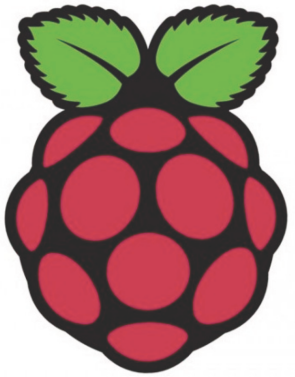
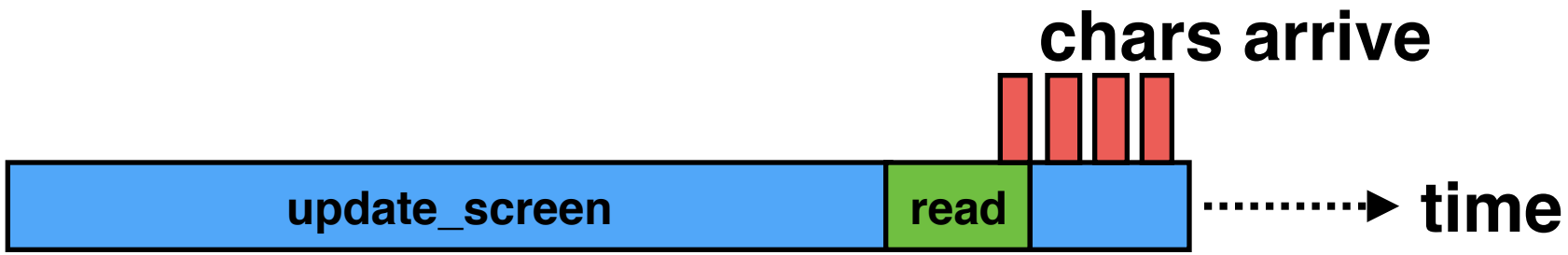
Interrupts, Part 2



now, we're cooking with gas

Blocking I/O

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



Button example

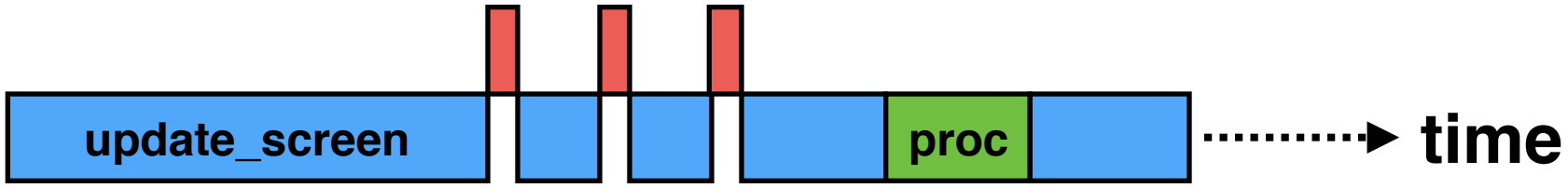
(code/blocking_button)

Keyboard example
(code/blocking_keyboard)

Concurrency

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

while (1) {
   // Doesn't block
   process_scan_codes_to_screen();
   update_screen();
}
```



Interrupts

Events that cause processor to stop what it's doing and immediately execute other code, returning to original code when done.

Examples: external events (I/O, reset, timer) as well as internal events (bad memory access, software trigger).

We Need To

1. Set up the interrupt stack.

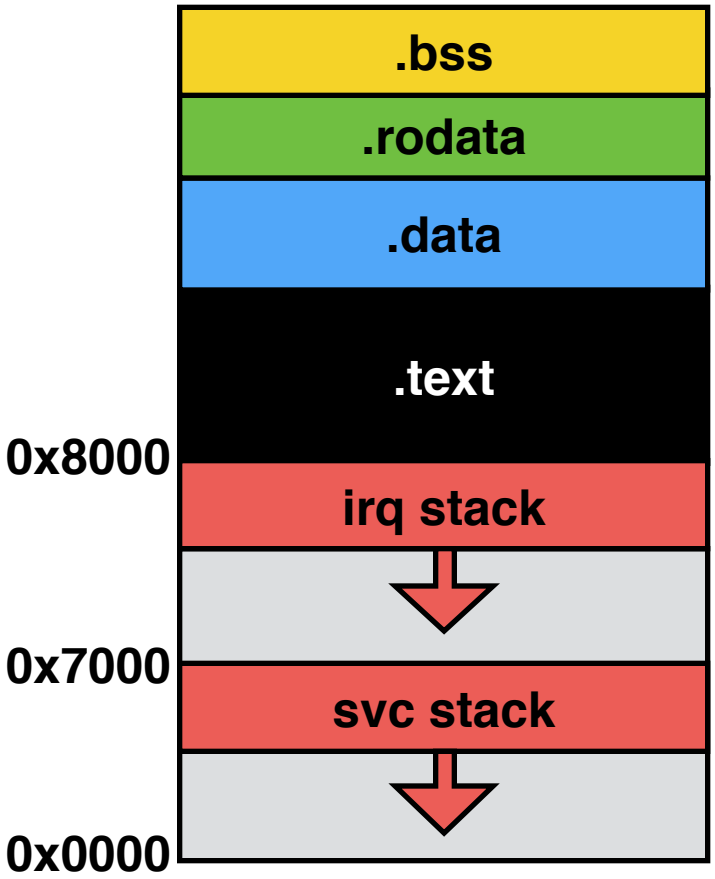
2. Write interrupt handler.
   - On clock from PS/2, read in bit, at end of byte put in buffer

3. Install interrupt handler code.

4. Tell CPU when to trigger interrupts.
   - When PS/2 clock line has a falling edge

5. Enable interrupts!

Interrupt Stack




```
_start:
    mov r0, #0xD2     @ IRQ mode
    msr cpsr_c, r0    @ Put in IRQ mode, don't clear C bits
    mov sp, #0x8000   @ Set IRQ stack pointer
    mov r0, #0xD3     @ SVC mode
    msr cpsr_c, r0    @ Put in SVC mode, don't clear C bits
    mov sp, #0x7000   @ Set SVC stack pointer
    bl _cstart        @ Jump to C start routine
```

Write Interrupt Handler

```
interrupt_asm:
  sub   lr, lr, #4          @ Have to subtract 4 from LR
  push  {lr}
  push  {r0-r12}
  mov   r0, lr              @ Pass old pc as parameter
  bl    interrupt_vector    @ C function of handler
  pop   {r0-r12}
  ldm   sp!, {pc}^          @ Pop LR to PC, restore CPSR
```

Install Handler Code

```
.globl _table

_table:
ldr pc, _reset
ldr pc, _interrupt
ldr pc, _interrupt
ldr pc, _interrupt
ldr pc, _reset
ldr pc, _reset
ldr pc, _interrupt
ldr pc, _interrupt

_interrupt:  .word interrupt_asm
_reset:      .word reset
```

Have to explicitly embed constants

10 words = 8 handlers + 2 constants

```
extern unsigned int _table[];
#define INTERRUPT_TABLE_SIZE 10
#define RPI_INTERRUPT_VECTOR_BASE 0x0

unsigned int i;
for (i = 0; i < INTERRUPT_TABLE_SIZE; i++) {
  ((unsigned int*)RPI_INTERRUPT_VECTOR_BASE)[i] = _table[i];
}
```

We Need To

1. Set up the interrupt stack.

2. Write interrupt handler.
   - On clock from PS/2, read in bit, at end of byte put in buffer

3. Install interrupt handler code.

4. **Tell CPU when to trigger interrupts.**
   - **When PS/2 clock line has a falling edge**

5. Enable interrupts!

Current Event Detection

```
while (gpio_pin_read(GPIO_PIN23) == 0) {}
while (gpio_pin_read(GPIO_PIN23) == 1) {}
// Falling edge
```

GPIO Interrupts (pg. 96-98)

Event detect status register (GPEDSn)

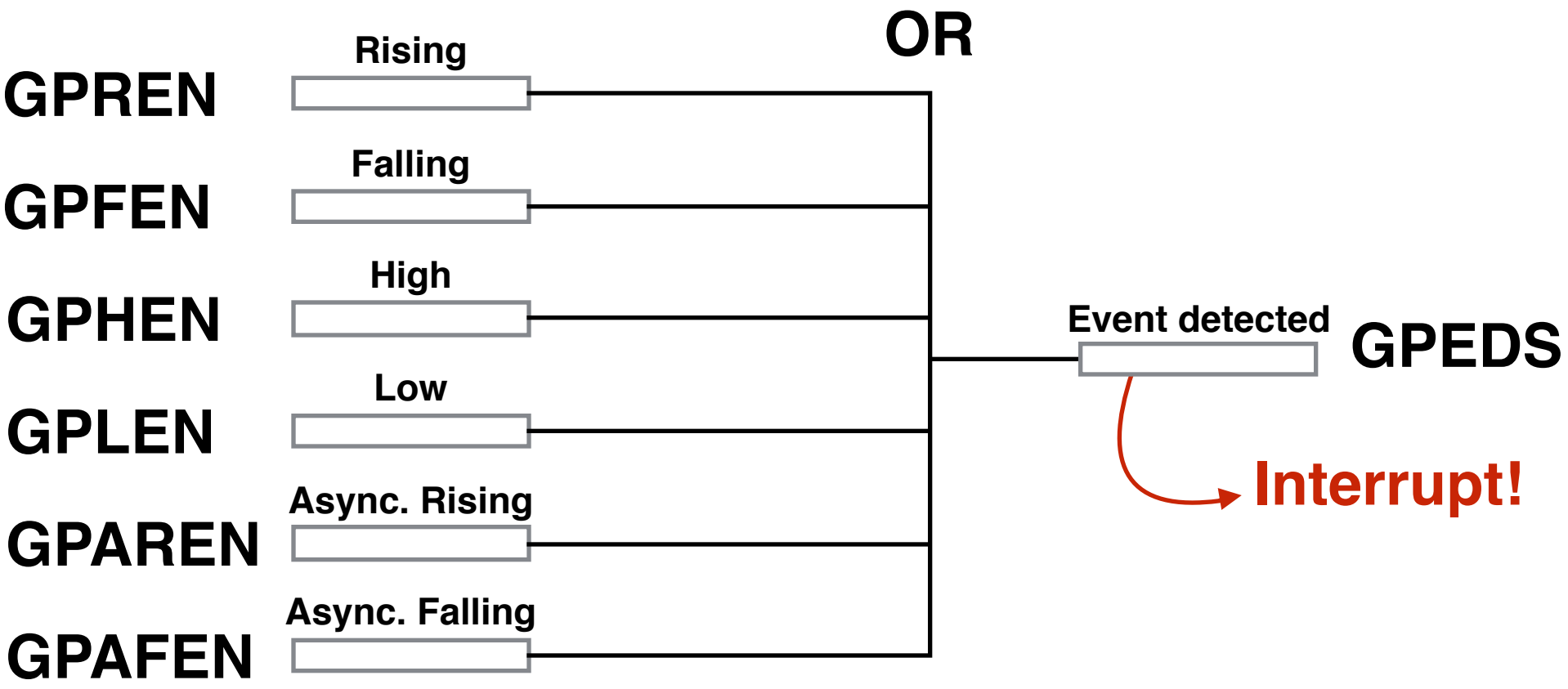
- Clear event by writing 1 to position, or will re-trigger

Falling edge detect enable register (GPFENn)

- Lots of other options! High level, low level, rising edge, etc.

Goal: Trigger interrupt on falling edge of clock, read data line in interrupt handler.

GPIO Event Detection



GPEDS demo

(code/interrupts)

We Need To

1. Set up the interrupt stack.

2. Write interrupt handler.
   - On clock from PS/2, read in bit, at end of byte put in buffer

3. Install interrupt handler code.

4. Tell CPU when to trigger interrupts.
   - When PS/2 clock line has a falling edge

5. **Enable interrupts!**

Two Parts

1. Enable/disable specific interrupts

2. Global interrupt enable/disable

Interrupt fires if and only if both are enabled

Specific Interrupts

(BCM Peripherals Manual, pages 110-113)

Register Address	Register
0x2000B200	Basic IRQ pending
0x2000B204	IRQ pending 1
0x2000B208	IRQ pending 2
0x2000B20C	FIQ Control
0x2000B210	Enable IRQs 1
0x2000B214	Enable IRQs 2
0x2000B218	Enable Basic IRQs
0x2000B21C	Disable IRQs 1
0x2000B220	Disable IRQs 2
0x2000B224	Disable Basic IRQs

BCM2835, Sec 7.5

#	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		23		39		55	pcm_int
8		24		40		56	
9		25		41		57	uart_int
10		26		42		58	
11		27		43	i2c_spi_slv_int	59	
12		28		44		60	
13		29	Aux int	45	pwa0	61	
14		30		46	pwa1	62	
15		31		47		63	

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

BCM2835, Sec 7.5

#	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		23		39		55	pcm_int
8		24		40		56	
9		25		41		57	uart_int
10		26		42		58	
11		27		43	i2c_spi_slv_int	59	
12		28		44		60	
13		29	Aux int	45	pwa0	61	
14		30		46	pwa1	62	
15		31		47		63	

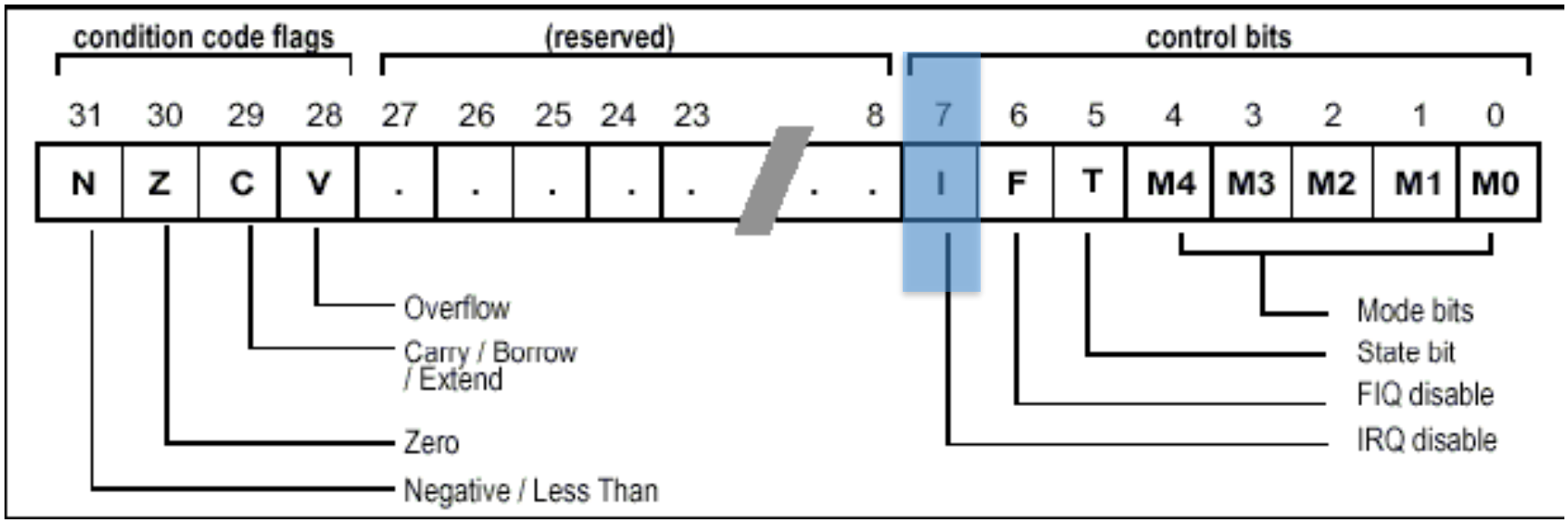
GPIO pin:	4	17	30	31	47
gpio_irq[0] (49)	Y	Y	Y	Y	N
gpio_irq[1] (50)	N	N	Y	Y	N
gpio_irq[2] (51)	N	N	N	N	Y
gpio_irq[3] (52)	Y	Y	Y	Y	Y

Enabling GPIO Interrupts

Write 0xFFFFFFFF to both disable registers

Write gpio[3] bit of enable register 2

Global Interrupts




```
void enable_interrupts() {
  asm("mrs r0, cpsr");
  asm("bic r0, r0, #0x80");
  asm("msr cpsr_c, r0");
}
```

```
void disable_interrupts() {
  asm("mrs r0, cpsr");
  asm("orr r0, r0, #0x80");
  asm("msr cpsr_c, r0");
}
```

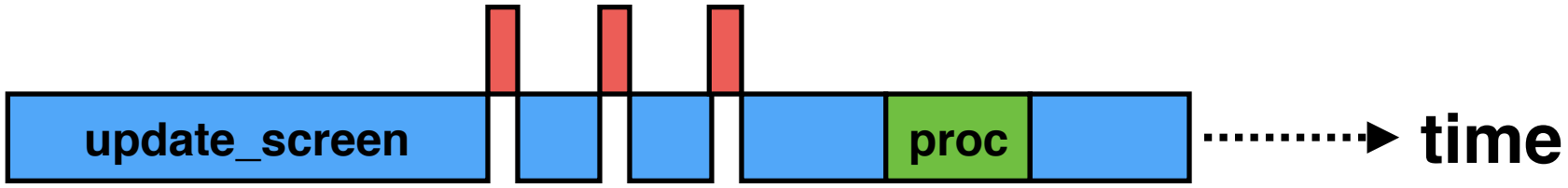
Put It All Together

(code/interrupts)

Concurrency

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

while (1) {
   // Doesn't block
   process_scan_codes_to_screen();
   update_screen();
}
```



One Problem

main code

```
extern int a;

a = a + 1;
```

```
00008000 <inc>:
    8000:   e52db004    push    {fp}        ; (str fp, [sp, #-4]!)
    8004:   e28db000    add fp, sp, #0
    8008:   e59f3018    ldr r3, [pc, #24]   ; 8028 <inc+0x28>
    800c:   e5933000    ldr r3, [r3]
    8010:   e2832001    add r2, r3, #1
    8014:   e59f300c    ldr r3, [pc, #12]   ; 8028 <inc+0x28>
    8018:   e5832000    str r2, [r3]
    801c:   e24bd000    sub sp, fp, #0
    8020:   e49db004    pop {fp}        ; (ldr fp, [sp], #4)
    8024:   e12fff1e    bx  lr
    8028:   00010070    .word   0x00010070
```

interrupt

```
extern int a;

a = a - 1;
```

```
0000802c <dec>:
    802c:   e52db004    push    {fp}        ; (str fp, [sp, #-4]!)
    8030:   e28db000    add fp, sp, #0
    8034:   e59f3018    ldr r3, [pc, #24]   ; 8054 <dec+0x28>
    8038:   e5933000    ldr r3, [r3]
    803c:   e2432001    sub r2, r3, #1
    8040:   e59f300c    ldr r3, [pc, #12]   ; 8054 <dec+0x28>
    8044:   e5832000    str r2, [r3]
    8048:   e24bd000    sub sp, fp, #0
    804c:   e49db004    pop {fp}        ; (ldr fp, [sp], #4)
    8050:   e12fff1e    bx  lr
    8054:   00010070    .word   0x00010070
```

Preemption and Safety

Very hard, many solutions, 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 forget

Disabling Interrupts

main code

```
extern int a;

disable_interrupts();
a = a + 1;
reenable_interrupts();
```

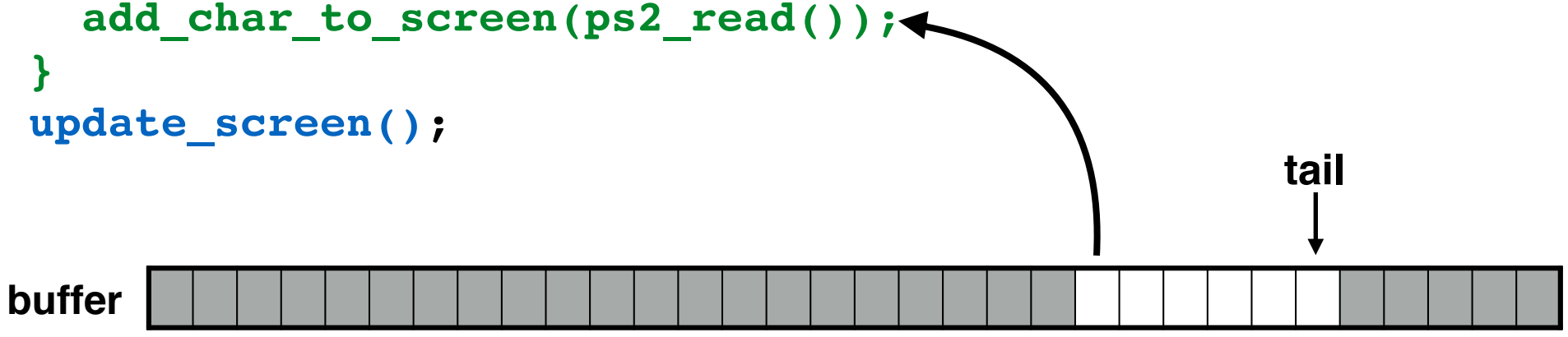
interrupt

```
extern int a;

a = a - 1;
```

PS2 Driver Software Model

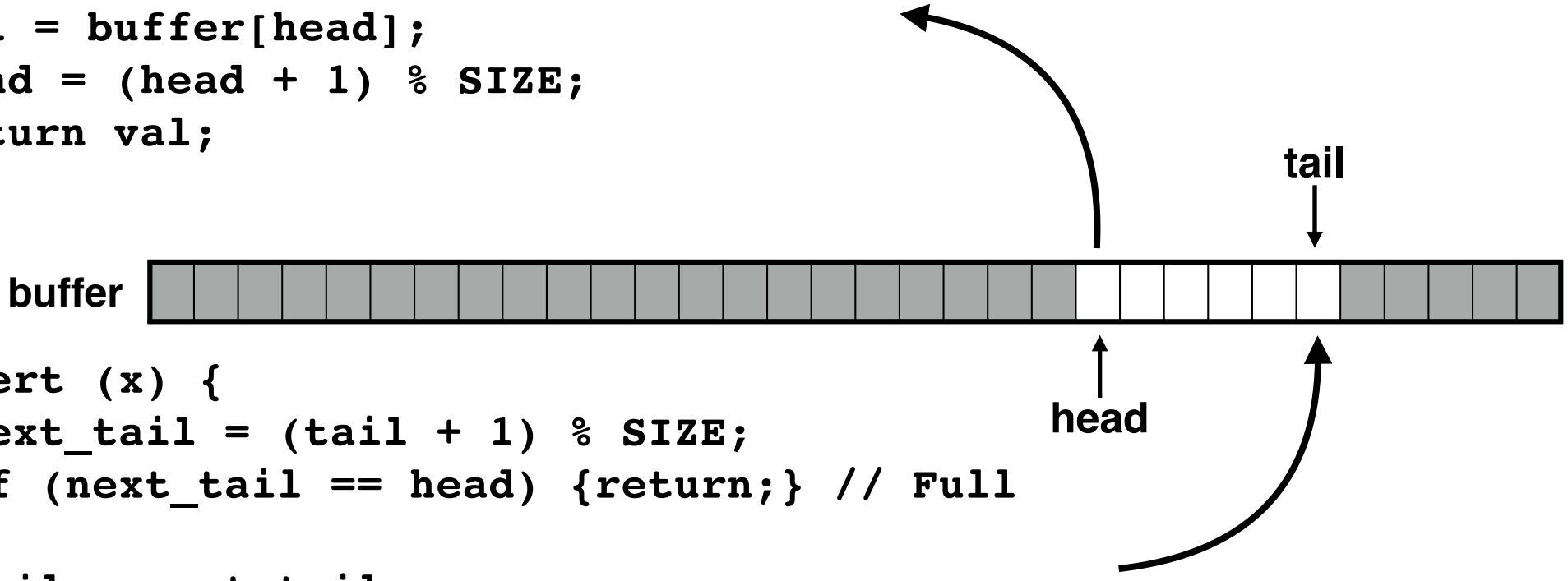
```
while (1) {
  // Doesn't block
  while (ps2_has_chars()) {
    add_char_to_screen(ps2_read());
  }
  update_screen();
}
```




```
interrupt {
  read_data_bit();
  if (code_complete) {
    buffer_add(scan_to_ascii(code));
  }
}
```

Safe Ring Buffer

```
has_elements() {
  return (tail != head);
}
remove() {
 val = buffer[head];
 head = (head + 1) % SIZE;
 return val;
}
```




```
insert (x) {
  next_tail = (tail + 1) % SIZE;
  if (next_tail == head) {return;} // Full
  …
  tail = next_tail;
}
```

Interrupts Overview

Run code in response to events or inputs.

Many can be enabled: processor interleaves events, no blocking needed.

Five steps to handle interrupts on Pi.

Preemption can lead to bugs and race conditions: be careful.