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 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?

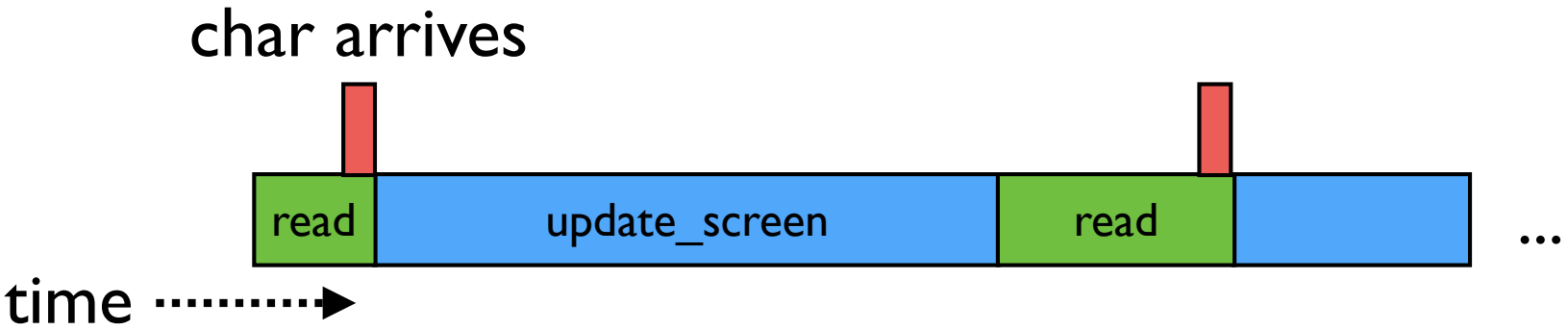
11 bits, clock rate 15kHz

How long does it take to update the screen?

What could go wrong?

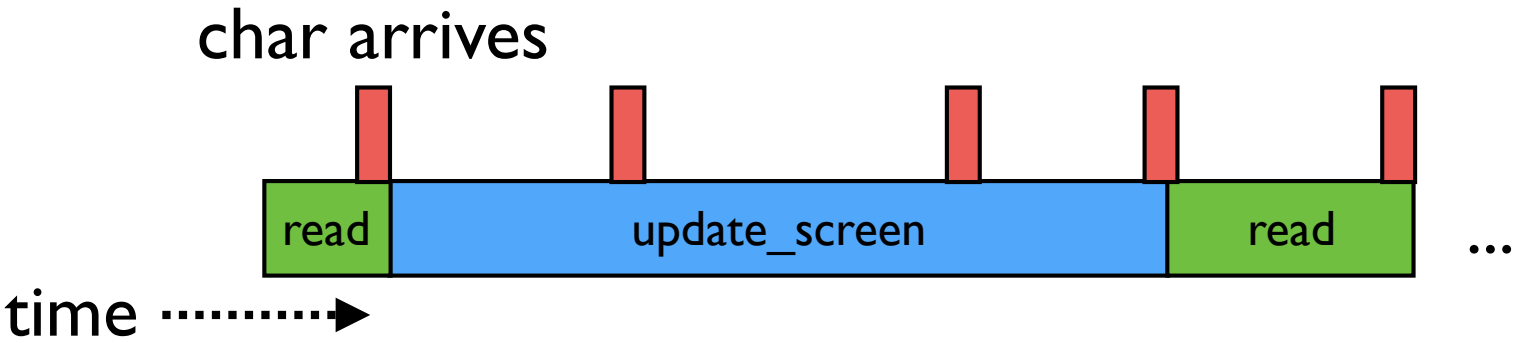
Synchronous I/O

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



Synchronous I/O

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



The Problem

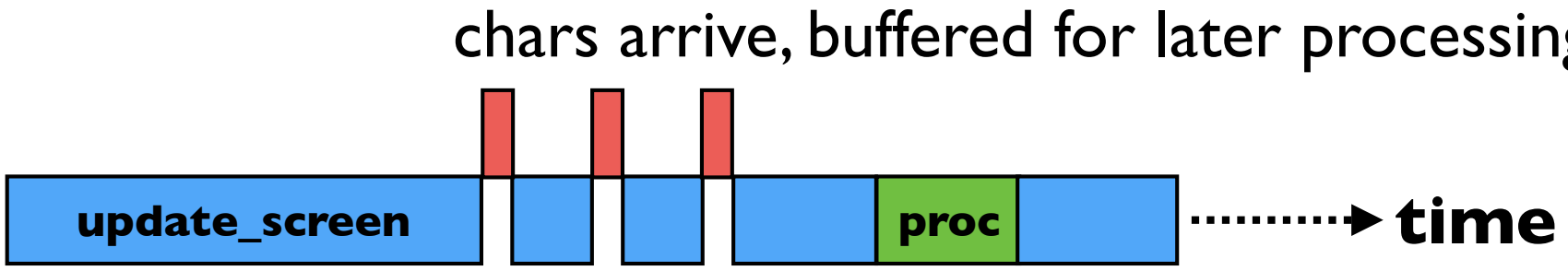
Ongoing and long-running computations (graphics, simulations, applications, …) are keeping CPU occupied, but…

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?

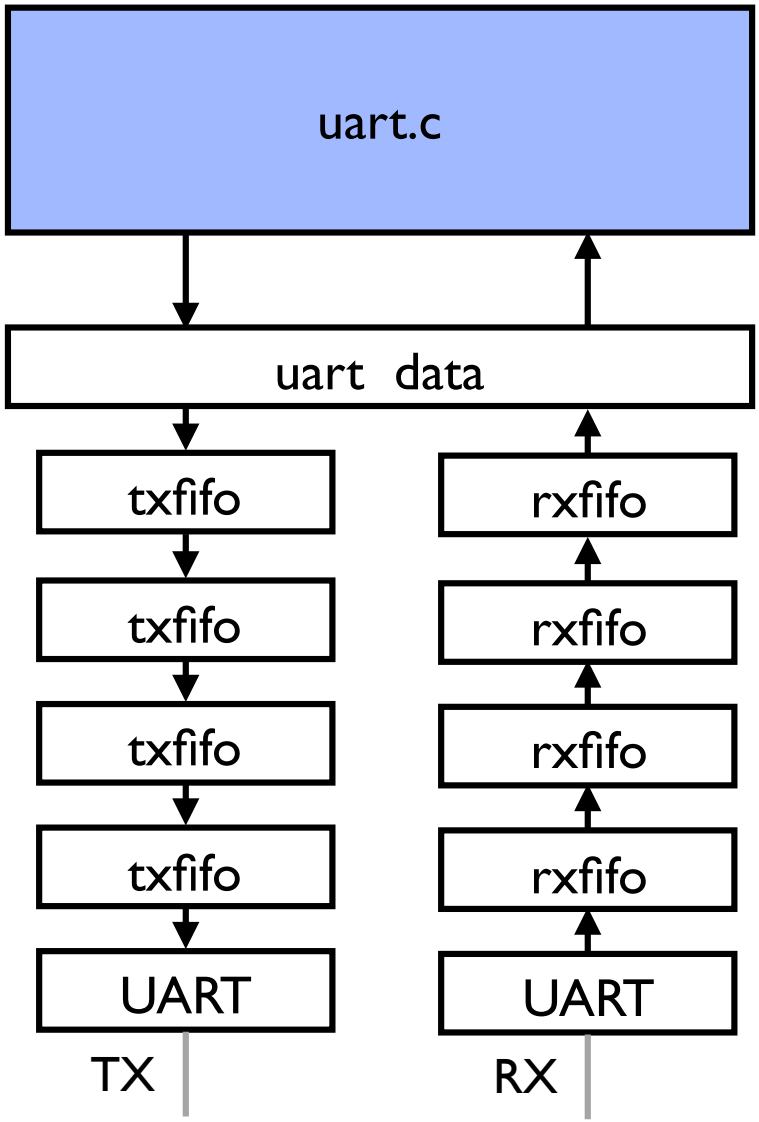
Asynchronous processing

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

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



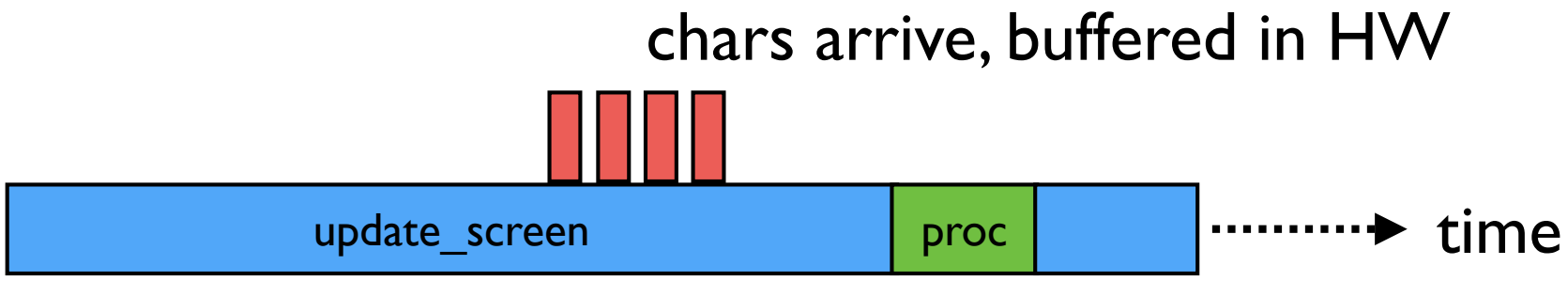
Hardware can help

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

void uart_putchar(int ch)
{
    while (!(uart->lsr & MINI_UART_LSR_TX_EMPTY)) ;
    uart->data = ch;
}
```



Asynchronous I/O (with HW help)

```
while (1) {
  dequeue next
  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

Interrupted control flow

```
static volatile int gCount;
```

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

```
bool button_pressed(unsigned int pc)
{
  if (gpio_check_and_clear_event(BUTTON)) {
     gCount++;
     return true;
   }
   return false;
}
```

23 23 23 23 23
23 23 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

  (different modes, banked registers)

Hardware support for interrupts

Processor executing in a particular "mode"

- Supervisor, interrupt, user, abort, ..
- Reset starts in supervisor mode (that's us!)
- Processor switches mode in response to interrupt

CPSR register tracks current mode, processor state

- Special instructions copy val to regular register to read/write

Banked registers

- unique sp and lr per-mode (sometimes others, too)

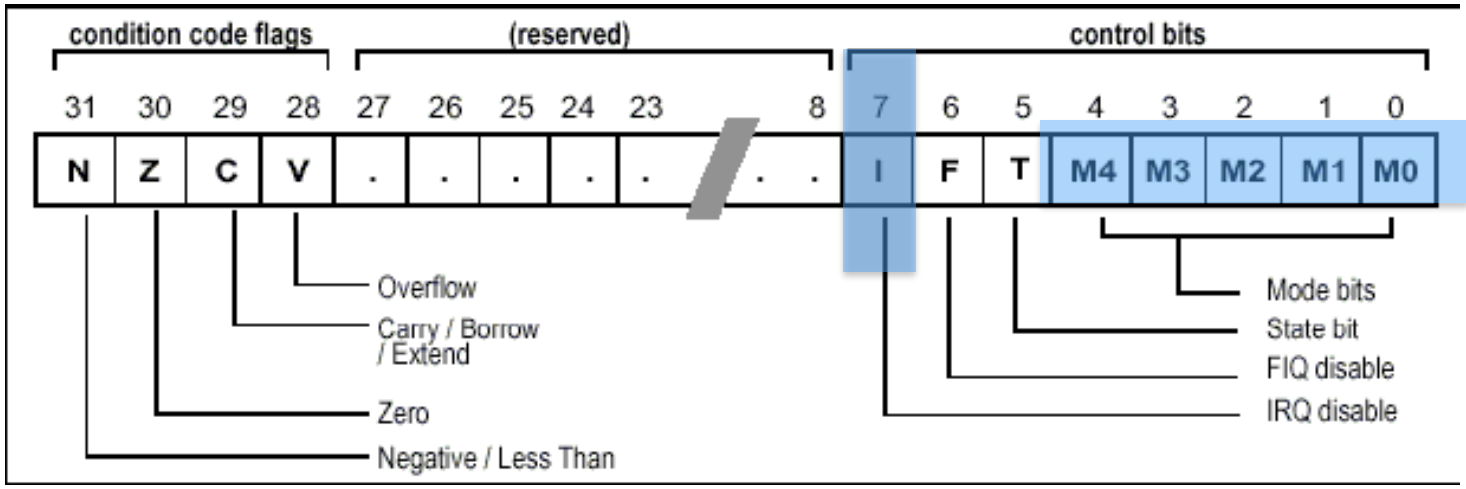
Interrupt vector

- fixed location in memory jumped to on interrupt

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

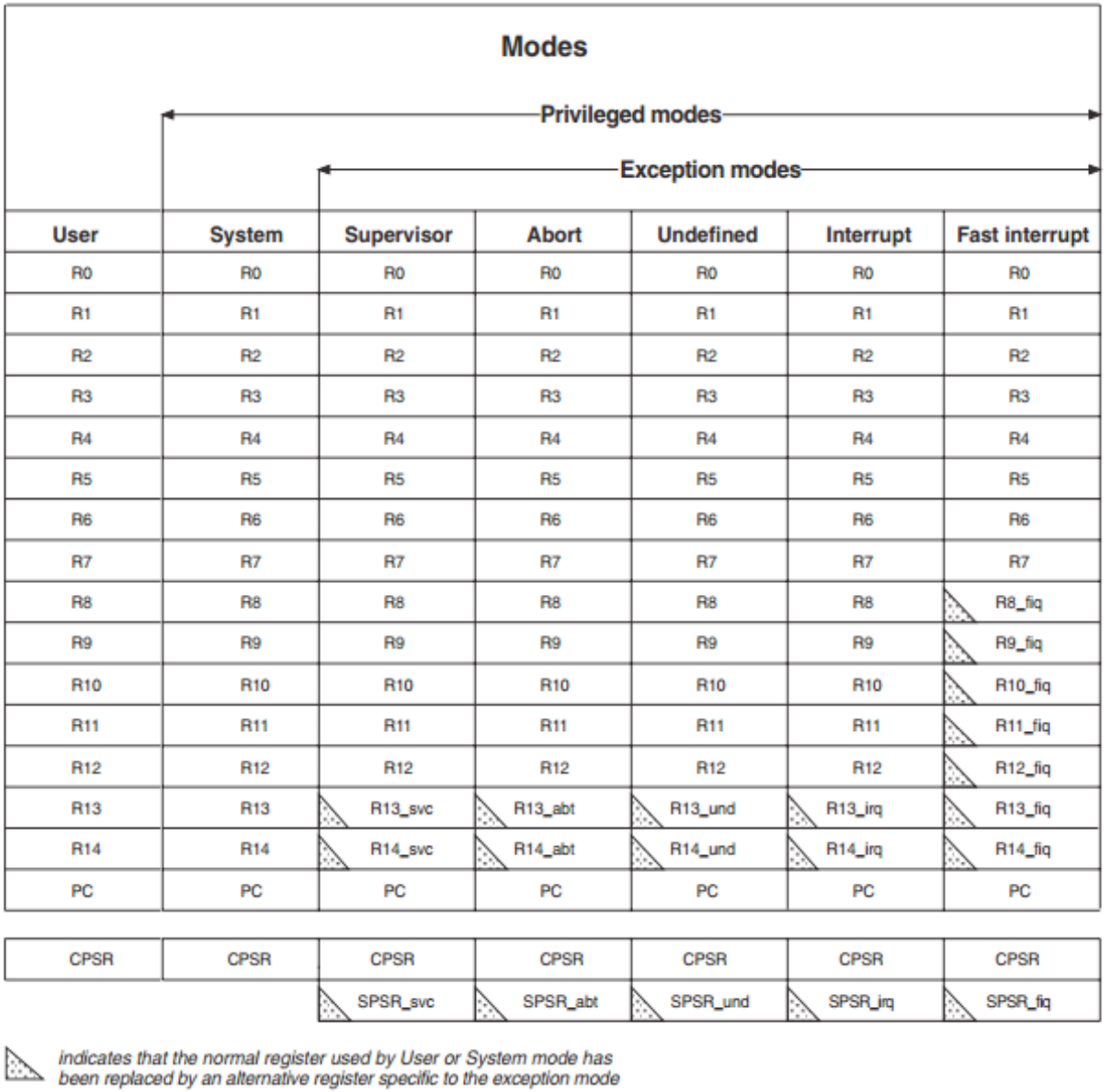


M[4:0]	Mode
b10000	User
b10001	FIQ
b10010	IRQ
b10011	Supervisor
b10111	Abort
b11011	Undefined
b11111	System

```
interrupts_global_enable:
    mrs r0, cpsr
    bic r0, r0, #0x80   @ clear I=0 enables IRQ interrupts
    msr cpsr_c, r0
    bx lr

interrupts_global_disable:
    mrs r0, cpsr
    orr r0, r0, #0x80   @ set I=1 disables IRQ interrupts
    msr cpsr_c, r0
    bx lr
```

Per-mode banked registers

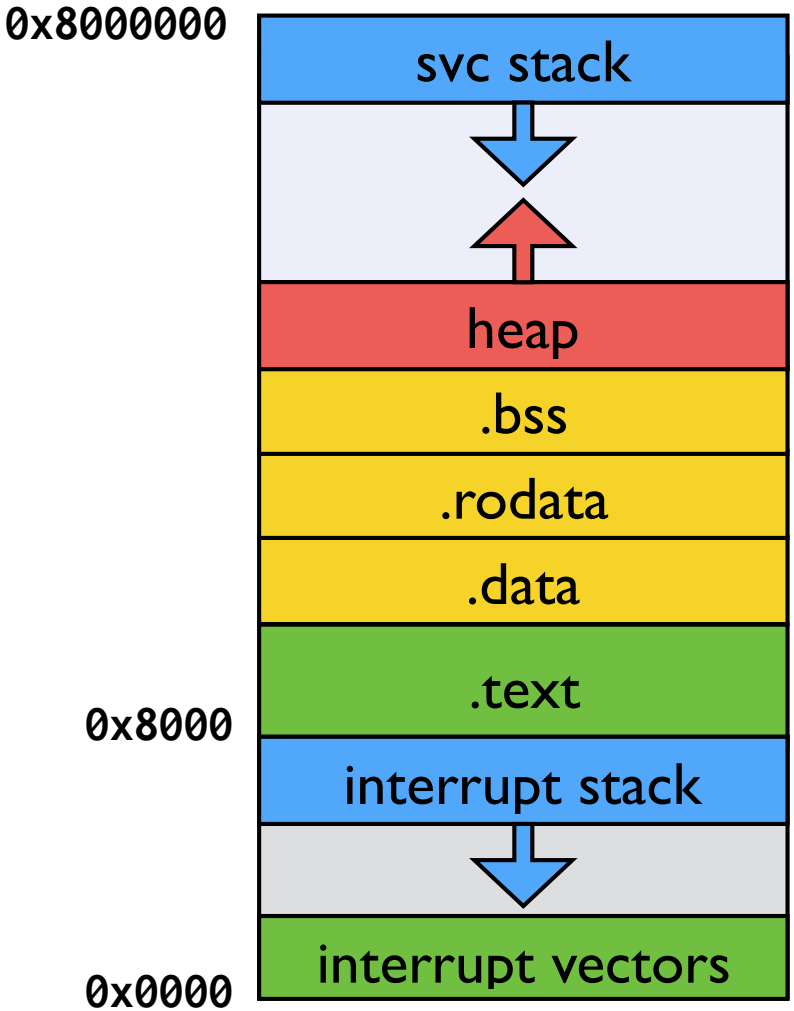
Register	SVC	IRQ
R0	R0	R0
R1	R1	R1
R2	R2	R2
R3	R3	R3
R4	R4	R4
R5	R5	R5
R6	R6	R6
R7	R7	R7
R8	R8	R8
R9	R9	R9
R10	R10	R10
fp	R11	R11
ip	R12	R12
sp	R13_svc	R13_irq
lr	R14_svc	R14_irq
pc	R15	R15
CPSR	CPSR	CPSR
SPSR	SPSR	SPSR



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

Full memory map



When/how to init interrupt stack?

When/how to install vector table?

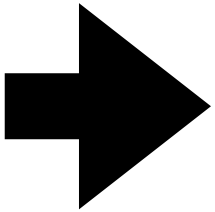
Install vectors

```
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

Relative vs absolute address

```
_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
_vectors_end:
```



```
_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"

```
_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:
```

Symbols **_vectors** and **_vectors_end** mark region to be copied

Interrupt, hardware-side

External event triggers interrupt. Processor response:

- Complete current instruction

- Change processor mode, save return address (PC+8) into LR of new mode, save CPSR into SPSR

Tricky! Needs to happen "simultaneously"…

- Further interrupts disabled until exit this mode

- Force pc address 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 return addr
    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 svc, ^ changes
                                        mode & restore cpsr
```

Could we do steps above in C?

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

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