Admin

Your system nearing completion -- exciting!

Interrupts

Today

- Exceptional control flow
  - Suspend, jump to different code, then resume
  - How to do this safely and correctly
- Focus on low-level mechanisms today

Monday

- Using interrupts as client
- Coordination of activity
  - (exception & non-exception, multiple handlers, shared data)

Blocking I/O

```
while (1) {
  char ch = keyboard_read_next();
  update_screen();
}
```

How long does it take to send a scan code?

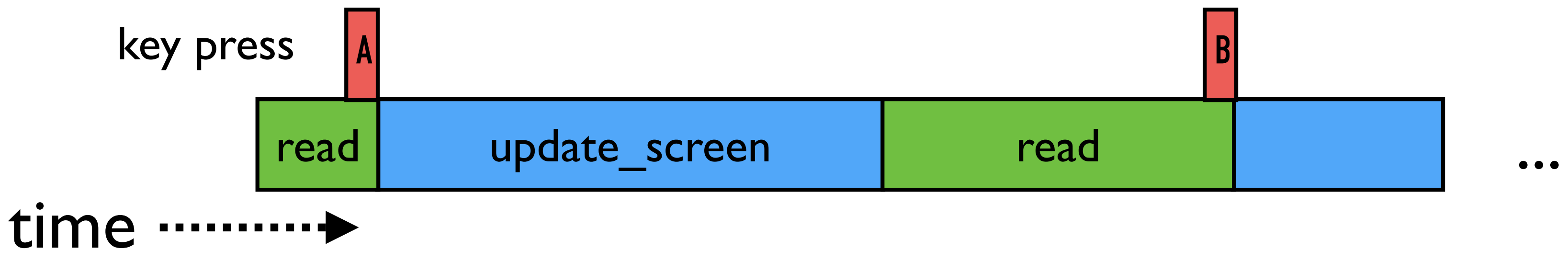
  11 bits, clock rate 15kHz

How long does it take to update the screen?

What could go wrong?

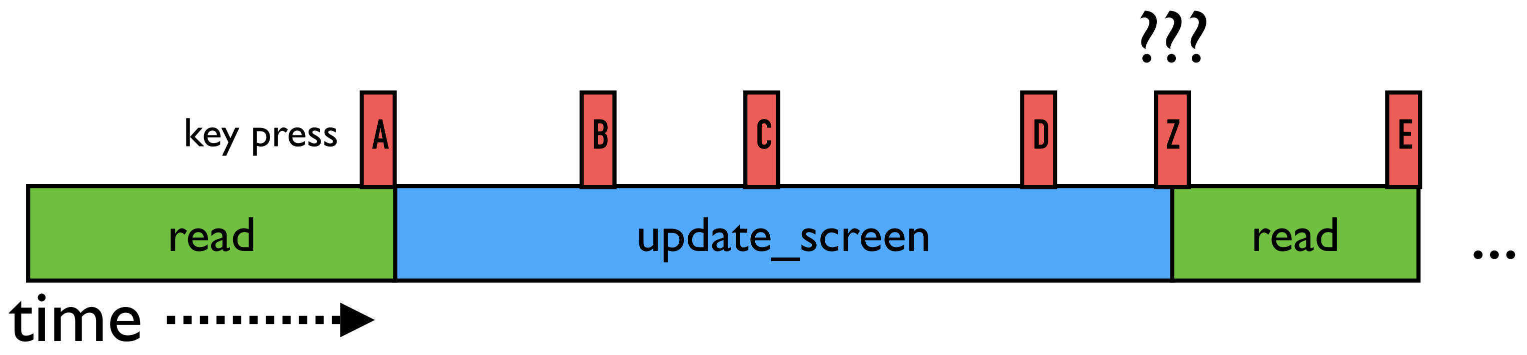
Blocking I/O

```
while (1) {
  char ch = keyboard_read_next();
  update_screen();
}
```



Blocking I/O

```
while (1) {
   char ch = keyboard_read_next();
   update_screen();
}
```



The Problem

Ongoing, long-running tasks (graphics, simulations, applications) keep CPU occupied, but …

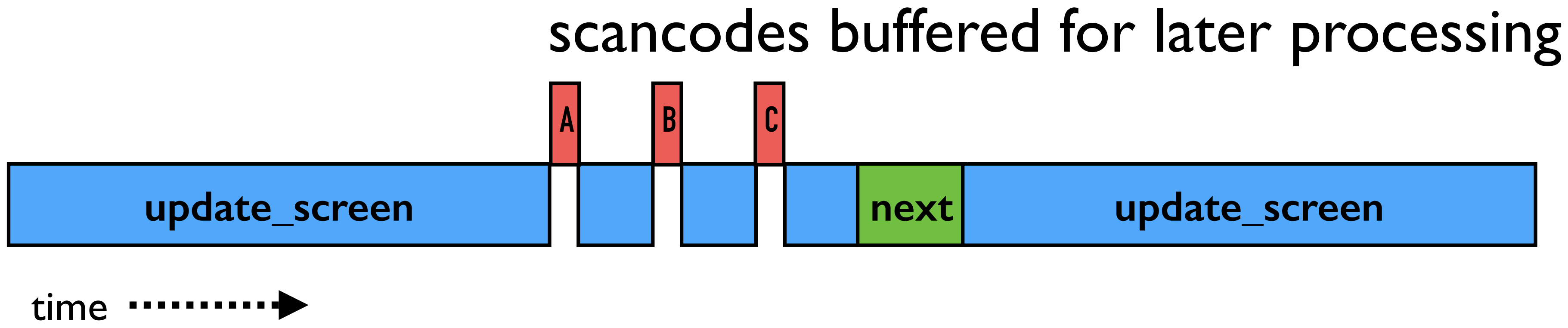
When an external event arises, need to respond quickly.

Consider: Why does your phone have a ringer? What would you do to receive a call if it didn’t?

Concurrency

```
when a scancode arrives {
  add scancode to queue;
}

while (1) {
   dequeue next
   update_screen();
}
```



Interrupts to the rescue!

Processor pauses current execution, switch to handle interrupt, then resume execution

- External events (peripherals, timer)
- Internal events (bad memory access, software trigger)

Critical for responsive systems, hosted OS

Interrupts are essential and powerful, but getting them right requires using everything you've learned:

- Architecture, assembly, linking, memory, C, peripherals, …

```
code/button-blocking
code/button-interrupt
```

Interrupted control flow

```
static volatile int gCount;
```

```
void update_screen(void)
{
  console_clear();
  for (int i = 0; i < N; i
     console_printf("%d",
}
```

```
void button_pressed(unsigned int pc, void *p)
{
  if (gpio_check_and_clear_event(BUTTON)) {
     gCount++;
  }
}
```

```
23 23 23 23 23
23 23 24 24 24
24 24
```

Suspend current activity, execute other code, then resume, ... this will be tricky!

Interrupt mechanics

Somewhat analogous to function call

- Suspend currently executing code, save state
- Jump to handler code, process interrupt
- When finished, restore state and resume

Must adhere to conventions to avoid stepping on each other

- Consider: processor state, register use, memory
- Hardware support helps out

  (separate modes, banked registers)

Consider at assembly level

```
update_screen:
  push    {r4, lr}
  bl   8298 <console_clear>
  b    .L2
.L1
  ldr r3, [pc, #32]
  ldr r1, [r3]
  ldr r0, [pc, #28]
  bl   803c <console_printf>
  add r4, r4, #1, 0
.L2
  cmp r4, #63, 0
  ble .L1
  pop {r4, lr}
  bx   lr
```

Interrupt!

```
button_pressed:
  push    {r4, lr}
  mov r0, #21, 0
  bl   9890 <gpio_check_and_clear_event>
  cmp r0, #0, 0
  ldrne    r2, [pc, #16]
  ldrne    r3, [r2]
  addne    r3, r3, #1, 0
  strne    r3, [r2]
  pop {r4, lr}
  bx   lr
```

Need to know what instruction to return to after interrupt.

Where can we store that information?

Hardware support for interrupts

Processor executing in a particular "mode"

- Supervisor, interrupt, user, abort
- Reset starts in **supervisor** mode (normal mode for our code)
- Hardware monitors interrupt sources. When event occurs, hardware switches to interrupt mode

CPSR register tracks current mode, processor state

- Special instructions copy CPSR value to/from regular register

Banked registers

- Each mode has its own unique sp and lr

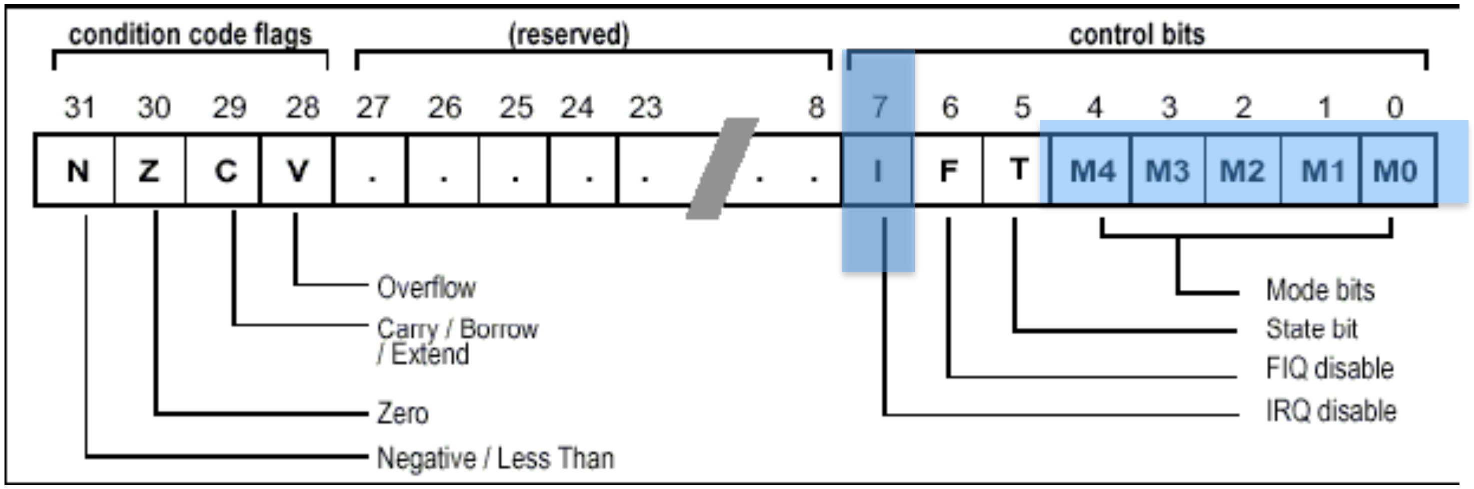
Interrupt vector

- fixed location in memory has instruction to execute on interrupt

ARM processor modes

User	unprivileged
IRQ	interrupt
FIQ	fast interrupt
Supervisor	privileged, entered on reset
Abort	memory access violation
Undefined	undefined instruction
System	privileged mode that shares user regs

CPSR



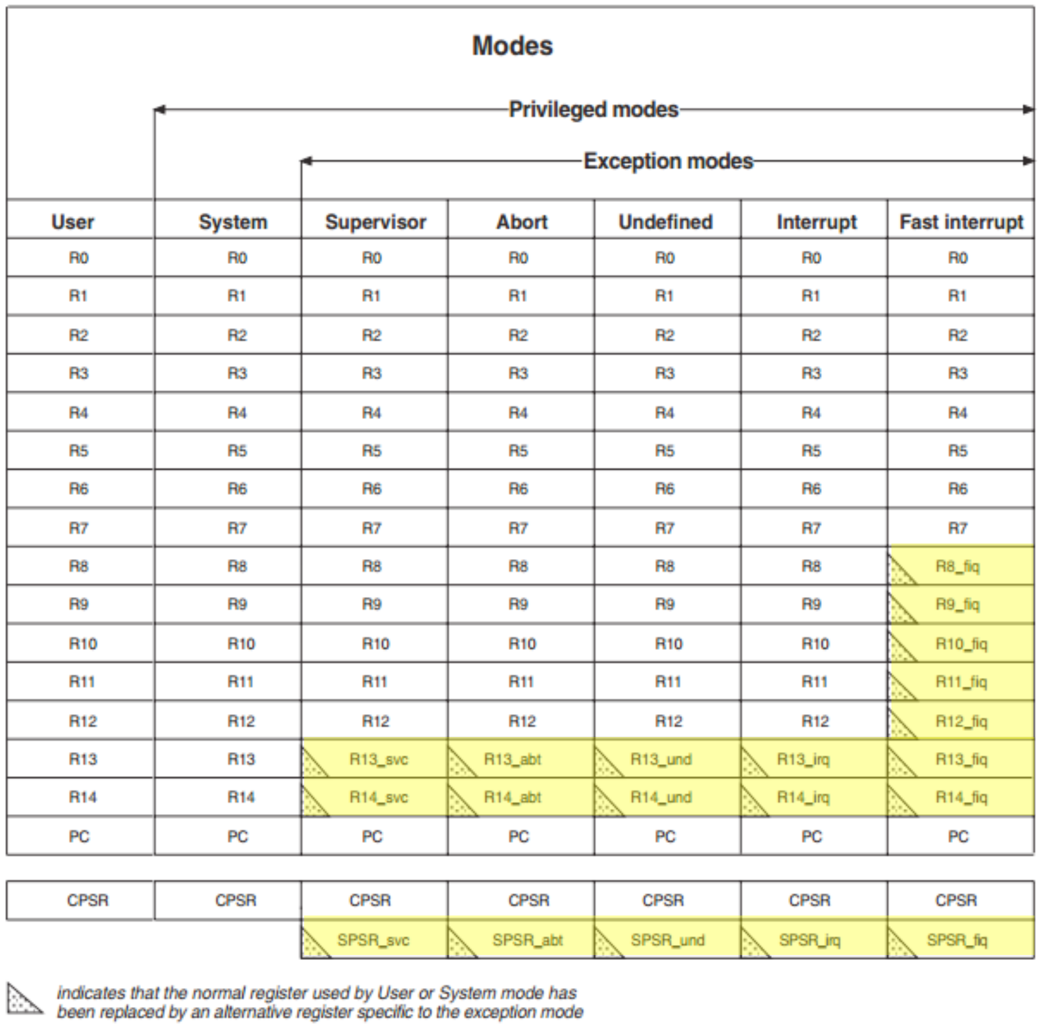
M[4:0]	Mode
b10000	User
b10001	FIQ
b10010	IRQ
b10011	Supervisor
b10111	Abort
b11011	Undefined
b11111	System

```
interrupts_global_enable:
    mrs r0, cpsr          @ copy cpsr to regular register
    bic r0, r0, #0x80     @ clear I=0 enables IRQ interrupts
    msr cpsr_c, r0        @ copy back, control bits only
    bx lr

interrupts_global_disable:
    mrs r0, cpsr          @ copy cpsr to regular register
    orr r0, r0, #0x80     @ set I=1 disables IRQ interrupts
    msr cpsr_c, r0        @ copy back, control bits only
    bx lr
```

Per-mode banked registers

Register	SVC	IRQ
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

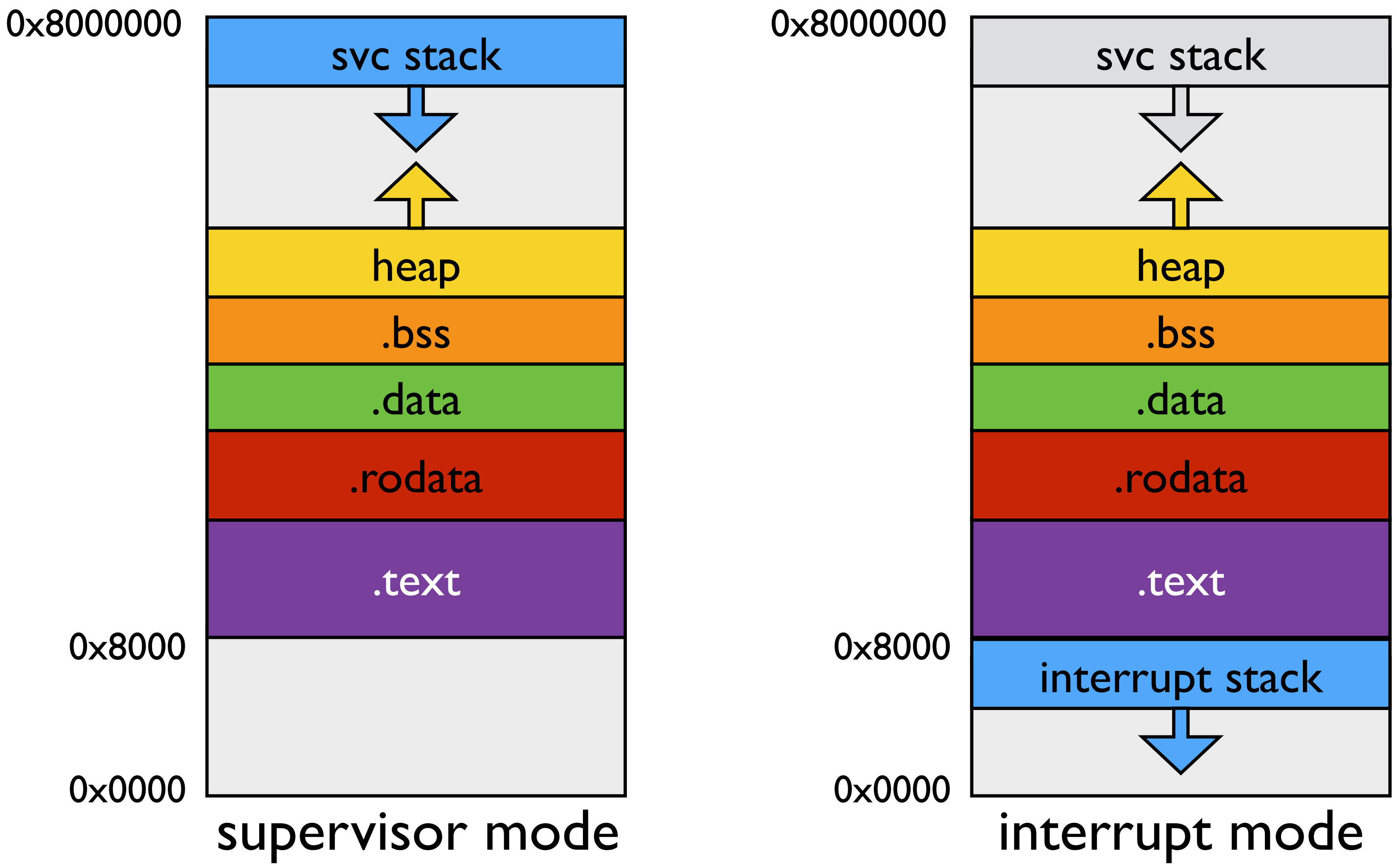
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	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

User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
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

Two stacks


0x8000000
svc stack
heap
.bss
.data
.rodata
.text
0x8000
0x0000
supervisor mode
0x8000000
svc stack
heap
.bss
.data
.rodata
.text
0x8000
interrupt stack
0x0000
interrupt mode

Interrupt, hardware-side

External event triggers interrupt. Processor response:

- Complete current instruction
- Change mode, store prev PC into LR of new mode to use as resume address, save CPSR of old mode into SPSR

  Tricky! Needs to happen "simultaneously"…
- Further interrupts disabled until exit interrupt mode
- Set PC = 0x18 (index 6 in vector table, IRQ)
- Software takes over

Interrupt, software-side

```
interrupt_asm:
    mov   sp, #0x8000           @ init stack
    sub   lr, lr, #4            @ compute resume addr
    push  {r0-r12, lr}          @ save all registers
    mov   r0, lr                @ pass resume addr as arg
    bl    interrupt_dispatch    @ call C function
    ldm   sp!, {r0-r12, pc}^    @ resume svc, ^ changes
                                  mode & restores cpsr
```

Could we do steps above in C?

```
void interrupt_dispatch(unsigned int pc)
{
  // process interrupt in C code
}
```

Does this code have additional restrictions on what it must/cannot do?

ARM Interrupts

Address	Exception	Mode
0x00000000	**Reset**	Supervisor
0x00000004	**Undefined instruction**	Undefined
0x00000008	**Software Interrupt (SWI)**	Supervisor
0x0000000C	**Prefetch Abort**	Abort
0x00000010	**Data Abort**	Abort
0x00000018	**IRQ (Interrupt)**	IRQ
0x0000001C	**FIQ (Fast Interrupt)**	IRQ

Installing Interrupt Code

The CPU will jump to specific addresses when an interrupt occurs.

We need to copy the code we want to run to these addresses.

Writing code that can be safely copied there requires a great deal of care, understanding assembly and linking.

Desired Assembly

Generate this assembly code and copy it to exception table location (0x00000000).

```
00000000:
    0: b abort_asm
    4: b abort_asm
    8: b abort_asm
    c: b abort_asm
   10: b abort_asm
   14: b abort_asm
   18: b interrupt_asm
   1c: b abort_asm
```

Install vectors

```
unsigned int *dst = _RPI_INTERRUPT_VECTOR_BASE;
unsigned int *src = &_vectors;
unsigned int n = &_vectors_end - &_vectors;

for (int i = 0; i < n; i++) {
    dst[i] = src[i];
}
```

Table has just one instruction per interrupt type
Use that instruction to "vector" to code elsewhere

Branches are relative!

`_vectors:`
~~`b abort_asm`~~
~~`b abort_asm`~~
~~`b abort_asm`~~
~~`b abort_asm`~~
~~`b abort_asm`~~
~~`b abort_asm`~~
~~`b interrupt_asm`~~
~~`b abort_asm`~~

Branch target is pc-relative. If we move the code, they won't jump to the right address.

```
0000807c <_vectors>:
    807c:       ea000006        b       809c <abort_asm>
    8080:       ea000005        b       809c <abort_asm>
    8084:       ea000004        b       809c <abort_asm>
    8088:       ea000003        b       809c <abort_asm>
    808c:       ea000002        b       809c <abort_asm>
    8090:       ea000001        b       809c <abort_asm>
    8094:       ea000001        b       80a0 <interrupt_asm>
    8098:       eaffffff        b       809c <abort_asm>

0000809c <abort_asm>:
    809c:       eafffffe        b       809c <abort_asm>

000080a0 <interrupt_asm>:
    80a0:       e3a0d902        mov     sp, #32768      ; 0x8000
```

Explicitly Embedded Absolute Addresses

```
_vectors:
    ldr pc, abort_addr
    ldr pc, abort_addr
    ldr pc, abort_addr
    ldr pc, abort_addr
    ldr pc, abort_addr
    ldr pc, abort_addr
    ldr pc, interrupt_addr
    ldr pc, abort_addr

    abort_addr:         .word abort_asm
    interrupt_addr:     .word interrupt_asm

_vectors_end:
```

"position-independent code"

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

Interrupts (so far)

Hardware support
    Processor modes, banked registers, exceptional control flow

Software init
    Install handler, position-independent code

Exception received, handler must:
    Init stack, save registers, process event
    Restore and resume

Next Lecture

Which interrupts are supported, how to configure
  Gpio events, timer, uart, ...

Dispatch for multiple event sources/handlers

What steps needed to init/enable interrupts

Writing safe interrupt handlers
    How to share state that can be modified at any time?