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 (so far)

Vector table installed in correct location

- Copy vector table to 0x0
- Embed addresses with table so jumps are absolute

Correct transfer of control to/from interrupt mode

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

Now:

- How to configure system so interrupts are generated

Three Layers

1. Enable specific interrupt event

   Which action to detect: e,g, line high on certain gpio pin, countdown timer elapsed, char received on part

2. Enable interrupt source

   e.g. gpio, armtimer, uart

3. Globally enable interrupts

Interrupt generated if and only if all three layers enabled

Forgetting to enable one is a common bug

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`

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`

Handling event?

Vector table installed in correct location

- Copy vector table to 0x0
- Embed addresses with table so jumps are absolute

Correct transfer of control to/from interrupt mode

- Assembly to set up stack, save registers
- Call into C code  Wait, what code is this again?
- Assembly to restore registers, resume interrupted code

Interrupt dispatch

All interrupts start with same actions

- Execute instruction at `vectors[IRQ]`, jump to `interrupt_asm` which calls C function `interrupt_dispatch`
- Single interrupts peripheral 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 determines which source had event and invokes handler registered for that 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
  - Handler may be just a few instructions and runs very often

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		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	

Huh??

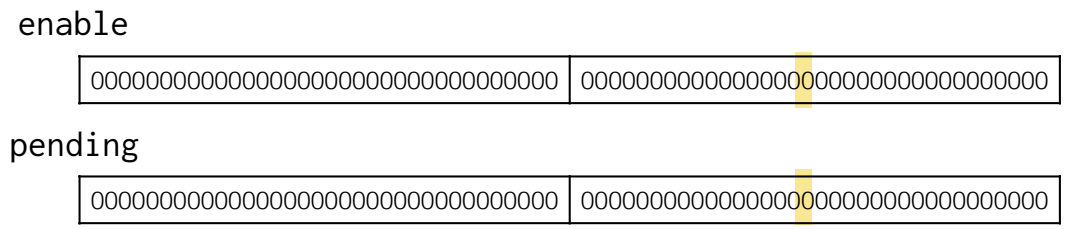
⚠ *Documentation is sparse ...*

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

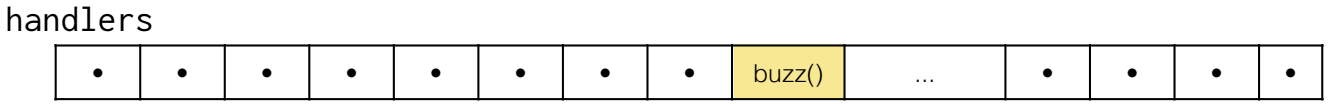
Implementation

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,
   ...
```



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




```
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:

- ✅ **Initialize interrupts (and possibly gpio_interrupts)**

Event-specific:

- ✅ **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

What's left?

An interrupt can fire at any time

- Interrupt handler adds a PS/2 scancode to a queue
- Could do so right as `main` is in middle of removing a scancode from the 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--;


8074: ldr  r3, [pc, #12]
8078: ldr  r2, [r3]
807c: sub  r2, r2, #1
8080: str  r2, [r3]

8088: .word  0x0000a678
```

interrupt handler

```
static int nevents;

    nevents++;


808c: ldr  r3, [pc, #12]
8090: ldr  r2, [r3]
8094: add  r2, r2, #1
8098: str  r2, [r3]

80a0: .word  0x0000a678
```

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

A problem

main code

```
static int nevents;

    nevents--;


8074: ldr  r3, [pc, #12]
8078: ldr  r2, [r3]
807c: sub  r2, r2, #1
8080: str  r2, [r3]

8088: .word  0x0000a678
```

interrupt handler

```
static int nevents;

    nevents++;


808c: ldr  r3, [pc, #12]
8090: ldr  r2, [r3]
8094: add  r2, r2, #1
8098: str  r2, [r3]

80a0: .word  0x0000a678
```

Resume instruction uses value previously loaded into r2. 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 CS110/CS140.

Two simple answers

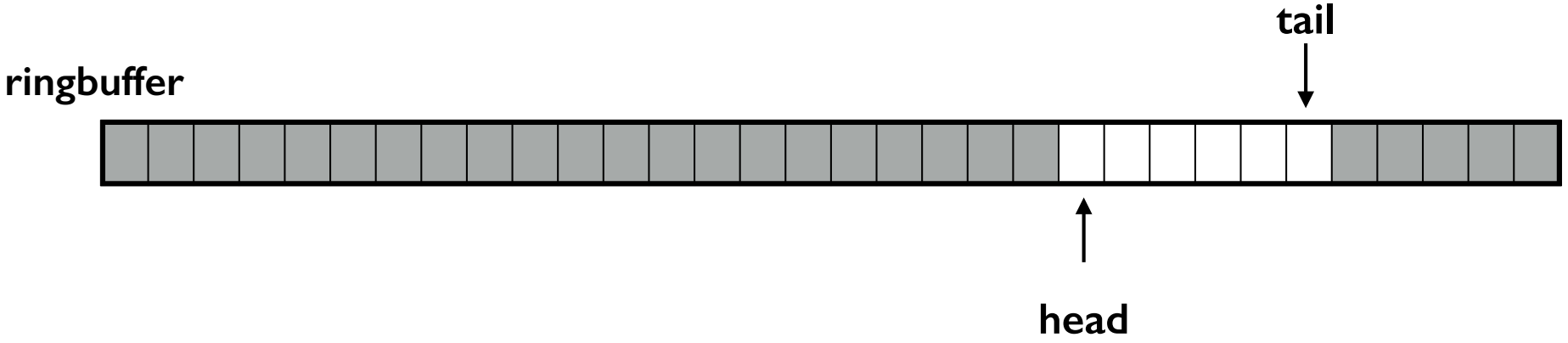
1. Use simple, safe data structures
   - single writer (not always possible)
2. Otherwise, temporarily disable interrupts
   - works if used correctly, easy to get wrong

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 *elem)
 {
     if (rb_empty(rb)) return false;

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

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

Summary

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

- Needed for responsiveness, e.g., not miss PS/2 scancodes 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 ps2 driver to use interrupts