Admin

Last "normal" week

Lab 7, Assign 7 (interrupts)

Nab that full system bonus! Fixes/retest accepted up to Sun 11/24

Projects!

Start brainstorming ideas and team formation (2 people is best)

More info coming up in Friday's lecture

Interrupts (resumed)

Last time

Exceptional control flow

Low-level mechanisms to transfer to/from trap handling code

Today

Steps to configure and enable interrupts

Design of module to manage interrupt system

Details encapsulated inside module

Client interface that is safe, convenient, flexible

Coordination of activity

Exceptional and non-exceptional code, dispatch to multiple handlers

Data sharing, code designed so that it can be safely interrupted

Interrupts (so far)

Top-level interrupts system configuration

- External interrupts enabled `mstatus`, `mie` CSRs
- Trap handler address installed in `mtvec` CSR

Transfer of control to enter/exit interrupt code

- Assembly to preserve registers
- Call into C code
- Assembly to restore registers, resume interrupted code

Today:

- How to configure system so interrupts are generated
- How to process and clear interrupt
- Design of interrupts module, dispatcher

Three layers

1. Configure/enable peripheral to generate interrupt on specific event

    E.g. line goes high on certain gpio pin, countdown timer elapsed, char received on uart

2. Configure/enable interrupt source at top-level

3. Globally enable interrupts

Interrupt generated if and only if all three layers enabled

Forgetting to enable one is a common bug

HStimer events

Config timer

- `hstimer_init(hstimer_id, int usecs_interval)`

  Countdown interval (microseconds)

Enable

- `hstimer_enable(id)` starts countdown, interrupt fires when countdown reaches zero

Clear interrupt

- `hstimer_interrupt_clear(id)` resets countdown

References

- Section 3.7 p. 192 in D1-H User Manual
- Review our code in `$CS107E/src/hstimer.c`

Gpio events

Configure specific event per pin to trigger interrupt

- `gpio_interrupt_config(pin, event, debounce)`

Enable will generate interrupt on event

- `gpio_interrupt_enable(pin)`

Clear interrupt to reset

- `gpio_interrupt_clear(pin)`

References

- Section 9.7.3.6 p. 1079 in D1-H User Manual
- Review our code in `$CS107E/src/gpio_interrupt.c`

Handling event?

Top-level interrupts system configuration

- External interrupts enabled `mstatus`, `mie` CSRs
- Trap handler address installed in `mtvec` CSR

Transfer of control to enter/exit interrupt code

- Assembly to preserve registers
- Call into C code  Wait, what code is this again?
- Assembly to restore registers, resume interrupted code

Dispatch to handler

Each interrupt starts with same actions

- Execute instruction at address stored in `mtvec CSR`
  - `trap_handler` C function
- Single interrupt controller shared by entire program
- How to support different response to timer event vs. button event vs. key event?

Need handler per-event

- **Function pointers** save the day!
- Each event source has independent handler
- Interrupts module invokes handler registered for source

Goals for `interrupts` module

Convenience, safety

- Abstract away details
- Simple consistent interface
- Defend against mis-use, avoid runtime failures (debugging!)

Flexible

- Support different use cases (individual handler per interrupt source, independent enable/disable per source)

Speed

- Minimize number of cycles spent in library
  - Ideally quick and direct handoff to client function

Interrupt sources

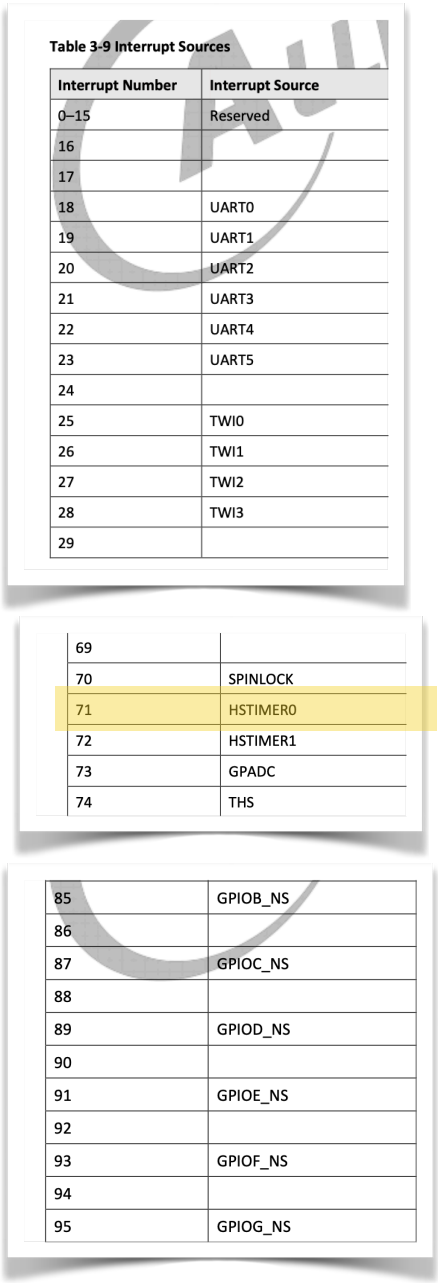


Table 3-9 Interrupt Sources

Interrupt Number	Interrupt Source
0–15	Reserved
16	
17	
18	UART0
19	UART1
20	UART2
21	UART3
22	UART4
23	UART5
24	
25	TWI0
26	TWI1
27	TWI2
28	TWI3
29	
69	
70	SPINLOCK
71	HSTIMER0
72	HSTIMER1
73	GPADC
74	THS
85	GPIOB_NS
86	
87	GPIOC_NS
88	
89	GPIOD_NS
90	
91	GPIOE_NS
92	
93	GPIOF_NS
94	
95	GPIOG_NS

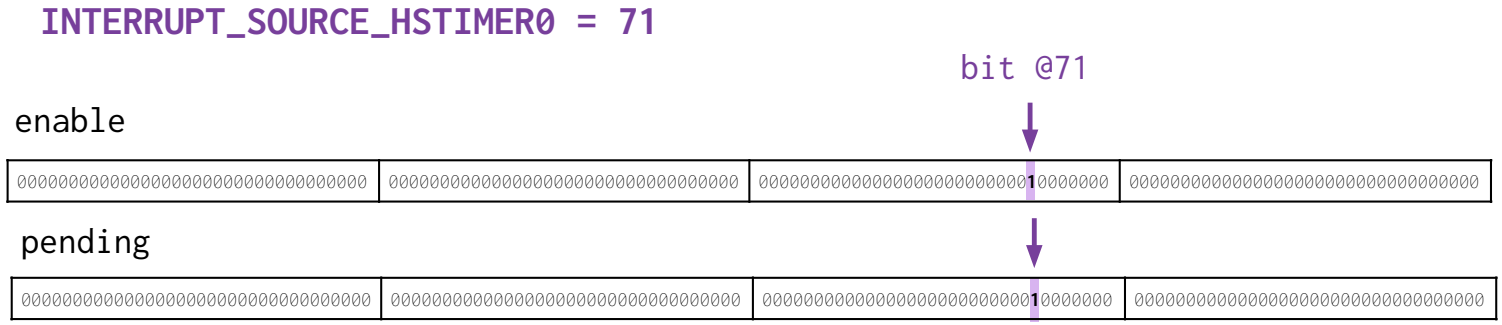
```
enum interrupt_source_t {
    INTERRUPT_SOURCE_UART0 = 18,
    INTERRUPT_SOURCE_UART1 = 19,
    INTERRUPT_SOURCE_UART2 = 20,
    INTERRUPT_SOURCE_UART3 = 21,
    INTERRUPT_SOURCE_UART4 = 22,
    INTERRUPT_SOURCE_UART5 = 23,
    INTERRUPT_SOURCE_HSTIMER0 = 71,
    INTERRUPT_SOURCE_HSTIMER1 = 72,
    INTERRUPT_SOURCE_GPIOB = 85,
    INTERRUPT_SOURCE_GPIOC = 87,
    INTERRUPT_SOURCE_GPIOD = 89,
    INTERRUPT_SOURCE_GPIOE = 91,
    INTERRUPT_SOURCE_GPIOF = 93,
    INTERRUPT_SOURCE_GPIOG = 95,
};
```

Each interrupt source assigned a number

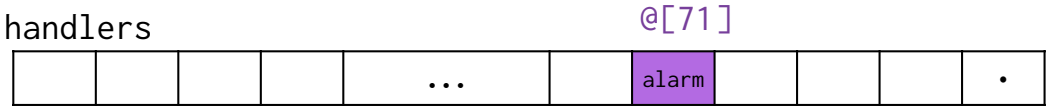
Interrupt event triggered by source number

Interrupts module dispatch

Interrupt peripheral registers, one bit per interrupt source



Array of handlers (function pointers)
mirrors structure of peripheral registers



Top-level dispatch in interrupts module

```
void trap_handler(void) {
   int idx = index_of_source();
   handlers[idx].fn(handlers[idx].data);
}
```

Calls client-supplied handler function for event

```
void alarm(void *data) {
    hstimer_interrupt_clear(HSTIMER0);
    play_sound();
}
```

Registering a handler

Client registers handler (function pointer) for interrupt source

Store function pointers in array, one per interrupt source

- Interrupt source number is index into array

Top-level trap handler dispatches to per-source handler

- Claim register has interrupt source number, use as index into 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 within a group

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

`gpio_interrupt_init` registers handler with top-level `interrupts` module

- Shared gpio_interrupt handler receives all gpio events, further dispatches to client's per-pin handler

Internal structure of `gpio_interrupt` similar to top-level `interrupts`

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

Review our code in `$CS107E/src/gpio_interrupt.c`

Client handler function

```
typedef void (*handlerfn_t)(void *);
```

One argument: client data **`void*`**

Ordinary C function (save/resume managed by top-level trap handler)

Handler operation should be lean (fast in & out!)

Handler must clear event!

- Otherwise event will continue to re-fire interrupt

Interrupt checklist for client

Client must:

- ✅ **Initialize interrupts module (and possibly gpio_interrupt module)**

Event-specific:

- ✅ **Config for desired event**
  - E.g., hstimer countdown reaches zero
- ✅ **Write handler function to process event**
  - Handler acts on event and **clears** it

- ✅ **Register handler with dispatcher**
  - `gpio_interrupt_register_handler` (if gpio event) OR `interrupts_register_handler` (all others)
- ✅ **Enable interrupt source**
  - `gpio_interrupt_enable` (if gpio event) OR `interrupts_enable_source` (all others)
- ✅ **Globally enable interrupts**
  - `interrupts_global_enable` (big switch on when everything ready)

All steps essential

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

Typical symptom is absence of action, revisit checklist to find what's missing

Sample client code

```
void timer_event(void *aux_data) {
   hstimer_interrupt_clear(HSTIMER0);
   uart_putchar('T');
}

void button_click(void *aux_data) {
   gpio_interrupt_clear(BUTTON);
   uart_putchar('B');
}

void config_timer(void) {
    hstimer_init(HSTIMER0, 1000000);
    hstimer_enable(HSTIMER0);
    interrupts_register_handler(INTERRUPT_SOURCE_HSTIMER0, timer_event, NULL);
    interrupts_enable_source(INTERRUPT_SOURCE_HSTIMER0);
}

void config_button(void) {
    gpio_interrupt_init();
    gpio_interrupt_config(BUTTON, GPIO_INTERRUPT_NEGATIVE_EDGE, true);
    gpio_interrupt_register_handler(BUTTON, button_click, NULL);
    gpio_interrupt_enable(BUTTON);
}

void main(void)  {
    uart_init();
    interrupts_init();
    config_timer();
    config_button();
    interrupts_global_enable();
    while (1) ;
}
```

code/interrupt_party

What's left?

An interrupt can fire at any time

- Interrupt handler adds a PS/2 scancode to a queue
- What if this interrupts `main` right as it is removing scancode from same queue?
- Need to maintain integrity of shared queue

Must write code so that it can be safely interrupted

Atomicity

main code

```
static int nevents;

    nevents--;
```

interrupt handler

```
static int nevents;

    nevents++;
```

Q. What is the atomic (i.e., indivisible) unit of computation?

Q. Can an update to **nevents** *be lost when switching between these two code paths?*

A problem

main code

```
static int nevents;

    nevents--;
```

```
li      a4,&nevents
lw      a5,0(a4)
addiw   a5,a5,-1
sw      a5,0(a4)
```

interrupt handler

```
static int nevents;

    nevents++;
```

```
li      a4,&nevents
lw      a5,0(a4)
addiw   a5,a5,1
sw      a5,0(a4)
```

How can an increment be lost if interrupt between these two instructions?

A problem

main code

```
static int nevents;

    nevents--;
```

```
li      a4,&nevents
lw      a5,0(a4)
addiw   a5,a5,-1
sw      a5,0(a4)
```

interrupt handler

```
static int nevents;

    nevents++;
```

```
li      a4,&nevents
lw      a5,0(a4)
addiw   a5,a5,1
sw      a5,0(a4)
```

Resume instruction uses value previously loaded into **a5**. What happened to increment done by interrupt handler?

Disabling interrupts

main code

```
interrupts_global_disable();
nevents--;
interrupts_global_enable();
```

interrupt handler

```
nevents++;
```

Q. Does increment need bracketing also?

Preemption and safety

Very hard, lots of bugs.

You'll learn more in CS111/CS140.

Two simple answers

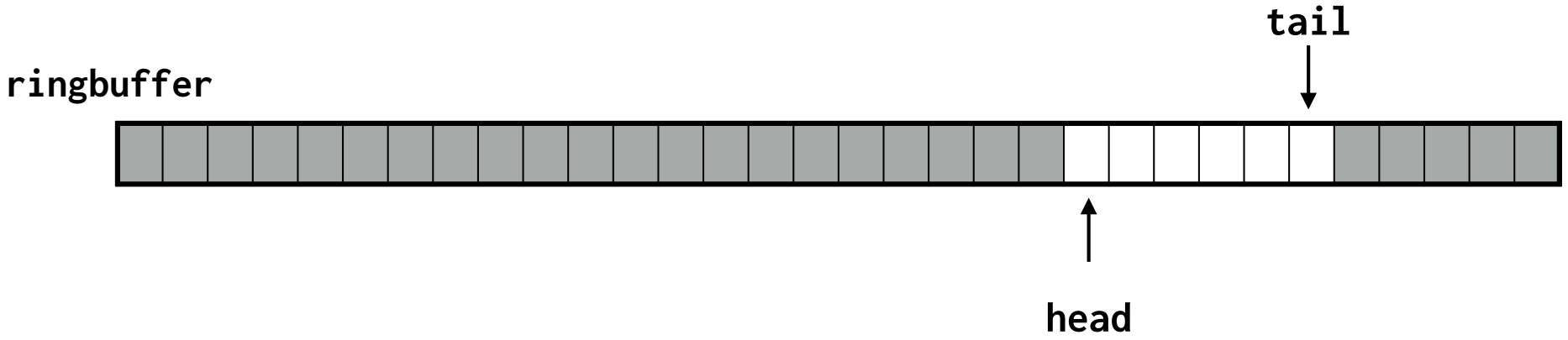
1. Use simple, safe data structures
   - single writer (not always possible)
2. Otherwise, temporarily disable interrupts
   - works if used correctly, easy to get wrong

Safe ringbuffer

A simple approach to avoid interference is for different code paths to not write to same variables

Queue implemented as ring buffer:

- Enqueue (interrupt) writes element to tail, advances tail
- Dequeue (main) reads element from head, advances head



Ringbuffer code

```
bool rb_enqueue(rb_t *rb, int elem)
{
     if (rb_full(rb)) return false;

     rb->entries[rb->tail] = elem;
     rb->tail = (rb->tail + 1) % LENGTH; // only changes tail
     return true;
}
```

```
 bool rb_dequeue(rb_t *rb, int *p_elem)
 {
     if (rb_empty(rb)) return false;

     *p_elem = rb->entries[rb->head];
     rb->head = (rb->head + 1) % LENGTH; // only changes head
     return true;
 }
```

Review our code in `$CS107E/src/ringbuffer.c`

Interrupts in summary

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

- Needed for responsiveness, e.g., not miss PS/2 scancode during drawing
- Most activity is interrupt-driven: responding to keystrokes, network packets, disk reads, timers, etc.

Config and debug interrupts is challenging!

- Deals with many of the hardest issues in systems

Assign7: update ps2 driver to read by interrupt