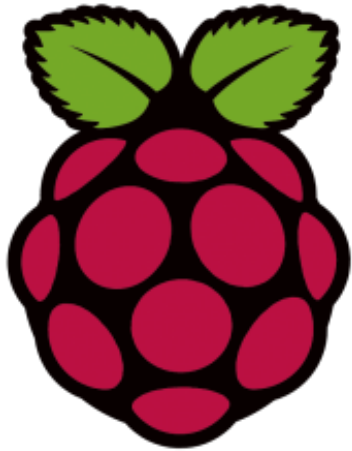
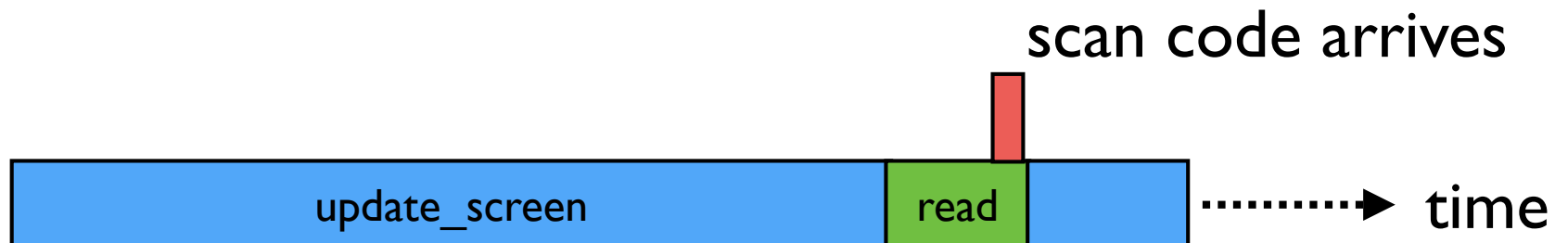


# Interrupts and Concurrency



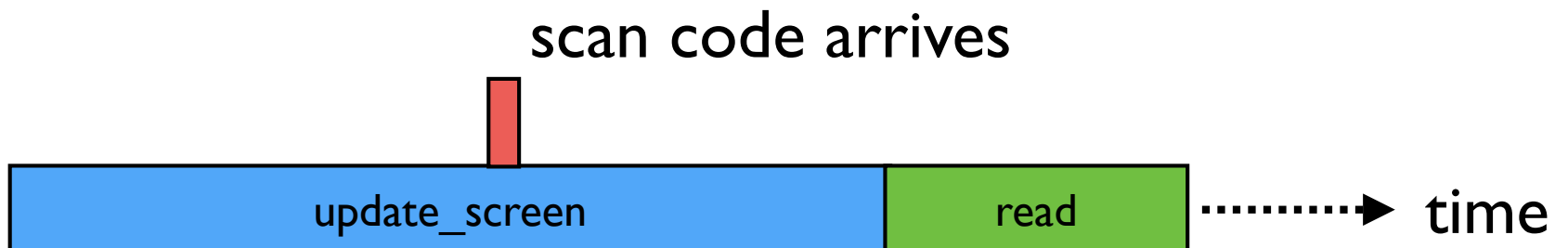
# Blocking I/O

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



# Problem!

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



**code/console**

# Interrupts, Redux

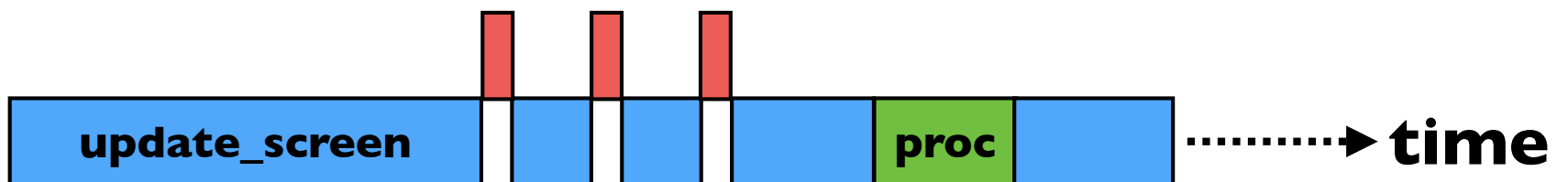
Cause processor to pause what it's doing and immediately execute interrupt code, returning to original code when done

- 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*

# Concurrency

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

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



# Last Lecture

8 different interrupts (we only care about one)

Processor specifies location (0x0) where it expects table of instructions, one per interrupt

- When an interrupt occurs, processor jumps to the corresponding instruction
- At that point, everything is software's responsibility

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

# Challenge

Processor specifies location (0x0) where it expects table of instructions, one per interrupt

- When an interrupt occurs, processor jumps to the corresponding instruction
- At that point, everything is software's responsibility

Writing code that can be safely copied to this location requires very careful assembly

- Have to make sure addresses are all absolute, and you know where they are stored

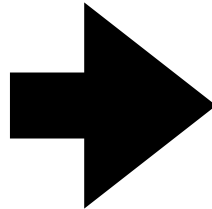


# Explicitly Embedded Absolute Addresses

```
.globl _vectors
```

```
_vectors:
```

```
ldr pc, =abort_asm  
ldr pc, =abort_asm  
ldr pc, =abort_asm  
ldr pc, =abort_asm  
ldr pc, =abort_asm  
ldr pc, =abort_asm  
ldr pc, =interrupt_asm  
ldr pc, =abort_asm
```



```
.globl _vectors
```

```
_vectors:
```

```
ldr pc, =_abort_asm  
ldr pc, =_abort_asm  
ldr pc, =_abort_asm  
ldr pc, =_abort_asm  
ldr pc, =_abort_asm  
ldr pc, =_abort_asm  
ldr pc, =_abort_asm  
ldr pc, =_interrupt_asm  
ldr pc, =_abort_asm
```


```
_abort_asm:          .word abort_asm  
_interrupt_asm:      .word interrupt_asm
```

Now we know the constants will follow the code.  
This works!!!

# Explicitly Embedded Absolute Addresses

8 instructions starting  
at 0x0

CPU jumps to instr[6]  
on a peripheral  
interrupt



```
.globl _vectors
```

```
_vectors:
```

```
ldr pc, =_abort_asm
```

```
ldr pc, =_abort_asm
```

```
ldr pc, =_abort_asm
```

```
ldr pc, =_abort_asm
```

```
ldr pc, =_abort_asm
```

```
ldr pc, =_abort_asm
```

```
ldr pc, =_interrupt_asm
```

```
ldr pc, =_abort_asm
```

```
_abort_asm: .word abort_asm
```

```
_interrupt_asm: .word interrupt_asm
```

Now we know the constants will follow the code.  
This works!!!

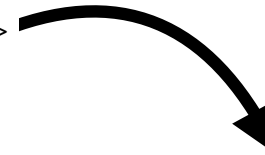
# C Code

```
#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}^
```

# 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}^
```

AMAZING INSTRUCTION, it does all of this:

Pop 14 registers from the stack

Store them in r0-r12 and pc

Increase the stack pointer by the amount popped

Restore the CPSR from the SPSR (only works on ldm with sp)

# 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*

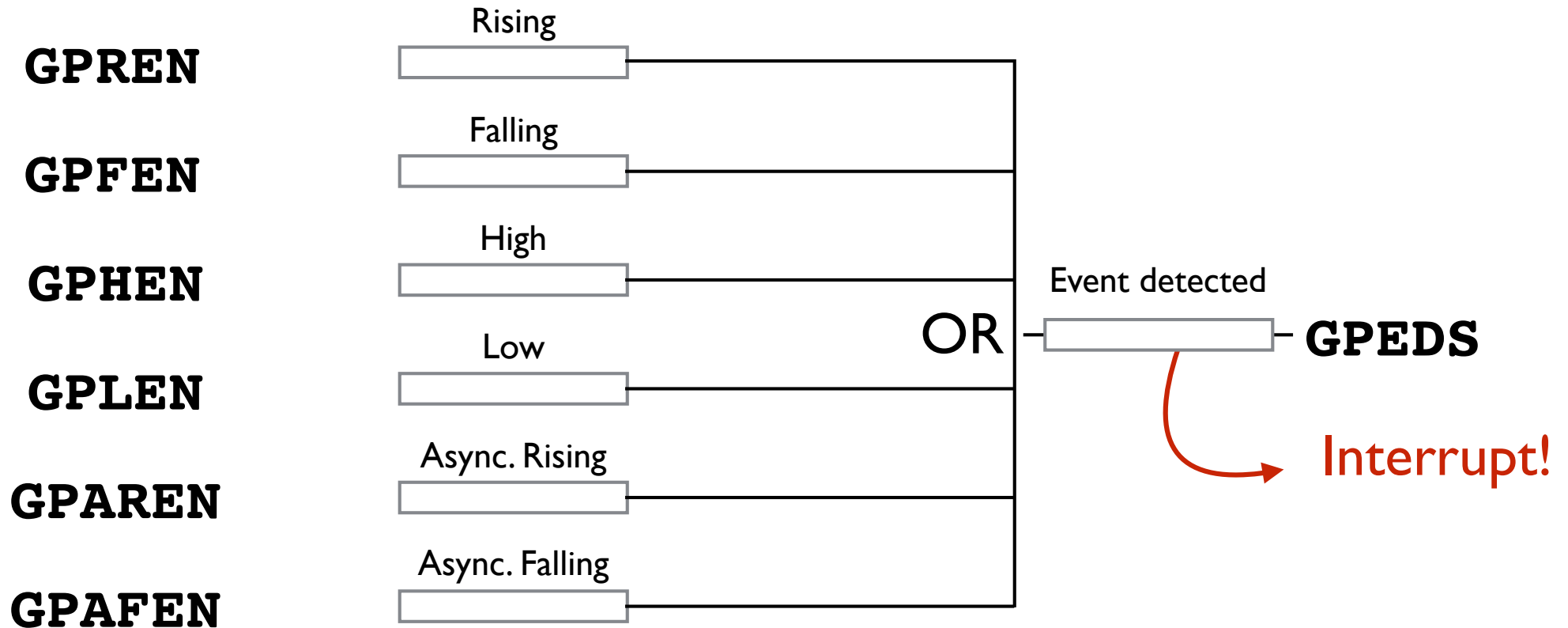
# **armtimer/timerblink.c**

The ARM timer can generate interrupts when it reaches a certain number of timer ticks: this is how a full OS can decide when to share the processor between multiple programs. We can also use it to blink an LED!



# GPIO Events

Peripheral Registers



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

# **GPIO Interrupts (pg. 96-98)**

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

Falling edge detect enable register (GPFENn)

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

Event detect status register (GPEDSn)

- Bit is set when an event on the given pin occurs
- Clear event by writing 1 to position, or will re-trigger an interrupt!

# 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	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	

*Documentation is sparse ...*

what we  
want

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

# 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	

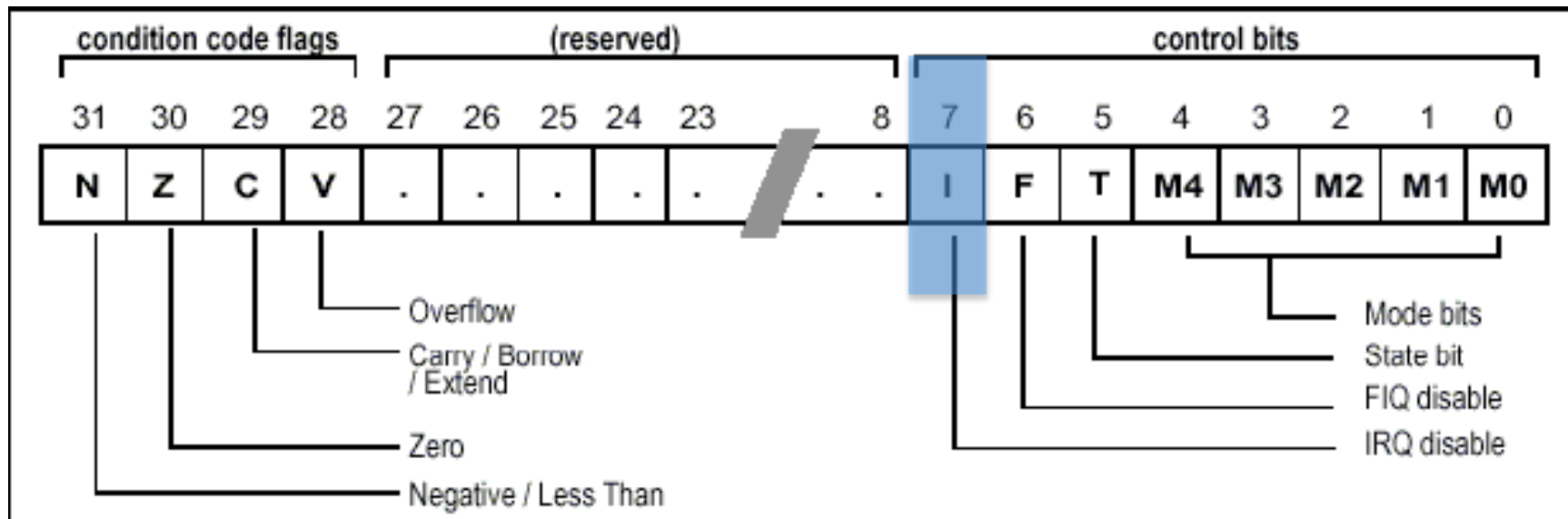
what we  
want

*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
```

**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



# Interrupt API Goals

## Speed!

- Minimize number of cycles spent in library: imagine if the handler is just a few instructions because we want to be able to run it very often

## Avoid runtime failures (i.e., debugging)

- You should always be able to register one handler for each interrupt: whether you can register it should be independent of how many modules there are

## Flexible

- Allow you to easily add new modules that handle interrupts

# Basic API

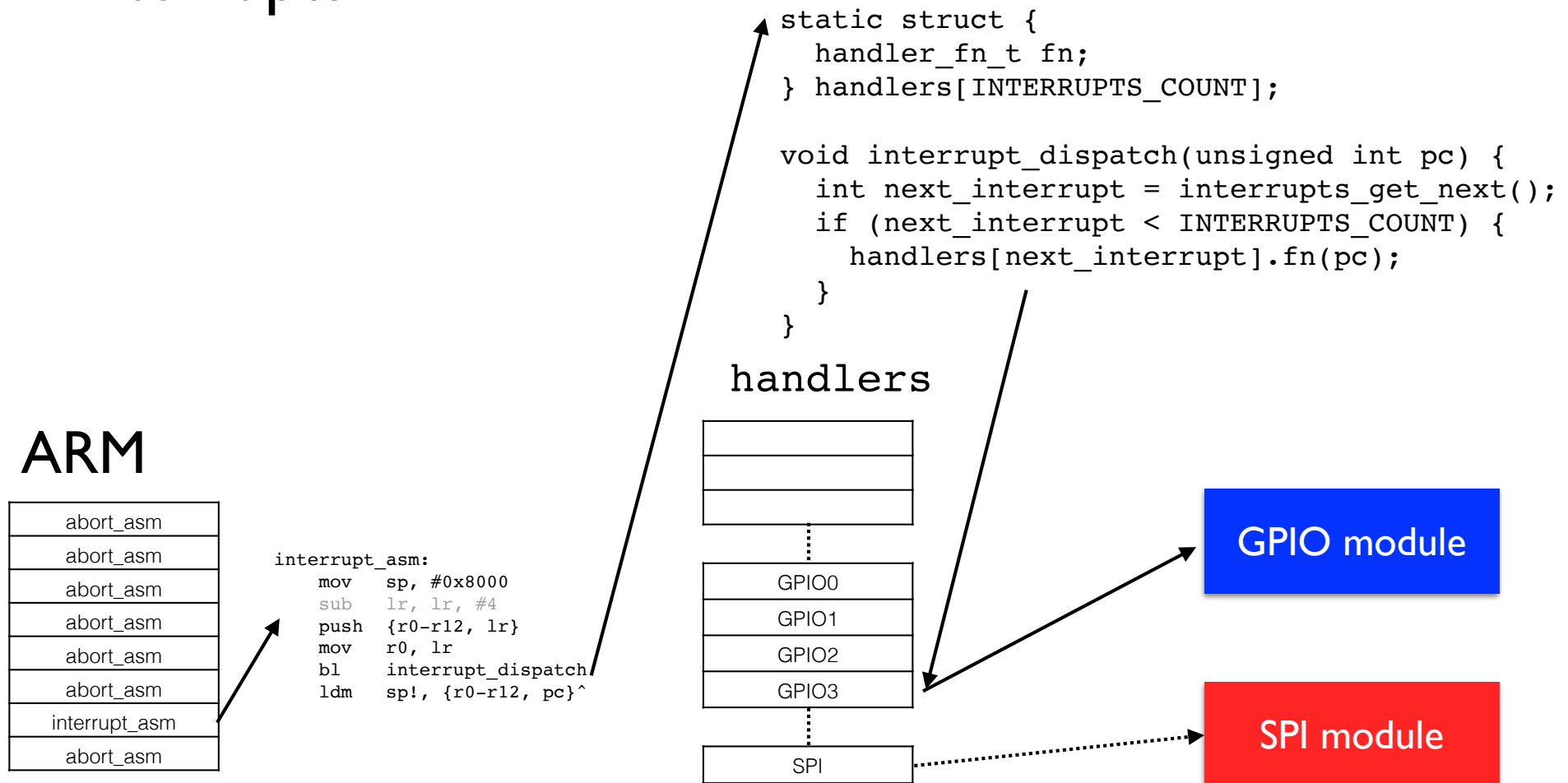
```
bool interrupts_enable_source(unsigned int source);
bool interrupts_disable_source(unsigned int source);

typedef bool (*handler_fn_t)(unsigned int);
handler_fn_t interrupts_register_handler(unsigned int source, handler_fn_t fn);

enum interrupt_source {
    INTERRUPTS_SHARED_START = 29,
    INTERRUPTS_AUX = 29,
    INTERRUPTS_I2CSPISLV = 43,
    ...
    INTERRUPTS_GPIO0 = 49,
    INTERRUPTS_GPIO1 = 50,
    INTERRUPTS_GPIO2 = 51,
    INTERRUPTS_GPIO3 = 52,
    INTERRUPTS_VC_I2C = 53,
    INTERRUPTS_VC_SPI = 54,
    INTERRUPTS_VC_I2SPCM = 55,
    INTERRUPTS_VC_UART = 57,
    ...
};
```

# Implementation

Array of function pointers, mirrors structure of 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

**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;
```

<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
```

interrupt

```
extern int a;
```

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

<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
```

code uses copy of a in r3, not a; decrement is lost

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/CS111/CS140.

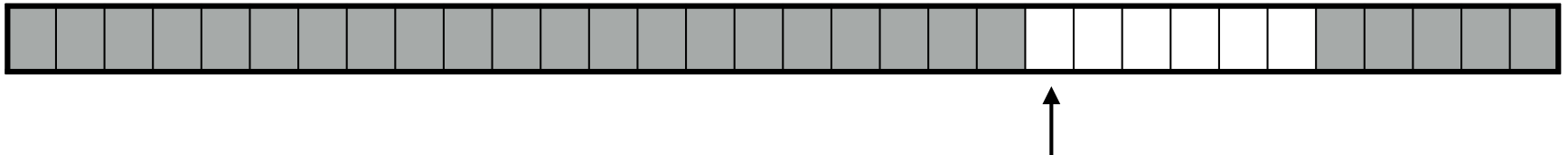
Two simple answers

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)

# 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
- Requires understanding hardware, assembly, linking, memory and C

Assignment 7: update keyboard to use interrupts