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 and non-exception, multiple handlers)

Synchronous I/O

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

How long does it take to send a scan code?

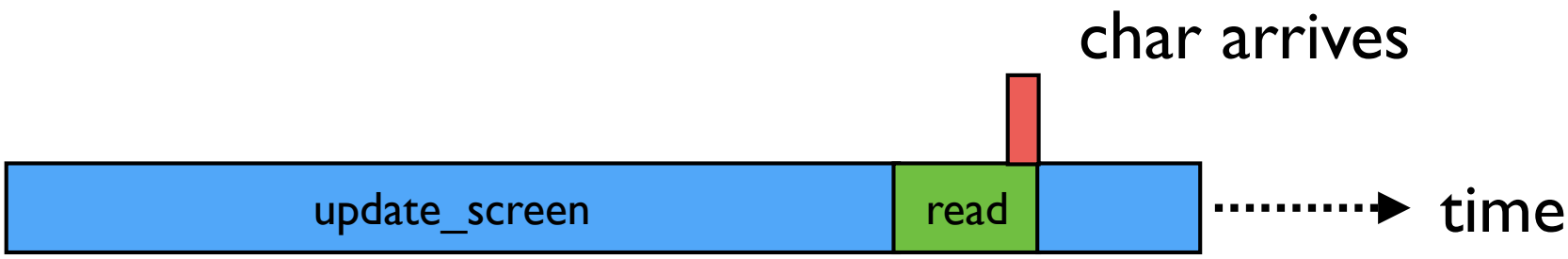
- 11kHz, 11 bits/scan code

How long does it take to update the screen?

What could go wrong?

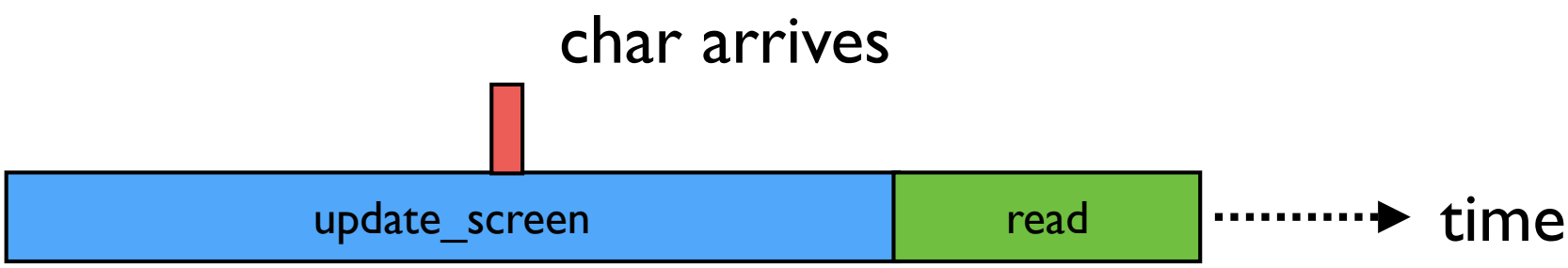
Synchronous I/O

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



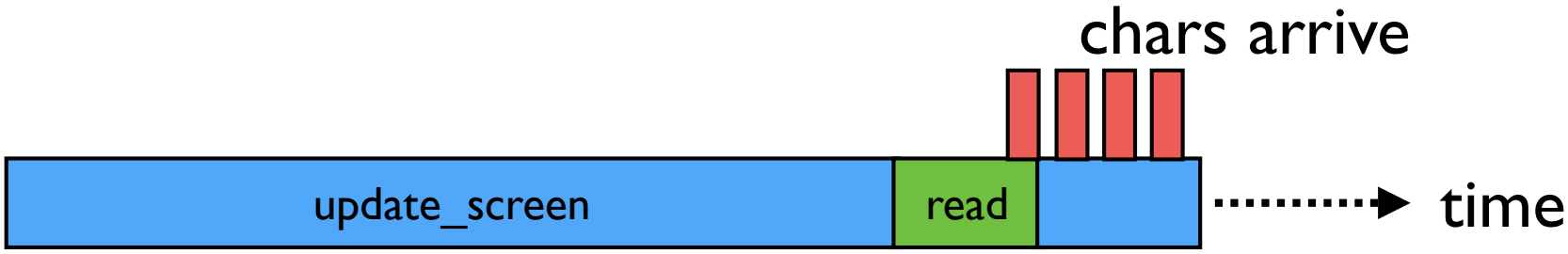
Synchronous I/O

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



Synchronous I/O

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



The Problem

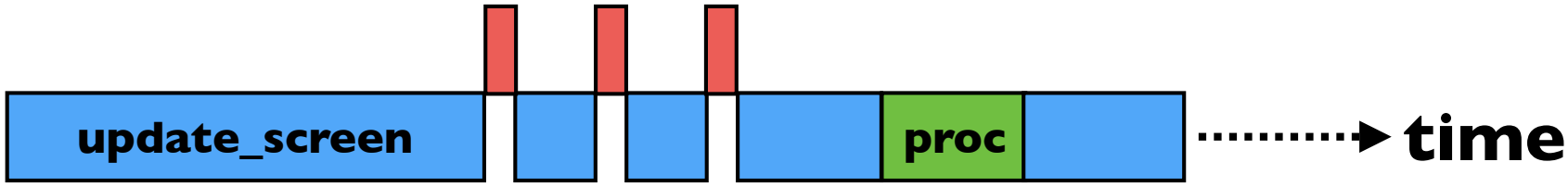
Ongoing and long-running computations (graphics, simulations, applications, …) are keeping CPU occupied, but…

When an external event arises, need to respond immediately/quickly.

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

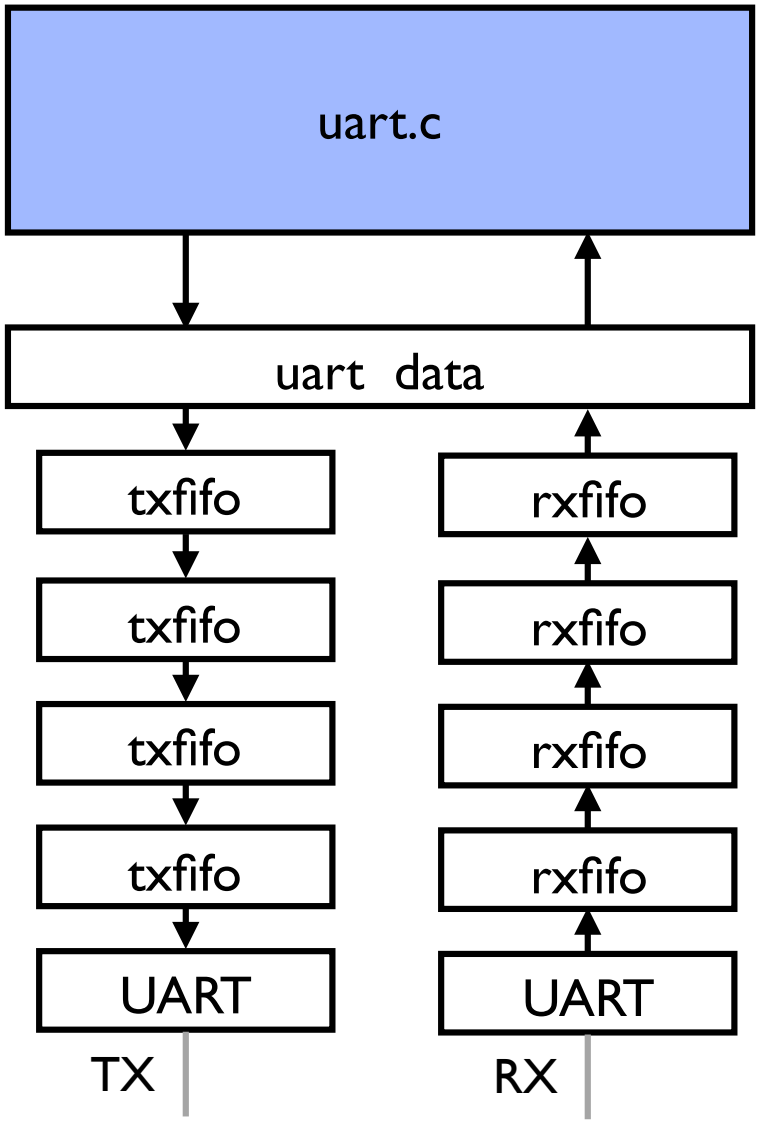
Asynchronous processing

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

while (1) {
   while (queue is empty) {}
   update_screen();
}
```



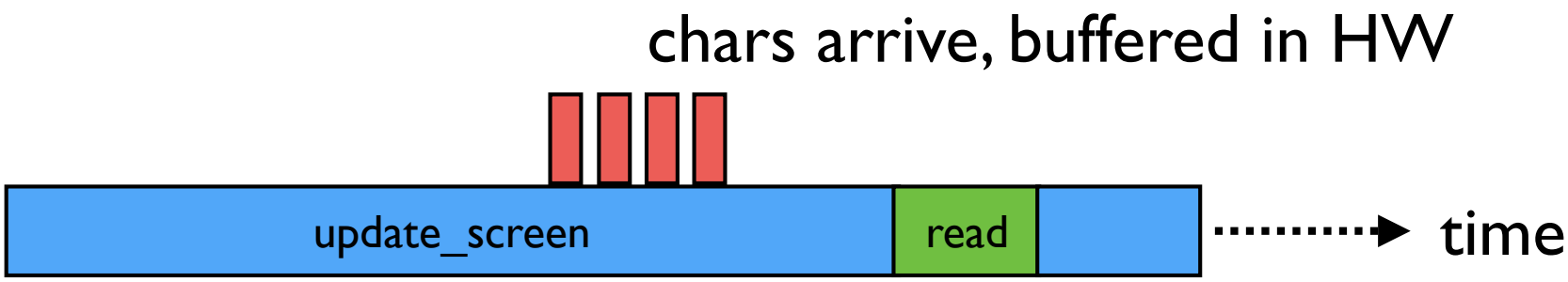
Hardware Can Help

```
int uart_getc(void) {
    while (!(uart->lsr & MINI_UART_LSR_RX_READY)) ;
    return uart->data & 0xFF;
}

void uart_putc(unsigned c) {
    if (c == '\n') {
        uart_putc('\r');
    }
    while (!(uart->lsr & MINI_UART_LSR_TX_EMPTY)) ;
    uart->data = c;
}
```



Asynchronous I/O (with HW help)

```
while (1) {
  while (queue is empty) {}
  update_screen();
}
```



Interrupts to the rescue!

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

- 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

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

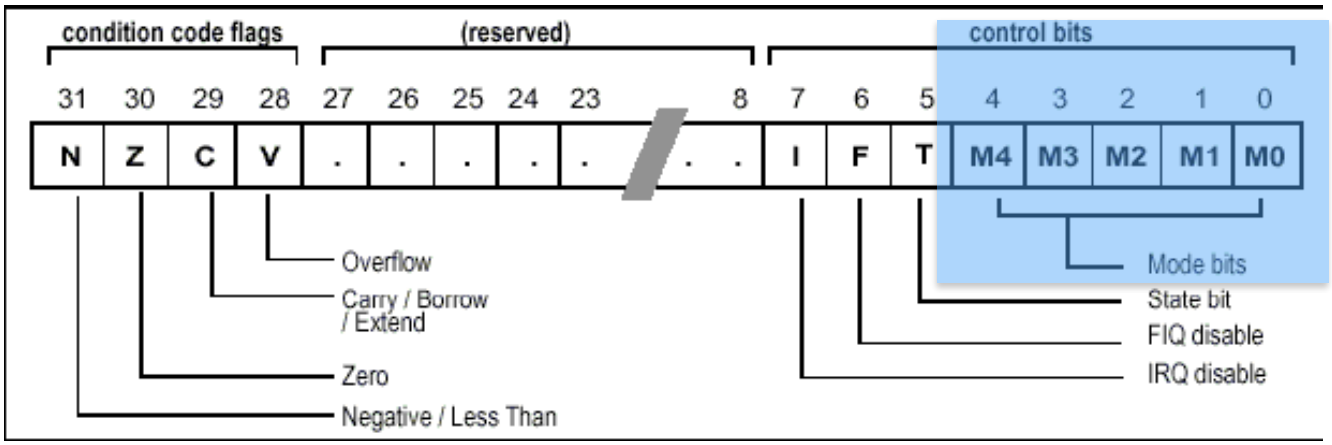
- Consider: processor state, register use, memory
- Hardware support helps out

  (different modes, banked registers)

ARM processor modes

User	unprivileged
IRQ	interrupt
FIQ	fast interrupt
Supervisor	privileged, entered on reset (this is us)
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

```
msr cpsr_c, r0        @ Copy r0 to CPSR
mrs r0, cpsr_c        @ Copy CPSR to r0
```

Per-mode banked registers

Register	supervisor	interrupt
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

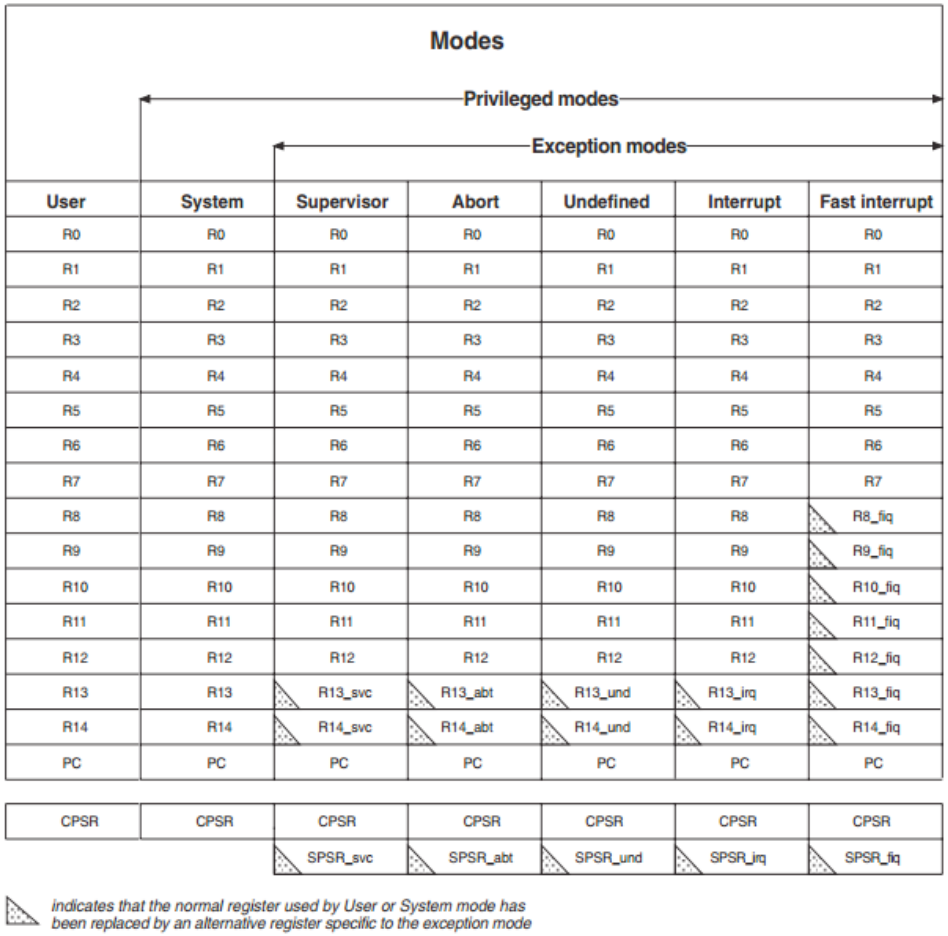


Figure A2-1 Register organization

Interrupts step-by-step

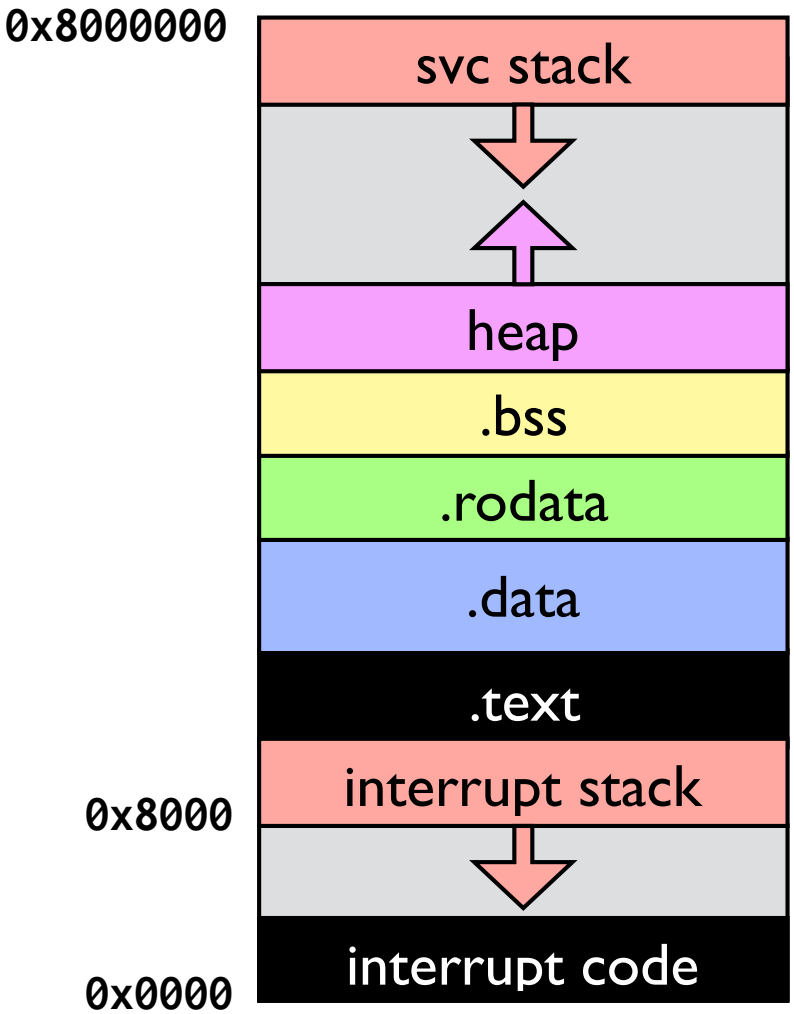
External event triggers interrupt. Processor response:

- Complete current instruction
- Switch processor mode
- Save return address (PC+8) into LR of new mode,
  save CPSR into SPSR
- Disable further interrupts until exit this mode
- Force PC to address 0x18 (location in vector table)
- Software takes over

ARM Interrupts

Normal 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

Start sequence



start.s

```
_start:
    mov r0, #0xD2      @ mode = interrupt
    msr cpsr_c, r0
    mov sp, #0x8000

    mov r0, #0xD3      @ mode = supervisor
    msr cpsr_c, r0
    mov sp, #0x8000000
    mov fp, #0

    bl _cstart
```

cstart.c

```
int *vectorsdst = _RPI_INTERRUPT_VECTOR_BASE;
int *vectors = &_vectors;
int *vectors_end = &_vectors_end;
while (vectors < vectors_end) {
    *vectorsdst++ = *vectors++;
}
```

Symbols `_vectors` and `_vectors_end` used to mark region to be copied to vector table

Relative vs absolute address

```
_vectors:
    b abort_asm
    b abort_asm
    b abort_asm
    b abort_asm
    b abort_asm
    b abort_asm
    b abort_asm
    b interrupt_asm
    b abort_asm
_vectors_end:
```

```
_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
_vectors_end:
```

"position-independent code"

code/vectors

Interrupt vector

```
interrupt_asm:
    sub   lr, lr, #4                  @ compute return add
    push  {r0-r3, r12, lr}            @ save registers
    mov   r0, lr                      @ pass old pc as arg
    bl    interrupt_vector            @ call C function
    ldm   sp!, {r0-r3, r12, pc}^      @ resume, ^ to change
                                         mode + restore cpsr
```

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