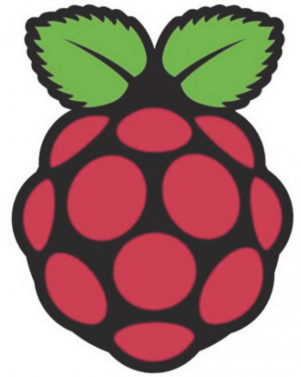




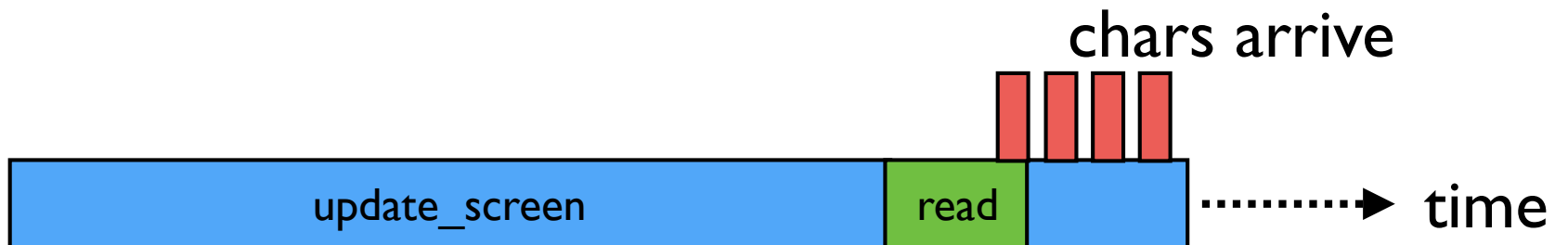
Interrupts, Part 2

now, we're cooking with gas



Blocking I/O

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



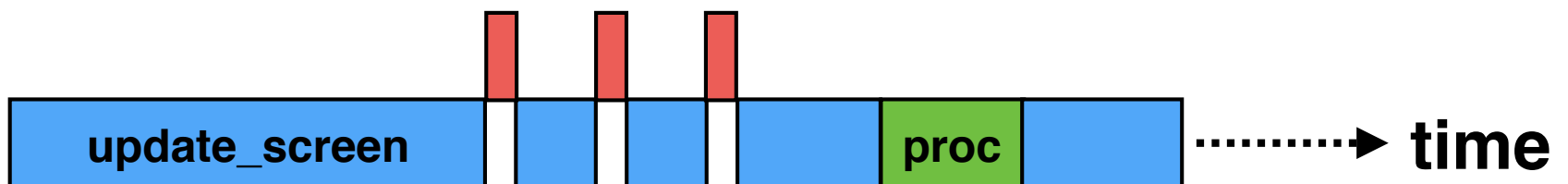
Button example

(code/button_blocking)

Concurrency

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

```
while (1) {  
    // Doesn't block  
    while (read_chars_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).

- external: “interrupts” (like PS/2 clock line!)
- internal: “exceptions”

Last Lecture

Reviewed code/`blink`

Interrupt code is stored at 0x0

- Have to copy “table” of interrupt handlers to 0x0 at boot

Interrupt code is carefully written assembly

- Need to invoke assembly routine to save CPU state
- Explicitly embed constants so they are copied too
- Interrupt mode has its own `sp` and `lr`

This Lecture

Writing the code that runs in interrupts

Setting up the CPU to issue interrupts

Writing code that can be safely interrupted

This Lecture

Writing the code that runs in interrupts

Setting up the CPU to issue interrupts

Writing code that can be safely interrupted

Interrupt Table (reminder)

8 instructions at
0x0, CPU jumps to
instr[6] when an
interrupt happens

libpi/src/vectors.s

```
.globl _vectors
```

```
_vectors:
```

```
ldr pc, _reset_asm
```

```
ldr pc, _undefined_instruction_asm
```

```
ldr pc, _software_interrupt_asm
```

```
ldr pc, _prefetch_abort_asm
```

```
ldr pc, _data_abort_asm
```

```
ldr pc, _reset_asm
```

```
ldr pc, _interrupt_asm
```

```
ldr pc, _fast_interrupt_asm
```

```
_reset_asm: .word reset_asm
```

```
_interrupt_asm: .word interrupt_asm
```

```
// ... other _asm symbols
```

```
_vectors_end:
```

Write Interrupt Handler

libpi/src/vectors.s

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 |

Processor Modes

| Register | supervisor | interrupt |
|----------|------------|-----------|
| R0 | R0 | R0 |
| R1 | R1 | R1 |
| R2 | R2 | R2 |
| R3 | R3 | R3 |
| R4 | R4 | R4 |
| R5 | R5 | R5 |
| R6 | R6 | R6 |
| R7 | R7 | R7 |
| R8 | R8 | R8 |
| R9 | R9 | R9 |
| R10 | R10 | R10 |
| fp | R11 | R11 |
| ip | R12 | R12 |
| sp | R13_svc | R13_irq |
| lr | R14_svc | R14_irq |
| pc | R15 | R15 |
| CPSR | CPSR | CPSR |
| SPSR | SPSR | SPSR |

| Modes | | | | | | |
|------------------|--------|------------|----------|-----------|-----------|----------------|
| Privileged modes | | | | | | |
| Exception modes | | | | | | |
| User | System | Supervisor | Abort | Undefined | Interrupt | Fast interrupt |
| R0 | R0 | R0 | R0 | R0 | R0 | R0 |
| R1 | R1 | R1 | R1 | R1 | R1 | R1 |
| R2 | R2 | R2 | R2 | R2 | R2 | R2 |
| R3 | R3 | R3 | R3 | R3 | R3 | R3 |
| R4 | R4 | R4 | R4 | R4 | R4 | R4 |
| R5 | R5 | R5 | R5 | R5 | R5 | R5 |
| R6 | R6 | R6 | R6 | R6 | R6 | R6 |
| R7 | R7 | R7 | R7 | R7 | R7 | R7 |
| R8 | R8 | R8 | R8 | R8 | R8 | R8_fiq |
| R9 | R9 | R9 | R9 | R9 | R9 | R9_fiq |
| R10 | R10 | R10 | R10 | R10 | R10 | R10_fiq |
| R11 | R11 | R11 | R11 | R11 | R11 | R11_fiq |
| R12 | R12 | R12 | R12 | R12 | R12 | R12_fiq |
| R13 | R13 | R13_svc | R13_abt | R13_und | R13_irq | R13_fiq |
| R14 | R14 | R14_svc | R14_abt | R14_und | R14_irq | R14_fiq |
| PC | PC | PC | PC | PC | PC | PC |
| CPSR | CPSR | CPSR | CPSR | CPSR | CPSR | CPSR |
| | | SPSR_svc | SPSR_abt | SPSR_und | SPSR_irq | SPSR_fiq |


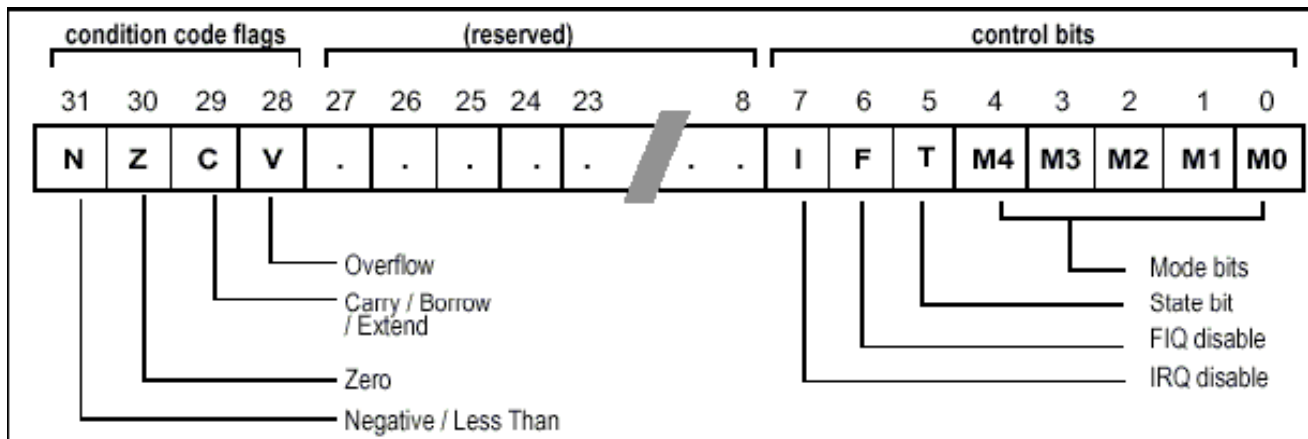
 indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

Figure A2-1 Register organization

CPSR



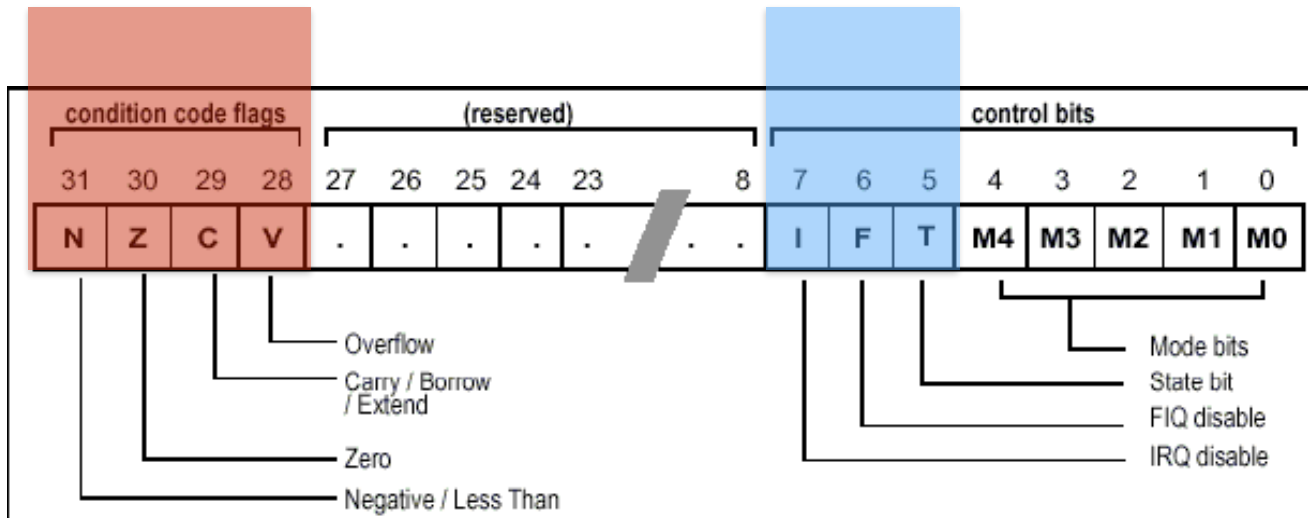
| M[4:0] | Mode |
|--------|------------|
| b10000 | User |
| b10001 | FIQ |
| b10010 | IRQ |
| b10011 | Supervisor |
| b10111 | Abort |
| b11011 | Undefined |
| b11111 | System |

| | | |
|------------------|-------------------------|---------------------------------------|
| <code>msr</code> | <code>psr, Rm</code> | <code><- Store Rd into psr</code> |
| <code>mrs</code> | <code>Rd, psr</code> | <code><- Load Rd with psr</code> |
| | | |
| <code>msr</code> | <code>cpsr_c, r0</code> | <code><- Store CPSR with r0</code> |
| <code>mrs</code> | <code>r0, cpsr_c</code> | <code><- Load r0 with CPSR</code> |

CPSR

don't touch these
bits (cpsr_c)

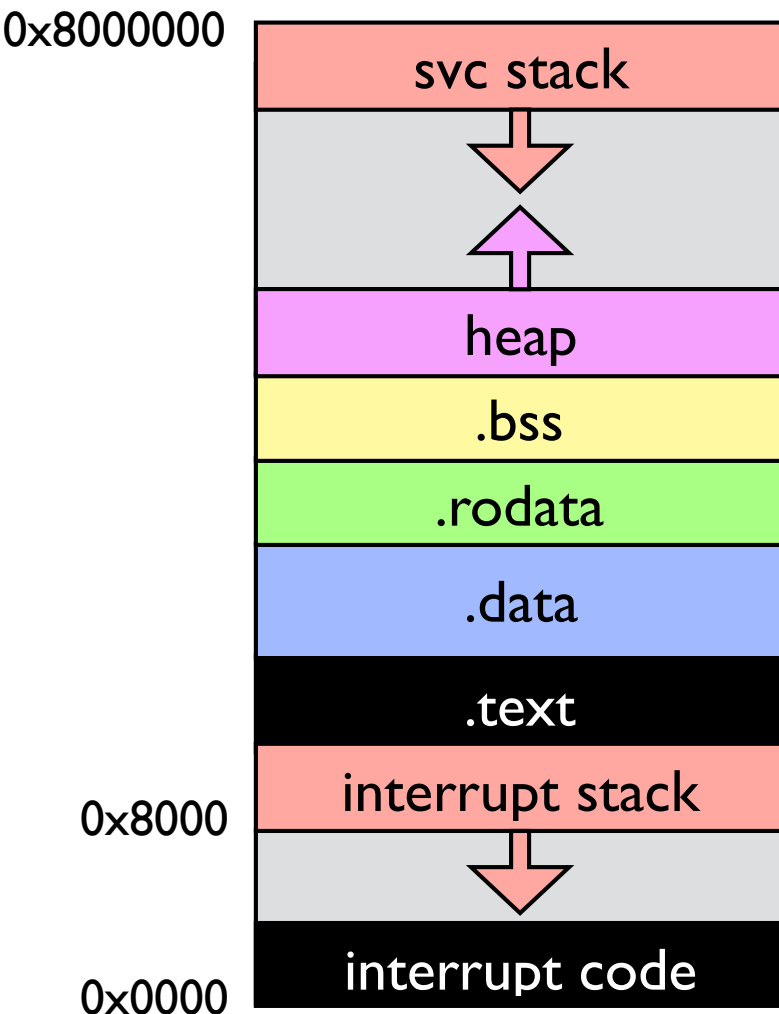
these bits should be 0b110



| M[4:0] | Mode |
|--------|------------|
| b10000 | User |
| b10001 | FIQ |
| b10010 | IRQ |
| b10011 | Supervisor |
| b10111 | Abort |
| b11011 | Undefined |
| b11111 | System |

| | | |
|------------------|-------------------------|---------------------------------------|
| <code>msr</code> | <code>psr, Rm</code> | <code><- Store Rd into psr</code> |
| <code>mrs</code> | <code>Rd, psr</code> | <code><- Load Rd with psr</code> |
| | | |
| <code>msr</code> | <code>cpsr_c, r0</code> | <code><- Store CPSR with r0</code> |
| <code>mrs</code> | <code>r0, cpsr_c</code> | <code><- Load r0 with CPSR</code> |

Set up Interrupt Stack



start.s

```
_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, #0x80000000    @ Set SVC stack pointer
    bl _cstart             @ Jump to C start routine
```

This Lecture

Writing the code that runs in interrupts

Setting up the CPU to issue interrupts

Writing code that can be safely interrupted

Three steps

1. Turn on a specific interrupt source

- E.g., when we detect a falling clock edge on PS/2 clock line

2. Enable/disable specific interrupts

- E.g., GPIO interrupts

3. Global interrupt enable/disable

Interrupt fires if and only all three are enabled

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

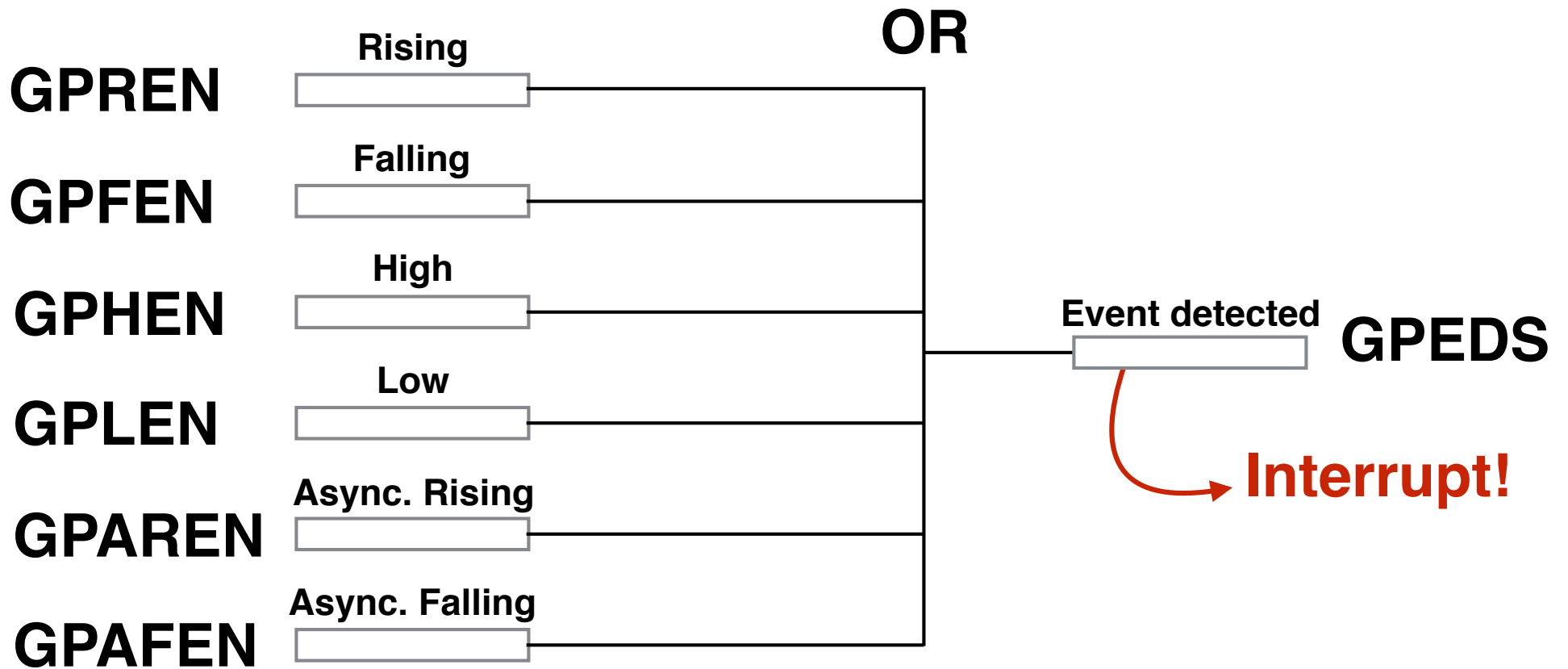
Falling edge detect enable register (GPFENn)

- 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.

```
libpi/include/gpioevent.h
```

GPIO Event Detection



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

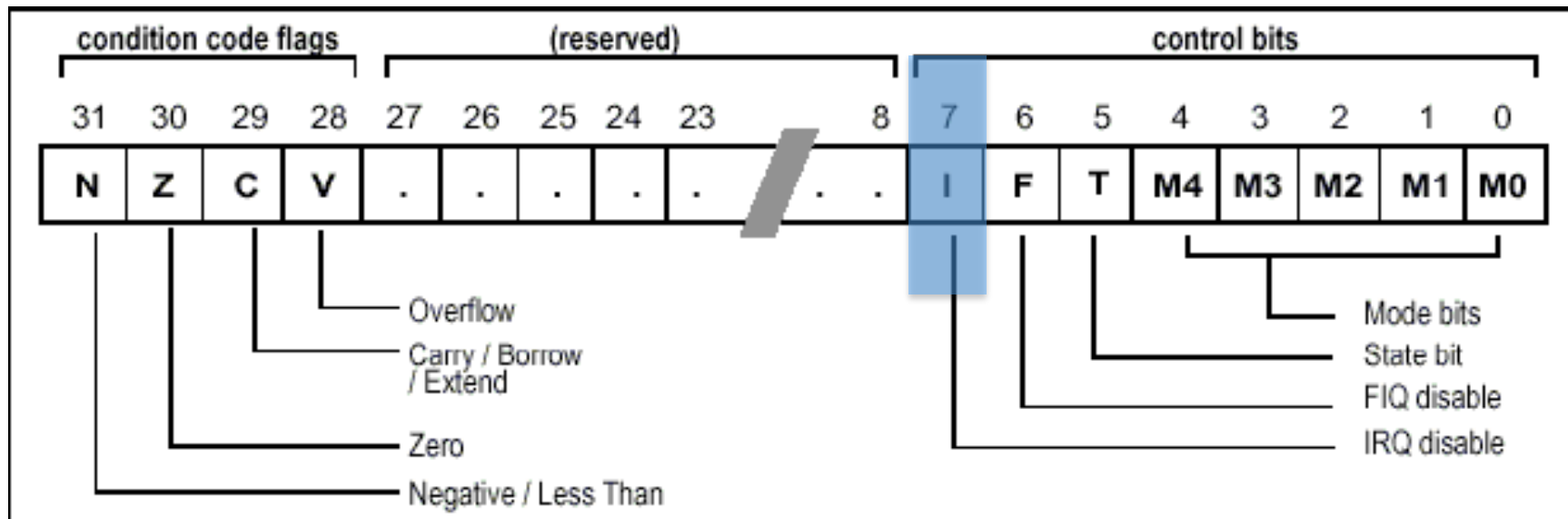
Write 0xFFFFFFFF to both disable registers

Write gpio[3] bit of enable register 2

libpi/src/interrupts.c

```
interrupts_enable(INTERRUPTS_GPIO3);
```

Global Interrupts



```
system_enable_interrupts();
```

libpi/src/register.s

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

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


GPEDS demo

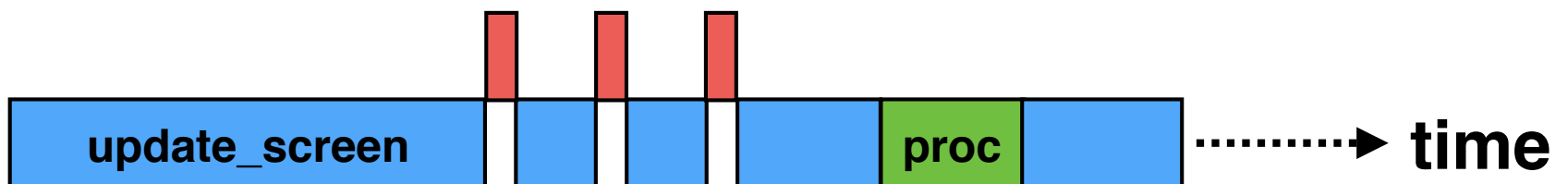
(code/button-interrupts)

We're done!

Concurrency

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

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



Not Quite

An interrupt can fire at any time

- Interrupt handler may put a PS/2 scan code in a buffer
- Could do so 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

Need to write code that can be safely interrupted

This Lecture

Writing the code that runs in interrupts

Setting up the CPU to issue interrupts

Writing code that can be safely interrupted

One Problem

main code

```
extern int a;
```

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

interrupt

```
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
```

danger

```
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 CSI 10/CSI 40.

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

Race condition demo

(code / race)

Disabling Interrupts

main code

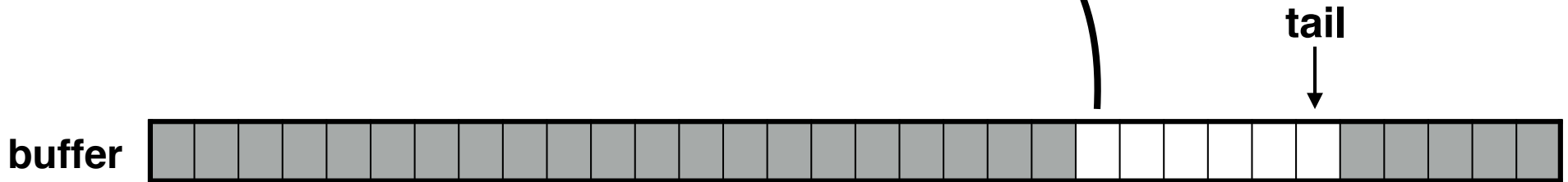
```
extern int a;  
  
disable_interrupts();  
a = a + 1;  
reenable_interrupts();
```

interrupt

```
extern int a;  
  
a = a - 1;
```

PS2 Driver Software Model

```
keyboard_read_event() {  
    while (buffer empty) {}  
    sc = read_scancode_from_buffer();  
}
```



```
interrupt {  
    read_data_bit();  
    if (code_complete) {  
        buffer_add(scancode);  
    }  
}
```

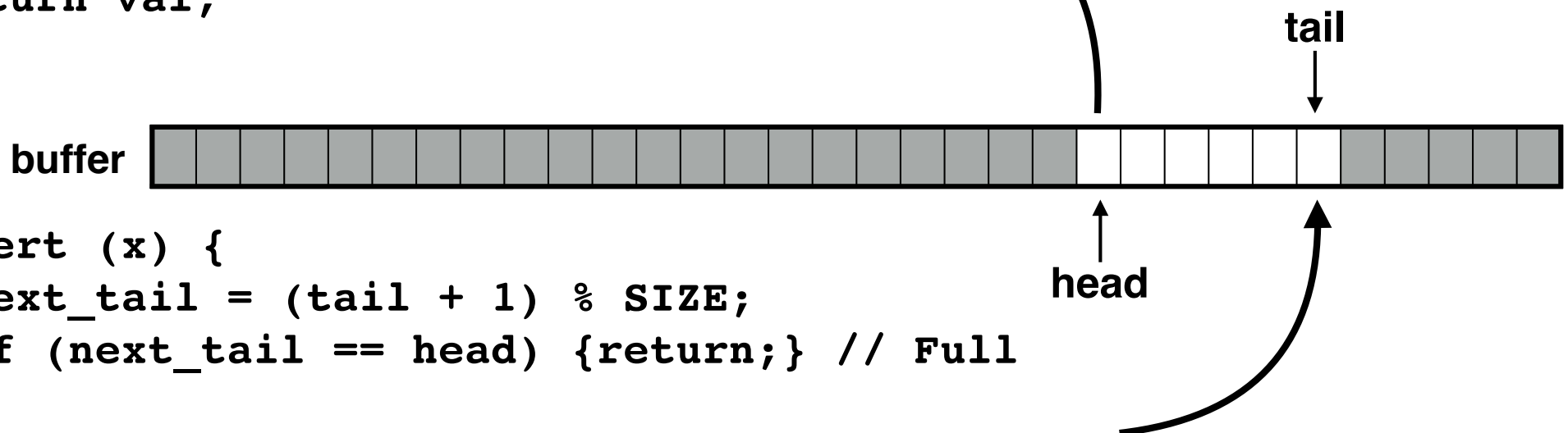
Safe Ring Buffer

libpi/src/circular.c

```
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;  
}
```



This Lecture

Writing the code that runs in interrupts

- Some assembly code due to processor state
- Interrupt table copied to 0x0 on boot

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., not missing PS/2 scan codes

Simple goal, but working correctly is very tricky!

- Deals with many of the hardest issues in systems

Assignment 7: update keyboard to use interrupts

- Each clock edge is an interrupt; after 11 interrupts, push scan code into a ring buffer; keyboard driver pulls scan codes from ring buffer

Keyboard with interrupts

(code/shell)