, when we detect a falling clock edge on GPIO_PIN23 (PS/2 CLK)

2. Enable/disable type of interrupts

- E.g., GPIO interrupts

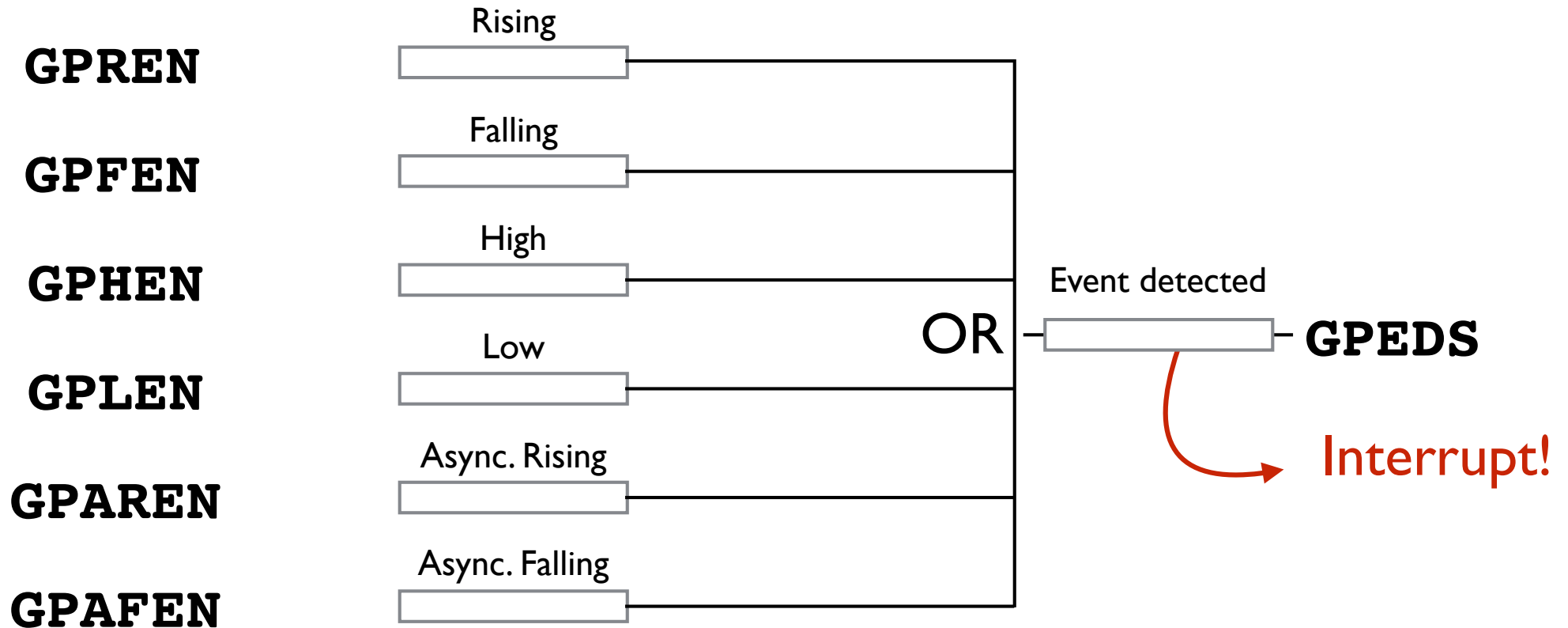
3. Global interrupt enable/disable

Interrupt fires if and only all three are enabled

Forgetting to enable one is a common bug

GPIO Events

Peripheral Registers



See `gpioextra.h` and `gpioextra.c`

Interrupt sources



BCM2835 ARM Peripherals

ARM peripherals interrupts table.

#	IRQ 0-15	#	IRQ 16-31	#	IRQ 32-47	#	IRQ 48-63
0		16		32		48	smi
1		17		33		49	gpio_int[0]
2		18		34		50	gpio_int[1]
3		19		35		51	gpio_int[2]
4		20		36		52	gpio_int[3]
5		21		37		53	i2c_int
6		22		38		54	spi_int
7						55	pwm_int
8		24		40			
9		25		41			
10		26		42			
11		27		43	i2c_spi_slv_int		
12		28		44		60	
13		29	Aux int	45	pwa0	61	
14		30		46	pwa1	62	
15		31		47		63	

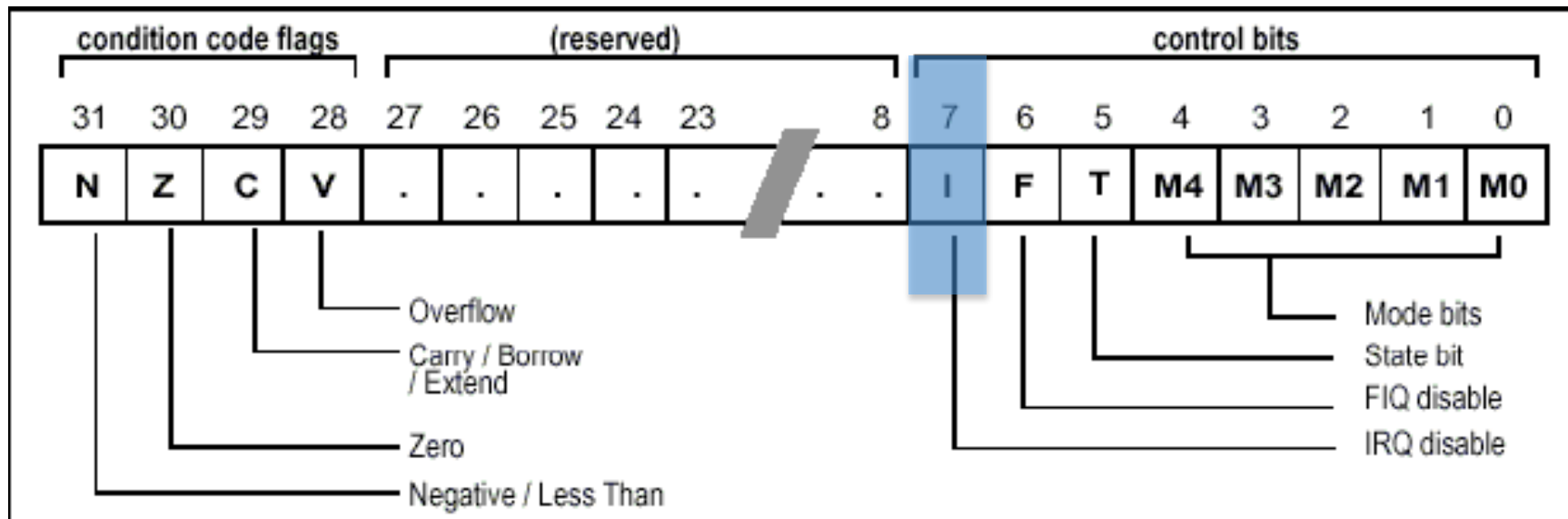
what we
want

Documentation is sparse ...

gpio_int[0] for BANK0 (pins 0-27)
gpio_int[1] for BANK1 (pins 28-45)
gpio_int[2] for BANK2 (pins 46-53)
gpio_int[3] for all the pins

The table above has many empty entries. These should not be enabled as they will interfere with the GPU operation.

Enabling Global Interrupts



```
.global interrupts_global_enable
interrupts_global_enable:
    mrs r0,cpsr
    bic r0,r0,#0x80
    // I=0 enables interrupts
    msr cpsr_c,r0
    bx lr
```

```
.global interrupts_global_disable
interrupts_global_disable:
    mrs r0,cpsr
    orr r0,r0,#0x80
    // I=1 disables interrupts
    msr cpsr_c,r0
    bx lr
```

Gpio events

GPIO event registers, detect event per-pin

- Event types: falling edge, rising edge, high level, ...
 - **gpio_enable_event_detection(pin, event)**
- Event occurs, turns on bit, check bit to see if event occurred
 - ***gpio_check_event(pin)***
- Must clear bit to process, if not, interrupt will keep re-triggering
 - ***gpio_clear_event(pin)***

References

- P. 96-99 in BCM2835 ARM Peripherals doc
- Review our code in **\$CS107E/src/gpio_extra.c**

Armtimer events

Initialize timer

- **armtimer_init(unsigned int nticks)**
- alarm period will countdown **nticks** (microseconds)

Enable

- **armtimer_enable()** starts timer running
- **armtimer_enable_interrupts()** will generate interrupt at end of period

Status, check, clear

- **armtimer_check_and_clear_interrupt()**

References

- P. 196 in BCM2835 ARM Peripherals doc
- Review our code in **\$CS107E/src/armtimer.c**

code/button-interrupts

We're done!

We now can write correct and safe interrupt code

- Assembly to save all registers
- Call into C code
- Assembly to restore registers, return to interrupted code

We can install the interrupt code table to 0x0

- Embed addresses of assembly routines so jumps are absolute
- Copy interrupt table to 0x0 in cstart

Enable and disable interrupts

- Specific interrupts, per-peripheral interrupts, global interrupts

Not Quite

Need to be able to specify what code to run

- Interrupts are bottom-up
- The library implements the calling function: we implement the handlers that should run on different interrupts
- Need API to be able to do this

Need to write code that can be safely interrupted

- Interrupt handler may put a PS/2 scan code in a buffer
- Could do so at any time: in the middle of when main() code is trying to pull a scan code out of the buffer
- Need to make sure the interrupt doesn't corrupt the buffer

Implementation

Interrupt peripheral, one bit per interrupt source

```
enum interrupt_source {
    INTERRUPTS_AUX           = 29,
    INTERRUPTS_I2CSPISLV    = 43,
    INTERRUPTS_PWA0          = 45,
    INTERRUPTS_PWA1          = 46,
    INTERRUPTS_CPR           = 47,
    INTERRUPTS_SMI           = 48,
    INTERRUPTS_GPIO0         = 49,
    INTERRUPTS_GPIO1         = 50,
    INTERRUPTS_GPIO2         = 51,
    INTERRUPTS_GPIO3         = 52,
    ...
}
```

enable

00000000000000000000000000000000	00000000000000000000000000000000
----------------------------------	----------------------------------

pending

000000000000000000000000000000000000	000000000000000000000000000000000000
--------------------------------------	--------------------------------------

Store array of handlers (function pointers)
mirrors structure of peripheral

handlers

•	•	•	•	•	•	•	•	buzz()	...	•	•	•	•
---	---	---	---	---	---	---	---	--------	-----	---	---	---	---

```
vectors[IRQ] = interrupt_asm
```

interrupt_asm:

```

mov    sp, #0x8000
sub    lr, lr, #4
push   {r0-r12, lr}
mov    r0, lr
bl     interrupt_dispatch
ldm    sp!, {r0-r12, pc}^

```

Dispatch to handler

```
void interrupt_dispatch(unsigned int pc) {
    int source = get_next_source();
    handlers[source].fn(pc,
        handlers[source].data);
}
```

Client code

```
void buzz(unsigned int pc,
          void *data) {
    armtimer_clear_event();
    play_sound();
}
```

Register handler

Client registers handler for a specific interrupt source

- A handler is a function pointer

Array of function pointers, one per interrupt source

- Interrupt source number is index into array

When interrupt occurs, dispatch identifies source

- Scan pending register, count zero bits, stop at first bit set, this is index into handler array

Aux data can be used to pass information into handler function

- If not needed, aux data can be **NULL**
- Data type is **void *** for flexibility

Review our code in **\$CS107E/src/interrupts.c**

GPIO interrupts

Single interrupt source shared by all GPIO pins/events

- Need *another* level of dispatch to support per-pin handler

gpio_interrupts_init registers a handler with top-level **interrupts** module

- All gpio events go to this handler, which in turn dispatches to client's per-pin handler

Internal structure of **gpio_interrupts** similar to top-level **interrupts**

- Array of handlers, one per pin
- Scan event detect register, count zero bits, stop at first set bit, this is index into handler array

Review our code in **\$CS107E/src/gpio_interrupts.c**

Interrupt checklist

Client must:

Event-specific

✓ Initialize interrupts (and possibly `gpio_interrupts`)

✓ Enable detection of desired event

- E.g., armtimer countdown reaches zero

✓ Write handler function to process event

- Handler acts on event and clears it

✓ Register handler with dispatcher

- `gpio_interrupts_register_handler` (if gpio event) or `interrupts_register_handler` (all others)

✓ Enable interrupt source

- `gpio_interrupts_enable` (if gpio event) or `interrupts_enable_source` (all others)

✓ Globally enable interrupts

- Throw the big switch to turn it all on when ready
- `interrupts_global_enable`

All steps essential

Fiddly code, easy to forget steps, mix up or do in wrong order

Bug symptom is absence of action, revisit checklist to find what's off

Sample client use of interrupts

```
void timer(unsigned int pc, void *aux_data) {  
    armtimer_clear_interrupt();  
    printf("T");  
}
```

```
void click(unsigned int pc, void *aux_data) {  
    gpio_clear_event(BUTTON);  
    printf("B");  
}
```

```
void main(void)  
{  
    interrupts_init();  
    armtimer_init(interval);  
    armtimer_enable_interrupts();  
    interrupts_register_handler(timer, INTERRUPTS_BASIC_ARM_TIMER_IRQ, NULL);  
    interrupts_enable_source(INTERRUPTS_BASIC_ARM_TIMER_IRQ);  
  
    gpio_interrupts_init();  
    gpio_enable_event_detection(BUTTON, GPIO_DETECT_FALLING_EDGE);  
    gpio_interrupts_register_handler(click, BUTTON, NULL);  
    gpio_interrupts_enable();  
  
    interrupts_global_enable();  
    ...  
}
```

code/interrupt_party

Not Quite

Need to be able to specify what code to run

- Interrupts are bottom-up
- The library implements the calling function: we implement the handlers that should run on different interrupts
- Need API to be able to do this

Need to write code that can be safely interrupted

- Interrupt handler may put a PS/2 scan code in a buffer
- Could do so at any time: in the middle of when main() code is trying to pull a scan code out of the buffer
- Need to make sure the interrupt doesn't corrupt the buffer

code/race

One Problem

main code

```
extern int a;
```

```
a = a + 1;
```

interrupt

```
extern int a;
```

```
a = a - 1;
```

One Problem

main code

```
extern int a;
```

```
a = a + 1;
```

interrupt

```
extern int a;
```

```
a = a - 1;
```

<inc>:

```
8000: e52db004 push {fp}
8004: e28db000 add fp, sp, #0
8008: e59f3018 ldr r3, [pc, #24]
800c: e5933000 ldr r3, [r3]
8010: e2832001 add r2, r3, #1
8014: e59f300c ldr r3, [pc, #12]
8018: e5832000 str r2, [r3]
801c: e24bd000 sub sp, fp, #0
8020: e49db004 pop {fp}
8024: e12fff1e bx lr
8028: 00010070 .word 0x00010070
```

<dec>:

```
802c: e52db004 push {fp}
8030: e28db000 add fp, sp, #0
8034: e59f3018 ldr r3, [pc, #24]
8038: e5933000 ldr r3, [r3]
803c: e2432001 sub r2, r3, #1
8040: e59f300c ldr r3, [pc, #12]
8044: e5832000 str r2, [r3]
8048: e24bd000 sub sp, fp, #0
804c: e49db004 pop {fp}
8050: e12fff1e bx lr
8054: 00010070 .word 0x00010070
```

One Problem

main code

```
extern int a;
```

```
a = a + 1;
```


interrupt

```
extern int a;
```

```
a = a - 1;
```

<inc>:

```
8000: e52db004 push {fp}
8004: e28db000 add fp, sp, #0
8008: e59f3018 ldr r3, [pc, #24]
800c: e5933000 ldr r3, [r3]
8010: e2832001 add r2, r3, #1
8014: e59f300c ldr r3, [pc, #12]
8018: e5832000 str r2, [r3]
801c: e24bd000 sub sp, fp, #0
8020: e49db004 pop {fp}
8024: e12fff1e bx lr
8028: 00010070 .word 0x00010070
```



<dec>:

```
802c: e52db004 push {fp}
8030: e28db000 add fp, sp, #0
8034: e59f3018 ldr r3, [pc, #24]
8038: e5933000 ldr r3, [r3]
803c: e2432001 sub r2, r3, #1
8040: e59f300c ldr r3, [pc, #12]
8044: e5832000 str r2, [r3]
8048: e24bd000 sub sp, fp, #0
804c: e49db004 pop {fp}
8050: e12fff1e bx lr
8054: 00010070 .word 0x00010070
```

Why will a decrement be lost if interrupt occurs here?

One Problem

main code

```
extern int a;
```

```
a = a + 1;
```


interrupt

```
extern int a;
```

```
a = a - 1;
```

<inc>:

```
8000: e52db004 push {fp}
8004: e28db000 add fp, sp, #0
8008: e59f3018 ldr r3, [pc, #24]
800c: e5933000 ldr r3, [r3]
8010: e2832001 add r2, r3, #1
8014: e59f300c ldr r3, [pc, #12]
8018: e5832000 str r2, [r3]
801c: e24bd000 sub sp, fp, #0
8020: e49db004 pop {fp}
8024: e12fff1e bx lr
8028: 00010070 .word 0x00010070
```



<dec>:

```
802c: e52db004 push {fp}
8030: e28db000 add fp, sp, #0
8034: e59f300c ldr r3, [pc, #12]
8044: e5832000 str r2, [r3]
8048: e24bd000 sub sp, fp, #0
804c: e49db004 pop {fp}
8050: e12fff1e bx lr
8054: 00010070 .word 0x00010070
```

Will volatile solve this?

Disabling Interrupts

main

interrupt handler

```
interrupts_global_disable();
```

```
a++;
```

```
b++;;
```

```
interrupts_global_enable();
```

```
a++;
```

```
b++;
```

Preemption and Safety

Very hard, lots of bugs.

You'll learn more in CS110/CS140.

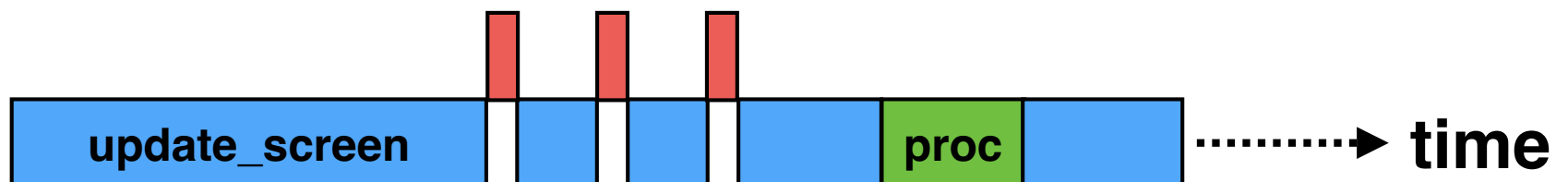
Two simple answers

1. Use simple, safe data structures
 - write once, but not always possible
2. Otherwise, temporarily disable interrupts
 - always works, but easy to forge
 - hard to compose (calling functions)

Concurrency

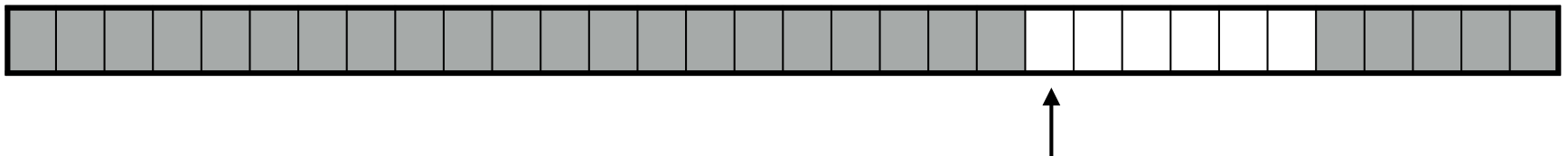
```
when a scan code arrives {  
    add_scan_code_to_buffer();  
}
```

```
while (1) {  
    while (read_chars_from_buffer()) {}  
    update_screen();  
}
```



Safe Ring Buffer

```
bool rb_enqueue(rb_t *rb, int elem) {  
    if (rb_full(rb)) {  
        return false;  
    } else {  
        rb->entries[rb->tail] = elem;  
        rb->tail = (rb->tail + 1) % LENGTH; // only writes tail  
        return true;  
    }  
}  
ringbuffer
```



```
bool rb_dequeue (rb_t *rb, int *elem) {  
    if (rb_empty(rb)) {  
        return false;  
    }  
    *elem = rb->entries[rb->head];  
    rb->head = (rb->head + 1) % LENGTH; // only writes head  
    return true;  
}
```

This Lecture

Writing the code that runs in interrupts

- Assembly code needed to change to processor models and special registers
- Interrupt table copied to 0x0 in `cstart.c`

Setting up the CPU to issue interrupts

- 3 levels: cause, type, global

Writing code that can be safely interrupted

- Race conditions though interrupt-safe ring buffer

Summary

Interrupts allow external events to preempt what's executing and run code immediately

- Needed for responsiveness, e.g., do not miss PS/2 scan codes from keyboard when drawing
- Without interrupts, most computers do nothing: they deliver keystrokes, network packets, disk reads, timers, etc.

Simple goal, but working correctly is very tricky!

- Deals with many of the hardest issues in systems

Assignment 7: update keyboard to use interrupts