# Admin

Momentum through end-game
Reach out if you need a pep talk

# Interrupts (resumed)

## Last time

Exceptional control flow

(low-level mechanisms)

## Today

Using interrupts as client

Configure, enable, register handler

Coordination of activity

Exceptional and non-exceptional code, dispatch to multiple handlers

Data sharing, writing code that can be safely interrupted

# Looking ahead

This week: Lab 7, Assign 7

*Finishing touches*

*Nab that complete system bonus!*

Brainstorm project ideas and form teams now

Labs 8 & 9 are group work on final project

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

Configure system to generate interrupts

- Three levels to enable:
  specific event/peripheral, interrupt source, global enable

# Armtimer events

Initialize timer

- **`armtimer_init(unsigned int nticks)`**
- period for alarm set in ticks (microseconds)

Enable

- **`armtimer_enable()`** starts timer running
- **`armtimer_enable_interrupts()`** will generate interrupt at end of period

Status

- **`armtimer_check_and_clear_interrupt()`**
- Clear starts timer again

References

- P. 196 in BCM2835 ARM Peripherals doc
- Review our code in **`$CS107E/src/armtimer.c`**

# Gpio events

Enable specific event per gpio pin

- **`gpio_enable_event_detection`**
- Different options: falling edge, rising edge, high level, etc.

Event detect register has bit per gpio pin

- Bit is set when event occurs, follow up to check/clear bit
- **`gpio_check_and_clear_event`**
- Must clear bit, if not, interrupt will keep re-triggering

References

- P. 96-99 in BCM2835 ARM Peripherals doc
- Review our code in **`$CS107E/src/gpio_extra.c`**

# We're done .... ?

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

Configure system to generate interrupts

- Three levels to enable:
  specific event/peripheral, interrupt source, global enable

# Interrupt dispatch

Every interrupt starts with same actions

- Executes instruction at `vectors[IRQ]` which jumps to `interrupt_asm` which calls C function `interrupt_dispatch`
- But need different handling for timer event vs. button event vs. key event!
- One interrupts module, shared by entire program, ...

Need handler per-event

- Function pointers save the day again!
- Each event source can have its own handler
- Interrupts module identifies which event source and invokes handler for that source

# Goals for interrupts API

Avoid runtime failures (i.e., debugging)

- Defend against mis-use
- Simplify steps where possible

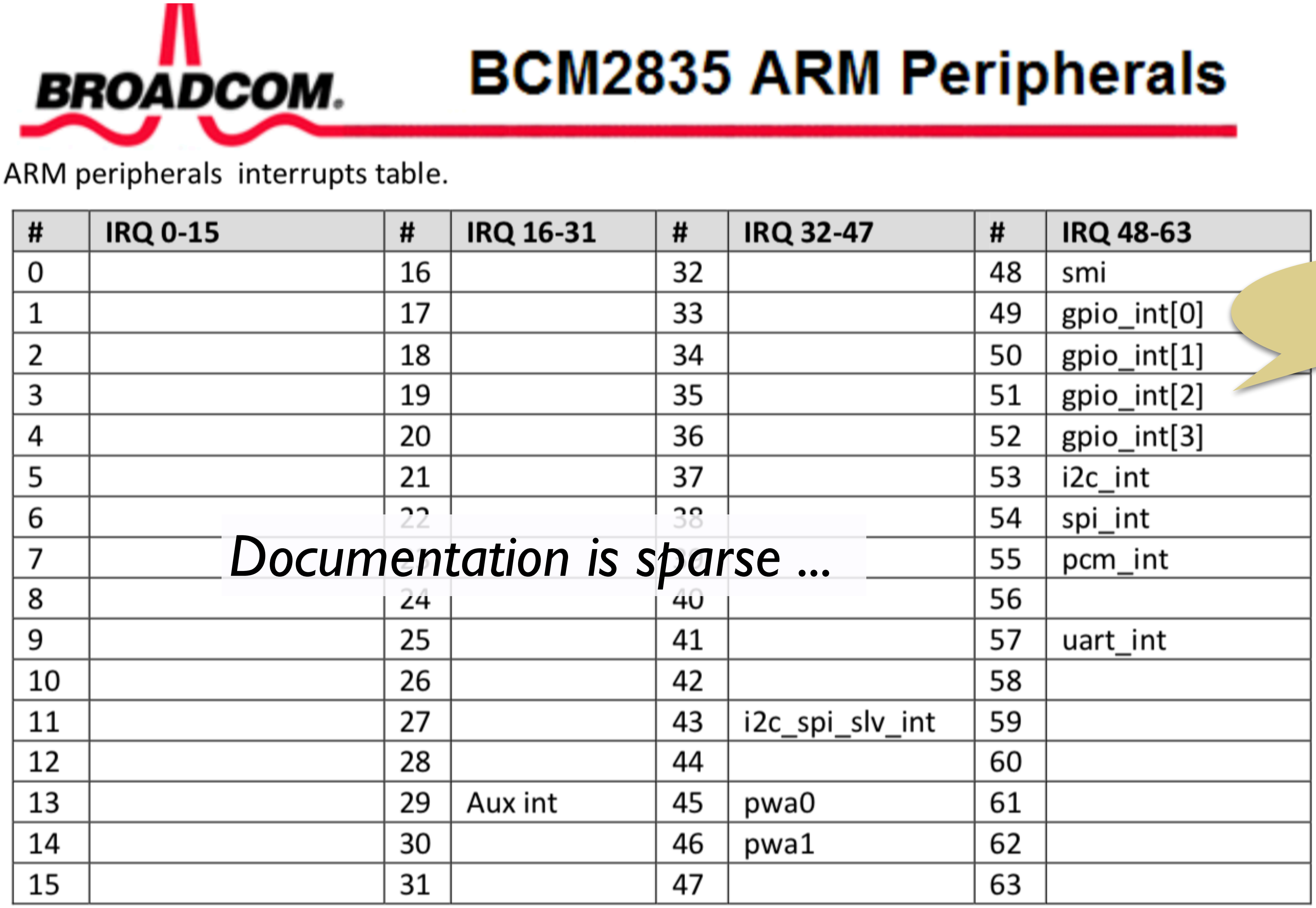
Flexible

- Allow add new modules that handle interrupts

Speed

- Minimize number of cycles spent in dispatch
  - Handler is typically brief task, may need to run very often

# Interrupt sources



BROADCOM.

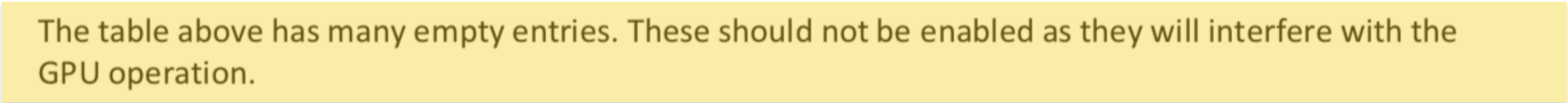
BCM2835 ARM Peripherals

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 | |

*Documentation is sparse …*

Huh??



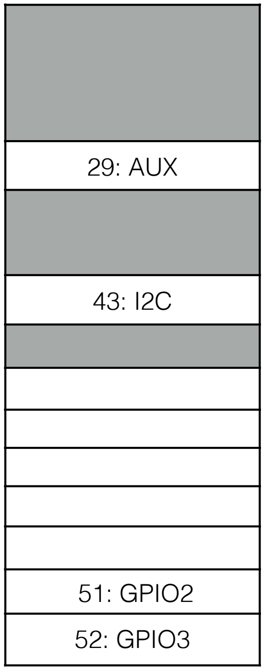
The table above has many empty entries. These should not be enabled as they will interfere with the GPU operation.

# Implementation

Array of function pointers, mirrors structure of interrupts

```
static struct {
    handler_fn_t fn;
    void *aux_data;
} handlers[INTERRUPTS_COUNT];
```

```
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,
   ...
```



# 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 which source it came from
  - Scan interrupt register to find set bit, this is source that had event
- Registered handler available in **`array[source_index]`**
- The aux data pointer can be used to pass information into the handler function
  - If not needed, aux data can be **`NULL`**
  - Data is **`void *`** to allow flexibility

From $CS107E/src/interrupts.c

```
void interrupts_register_handler(unsigned int source,
                          handler_fn_t fn, void *aux_data) {
{
   assert(interrupts_initialized);
   assert(is_valid(source));

   if (fn) {
        handlers[source].fn = fn;
        handlers[source].aux_data = aux_data;
        interrupts_enable_source(source);
    } else {
        interrupts_disable_source(source);
        handlers[source].fn = NULL;
        handlers[source].aux_data = NULL;
    }
}
```

From $CS107E/src/interrupts.c

```
// determine which source triggered interrupt
// then access index in table to get registered handler
// invoke handler, pass pc and aux_data as arguments


void interrupt_dispatch(unsigned int pc) {
    int next = get_next_source();
    if (next < INTERRUPTS_COUNT && handlers[next].fn) {
        handlers[next].fn(pc, handlers[next].aux_data);
    }
}
```

# GPIO interrupts

- One interrupt source shared by all GPIO events for all pins
- Need *another* level of dispatch to support per-pin handler
- **`gpio_interrupts_init`** registers a handler with top-level **`interrupts`** module
  - GPIO interrupt receives all gpio events
  - This handler will in turn dispatch event to per-pin handler
- Internal structure of **`gpio_interrupts`** similar to top-level **`interrupts`**
  - Array of handlers, one per pin
  - Scan event detect register to find set bit -- this identifies which gpio pin had event
- Call registered handler for that pin
- Review our code in **`$CS107E/src/gpio_interrupts.c`**

# Interrupt checklist

Client must:

✅ **Initialize interrupts, gpio_interrupts**

Event-specific {

✅ **Enable detection of desired kind of 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)

✅ **Globally enable interrupts**

- Throw the big switch to turn it all on when ready

***All steps essential***

*Fiddly code, easy to forget or mix up steps*

*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);

    gpio_interrupts_init();
    gpio_enable_event_detection(BUTTON, GPIO_DETECT_FALLING_EDGE);
    gpio_interrupts_register_handler(click, BUTTON, NULL);

    interrupts_global_enable();
    ...
```

code/interrupt_party

# What's left?

An interrupt can fire at any time

- Interrupt handler adds a PS/2 scan code to a queue
- Could do so right as `main` is in middle of removing a scan code 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 occurs here?

# 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
```

Instruction uses value copied into r2; increment of global by interrupt code is lost

Will volatile solve this?

# 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

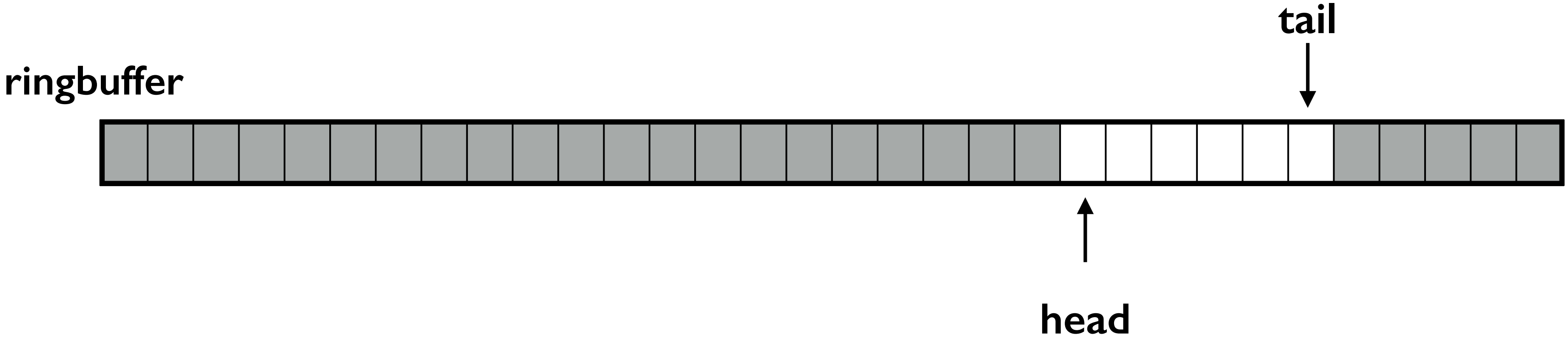
1. Use simple, safe data structures
   - write once, but not always possible
2. Otherwise, temporarily disable interrupts
   - always works, but easy to forget

# 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 writes 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 writes head
     return true;
 }
```

# Summary

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

- Needed for responsiveness, e.g., 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

Assignment 7: update ps2 driver to use interrupts