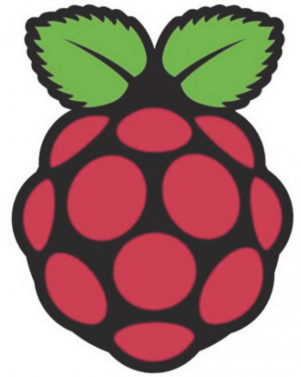




Interrupts

now, we're cooking with gas



Blocking I/O

```
while (1) {  
    read_char_to_screen();  
    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?

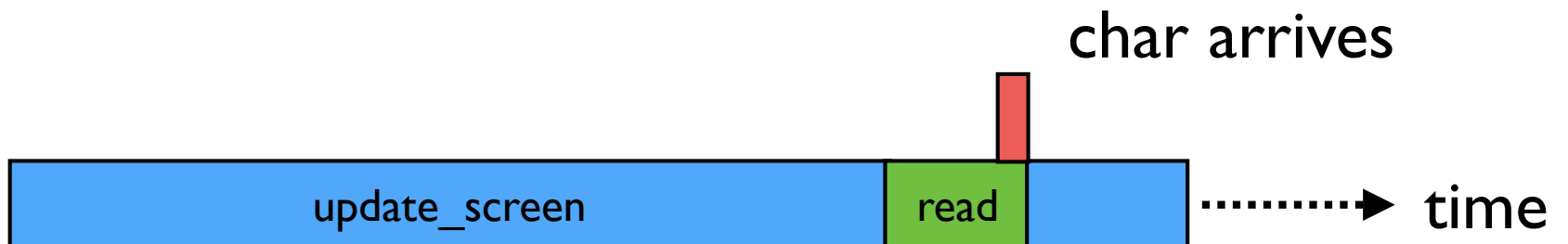
code/button

code/glbutton

code/glkeyboard

Blocking I/O

```
while (1) {  
    read_char_to_screen();  
    update_screen();  
}
```



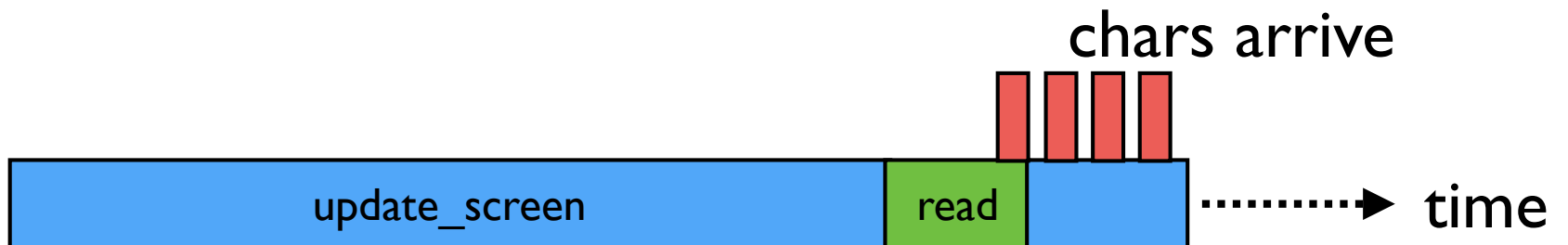
Blocking I/O

```
while (1) {  
    read_char_to_screen();  
    update_screen();  
}
```



Blocking I/O

```
while (1) {  
    read_char_to_screen();  
    update_screen();  
}
```



The Problem

Need long-running computations (graphics, computations, applications, etc.).

Need to respond to external events quickly.

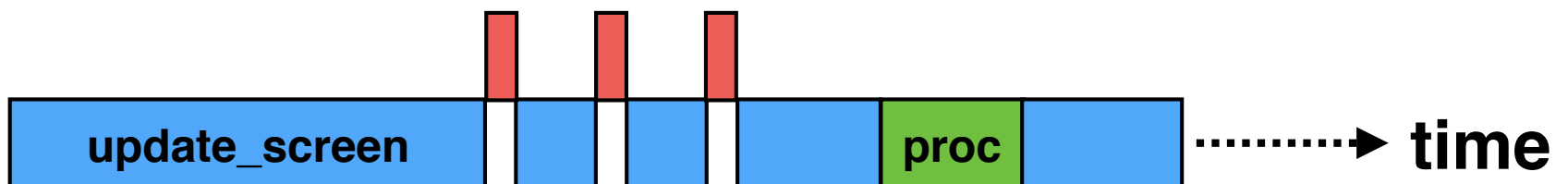
How could we change this code?

```
while (1) {  
    read_char_to_screen( );  
    update_screen( );  
}
```

Concurrency

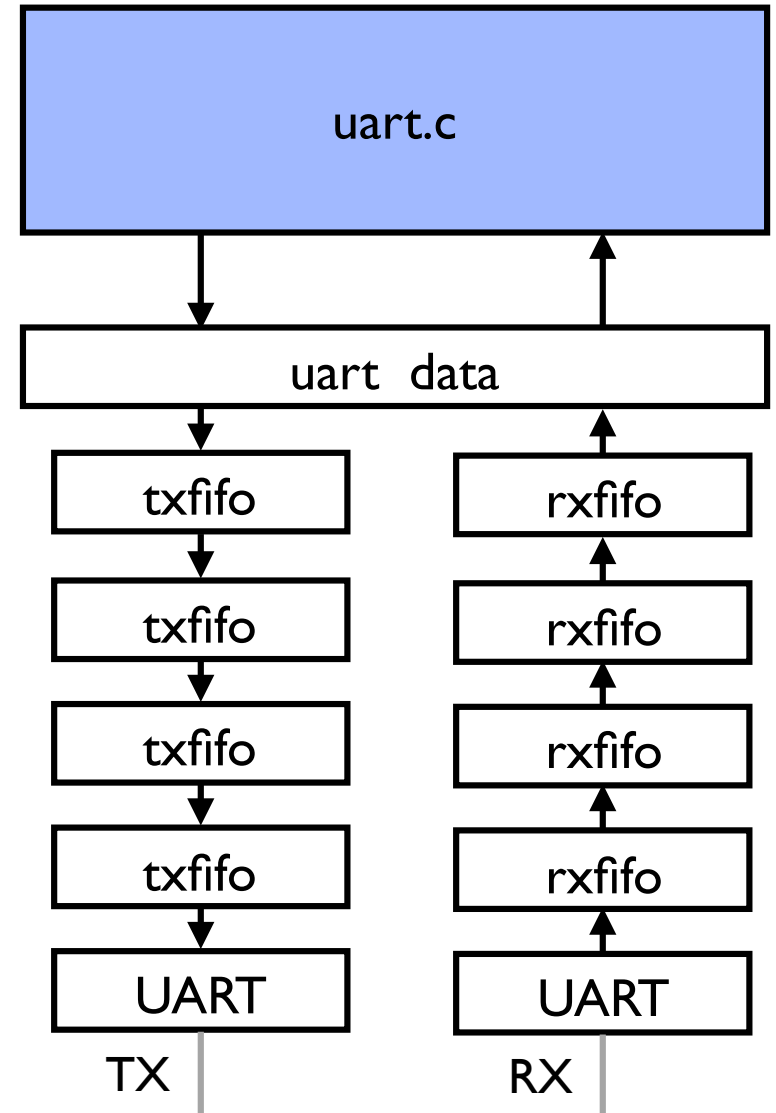
```
when a scan code arrives {  
    add_scan_code_to_buffer();  
}
```

```
while (1) {  
    // Doesn't block  
    while (read_chars_to_screen()) {}  
    update_screen();  
}
```



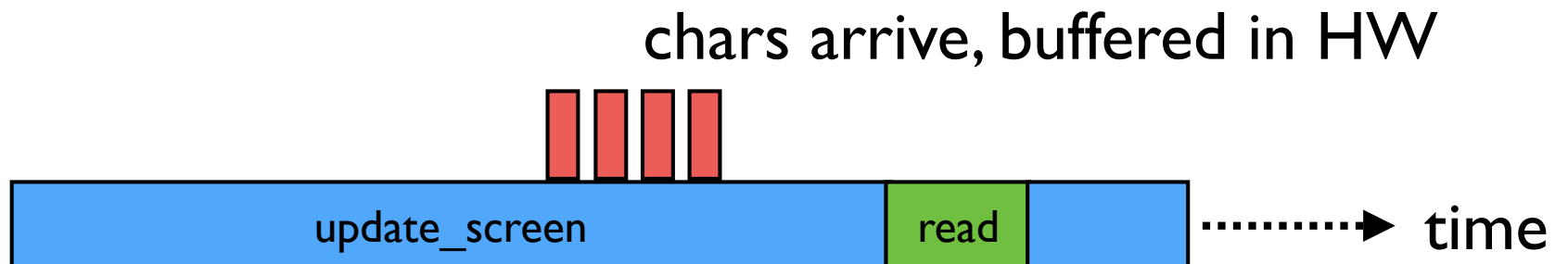
Hardware Can Help

```
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;  
}
```



Blocking I/O (with HW help)

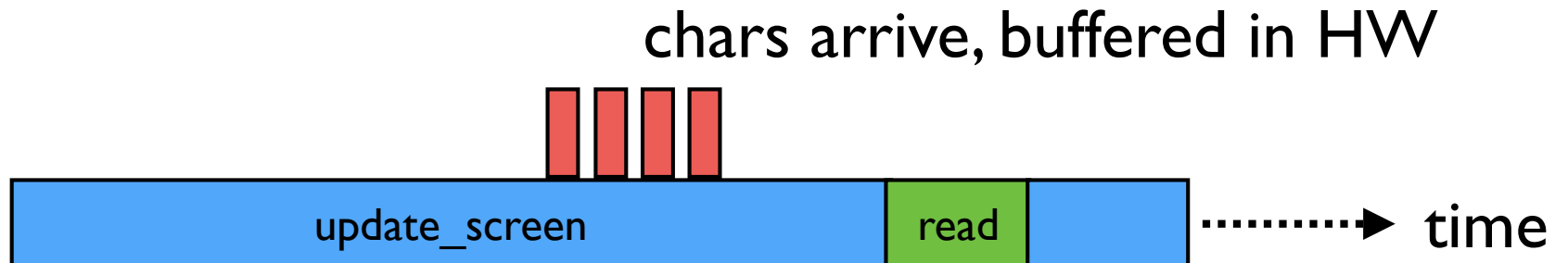
```
while (1) {  
    while (read_chars_to_screen()) {}  
    update_screen();  
}
```



Blocking I/O (with HW help)

```
while (1) {  
    while (read_chars_to_screen()) {}  
    update_screen();  
}
```

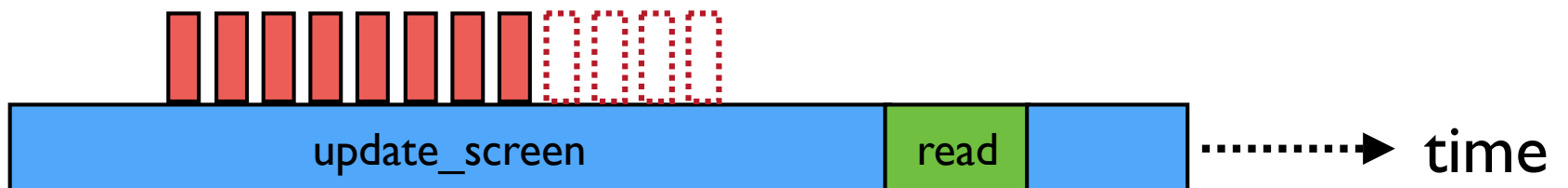
Can we still lose characters?



No Silver Bullet

```
while (1) {  
    while (read_chars_to_screen()) {}  
    update_screen();  
}
```

Yes! Chars overflow FIFO, dropped.



Interrupts!

Cause processor to stop what it's doing and immediately execute other code, returning to original code when done.

- External events (I/O, reset, timer)
- Internal events (bad memory access, software trigger).

Critical for responsive systems

Using interrupts exercises everything you've learned so far.

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

They'll complete your interactive graphics console.

code/blink

Example code

Uses a timer interrupt to increment a counter

- `interrupt_handler` in `main.c`

Increments counter despite `while()` loop in `main()`

start.s

```
interrupt_asm:  
    mov sp, #0x8000  
    sub lr, lr, #4  
  
    push {r0-r12,lr}  
  
    mov r0, lr  
    bl  interrupt_handler
```

What is happening in `interrupt_asm` in `start.s`?
What happens to the stack pointer?
Why do we save all of the registers?

Problem #1

Disassembly of section .text:

```
00008000 <_start>:
    8000:    e3a0d902        mov     sp, #32768      ; 0x8000
    8004:    eb000001        bl      8010 <_cstart>

00008008 <hang>:
    8008:    eb000039        bl      80f4 <led_on>
    800c:    eafffffe        b       800c <hang+0x4>

00008010 <_cstart>:
    8010:    e92d4800        push   {fp, lr} ← Interrupt!
```

Need to know what instruction to return to after interrupt.

Where can we store that information?

Processor Modes

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

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
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq


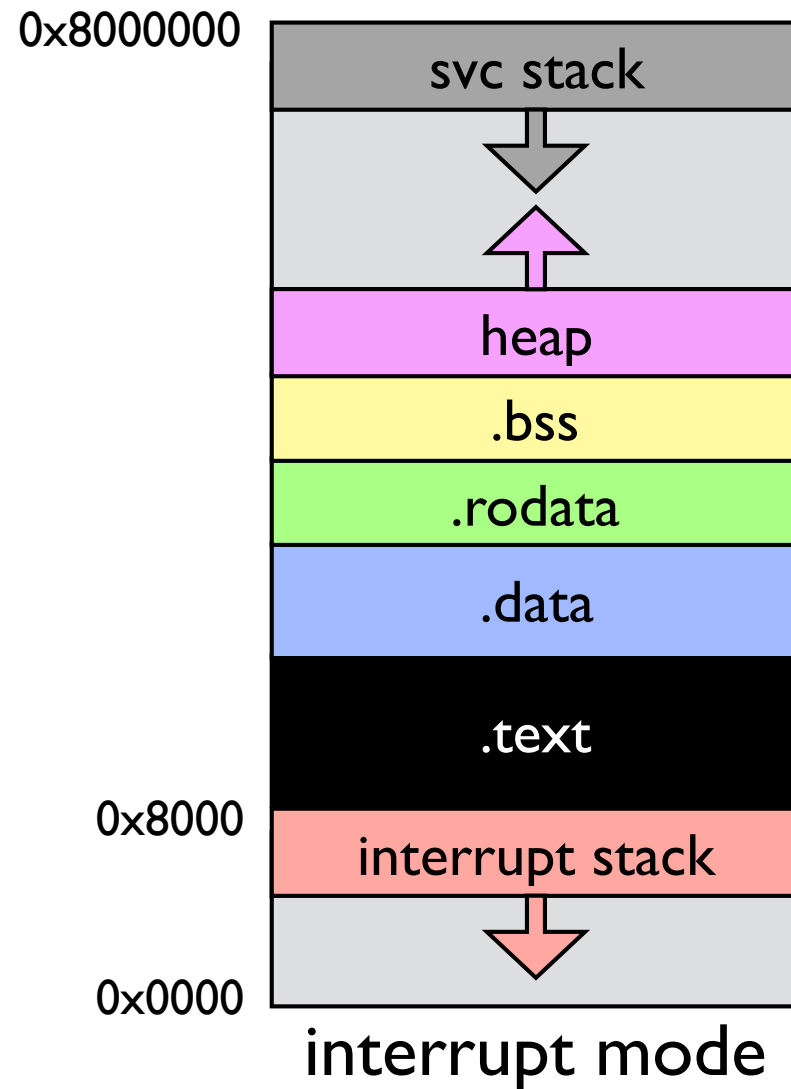
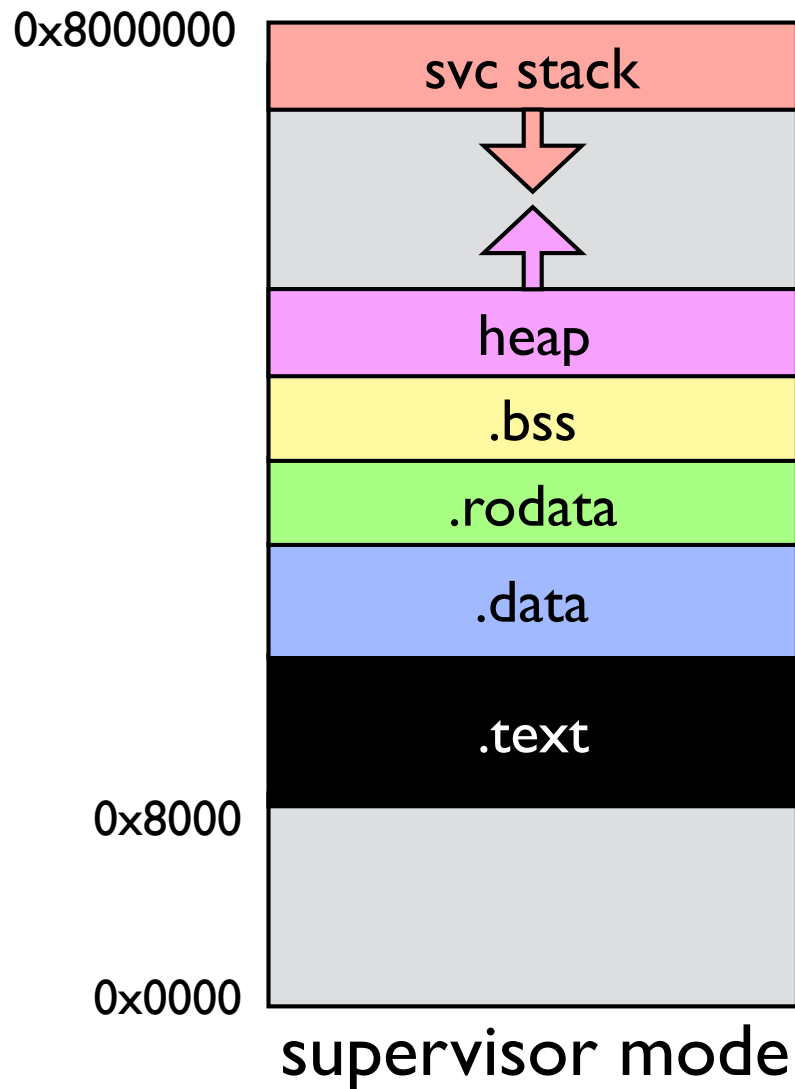
 indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

Figure A2-1 Register organization

Processor Modes, Cont'd



start.s

```
interrupt_asm:  
    mov sp, #0x8000  
    sub lr, lr, #4  
  
    push {r0-r12,lr}  
  
    mov r0, lr  
    bl  interrupt_handler
```


How does the processor know to call `interrupt_asm`?

start.s

```
_vectors:
    ldr pc, _reset_asm
    ldr pc, _undefined_instruction_asm
    ldr pc, _software_interrupt_asm
    ldr pc, _prefetch_abort_asm
    ldr pc, _data_abort_asm
    ldr pc, _reset_asm
    ldr pc, _interrupt_asm
fast_interrupt_asm:
    ldr pc, _fast_asm

_reset_asm:
_undefined_instruction_asm:
_software_interrupt_asm:
_prefetch_abort_asm:
_data_abort_asm:
_interrupt_asm:
_fast_asm:
    .word impossible_asm
    .word impossible_asm
    .word impossible_asm
    .word impossible_asm
    .word impossible_asm
    .word interrupt_asm
    .word impossible_asm
```

the value stored here



function in start.s



Why not `ldr pc, =interrupt_asm`?

cstart.c

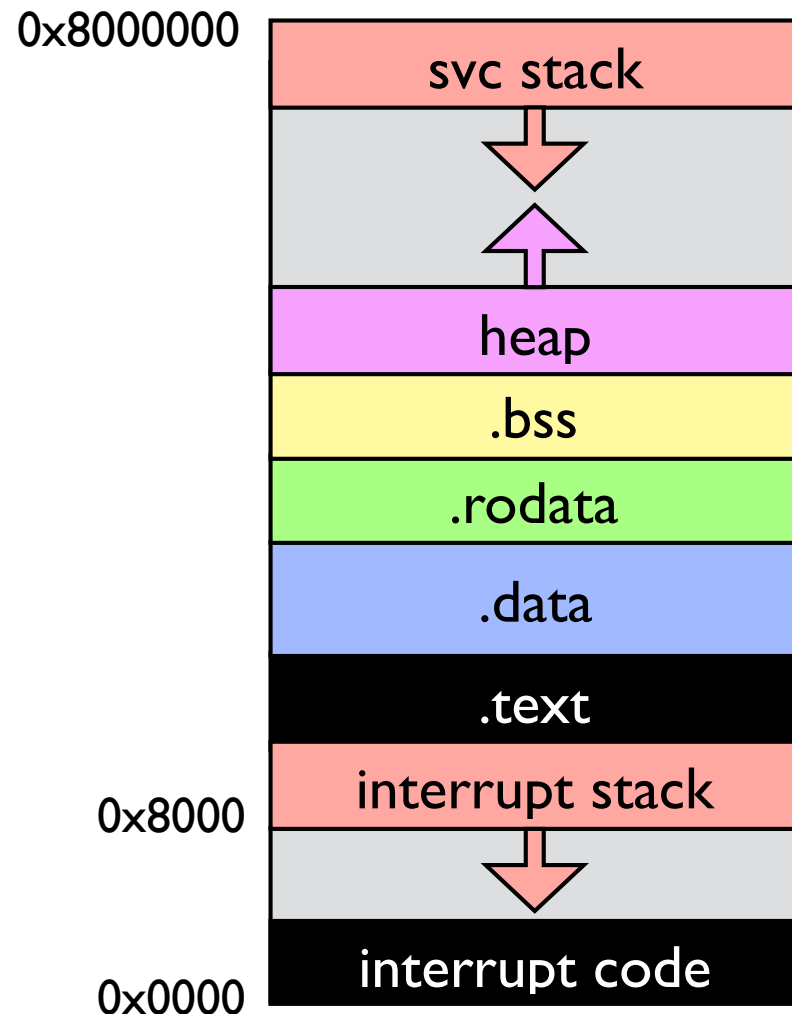
```
#define RPI_VECTOR_START 0x0
```

```
...
```

```
int* vectorsdst = (int*)RPI_VECTOR_START;  
int* vectors = &_amp;vectors;  
int* vectors_end = &_amp;vectors_end;  
while (vectors < vectors_end)  
    *vectorsdst++ = *vectors++;
```

Where are vectors and vectors end defined?

CPU Address Space, Revisited



Desired Assembly

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

00000000:

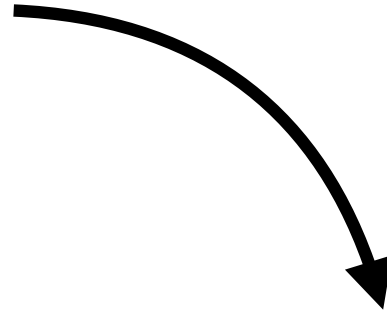
```
0: ldr pc, =impossible_asm
4: ldr pc, =impossible_asm
8: ldr pc, =impossible_asm
c: ldr pc, =impossible_asm
10: ldr pc, =impossible_asm
14: ldr pc, =impossible_asm
18: ldr pc, =interrupt_asm
1c: ldr pc, =impossible_asm
```


Generating Assembly

```
.globl _vectors
```

```
_vectors:
```

```
ldr pc, =impossible_asm  
ldr pc, =impossible_asm  
ldr pc, =impossible_asm  
ldr pc, =impossible_asm  
ldr pc, =impossible_asm  
ldr pc, =impossible_asm  
ldr pc, =interrupt_asm  
ldr pc, =impossible_asm
```



```
0000849c <_vectors>:
```

849c:	e59ff018	ldr	pc, [pc, #24]	; 84bc <_vectors+0x24>
84a0:	e59ff014	ldr	pc, [pc, #20]	; 84bc <_vectors+0x24>
84a4:	e59ff010	ldr	pc, [pc, #16]	; 84bc <_vectors+0x24>
84a8:	e59ff00c	ldr	pc, [pc, #12]	; 84bc <_vectors+0x24>
84ac:	e59ff00c	ldr	pc, [pc, #12]	; 84c0 <_vectors+0x24>
84b0:	e59ff008	ldr	pc, [pc, #8]	; 84c0 <_vectors+0x24>
84b4:	e51ff000	ldr	pc, [pc, #0]	; 84bc <_vectors+0x20>
84b8:	e51ff004	ldr	pc, [pc, #0]	; 84bc <_vectors+0x24>
84bc:	000096c0	.word	0x000096c0	
84c0:	00008290	.word	0x00008290	

Generating Assembly

```
.globl _vectors
```

```
_vectors:
```

```
ldr pc, =impossible_asm  
ldr pc, =impossible_asm  
ldr pc, =impossible_asm  
ldr pc, =impossible_asm  
ldr pc, =impossible_asm  
ldr pc, =impossible_asm  
ldr pc, =interrupt_asm  
ldr pc, =impossible_asm
```

These constants could end up anywhere.

```
0000849c <_vectors>:
```

849c:	e59ff018	ldr	pc, [pc, #24]	; 84bc <_vectors+0x24>
84a0:	e59ff014	ldr	pc, [pc, #20]	; 84bc <_vectors+0x24>
84a4:	e59ff010	ldr	pc, [pc, #16]	; 84bc <_vectors+0x24>
84a8:	e59ff00c	ldr	pc, [pc, #12]	; 84bc <_vectors+0x24>
84ac:	e59ff00c	ldr	pc, [pc, #12]	; 84c0 <_vectors+0x24>
84b0:	e59ff008	ldr	pc, [pc, #8]	; 84c0 <_vectors+0x24>
84b4:	e51ff000	ldr	pc, [pc, #0]	; 84bc <_vectors+0x20>
84b8:	e51ff004	ldr	pc, [pc, #0]	; 84bc <_vectors+0x24>
84bc:	000096c0	.word	0x000096c0	
84c0:	00008290	.word	0x00008290	

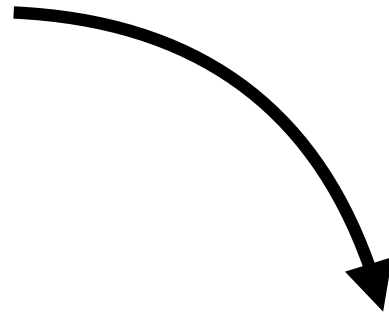
Generating Assembly

```
.globl _vectors
```

```
_vectors:
```

```
ldr pc, =impossible_asm
ldr pc, =impossible_asm
ldr pc, =impossible_asm
ldr pc, =impossible_asm
ldr pc, =impossible_asm
ldr pc, =impossible_asm
ldr pc, =interrupt_asm
ldr pc, =impossible_asm
```

These constants could end up anywhere.



What's funny here?

```
0000849c <_vectors>:
```

849c:	e59ff018	ldr	pc, [pc, #24]	; 84bc <_vectors+0x24>
84a0:	e59ff014	ldr	pc, [pc, #20]	; 84bc <_vectors+0x24>
84a4:	e59ff010	ldr	pc, [pc, #16]	; 84bc <_vectors+0x24>
84a8:	e59ff00c	ldr	pc, [pc, #12]	; 84bc <_vectors+0x24>
84ac:	e59ff00c	ldr	pc, [pc, #12]	; 84c0 <_vectors+0x24>
84b0:	e59ff008	ldr	pc, [pc, #8]	; 84c0 <_vectors+0x24>
84b4:	e51ff000	ldr	pc, [pc, #0]	; 84bc <_vectors+0x20>
84b8:	e51ff004	ldr	pc, [pc, #0]	; 84bc <_vectors+0x24>
84bc:	000096c0	.word	0x000096c0	
84c0:	00008290	.word	0x00008290	



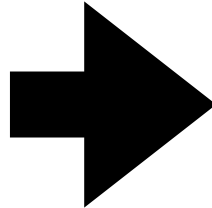
code/vectors

Explicit Embedding

```
.globl _vectors
```

```
_vectors:
```

```
ldr pc, =impossible_asm  
ldr pc, =impossible_asm  
ldr pc, =impossible_asm  
ldr pc, =impossible_asm  
ldr pc, =impossible_asm  
ldr pc, =impossible_asm  
ldr pc, =interrupt_asm  
ldr pc, =impossible_asm
```



```
.globl _vectors
```

```
_vectors:
```

```
ldr pc, _impossible_asm  
ldr pc, _impossible_asm  
ldr pc, _impossible_asm  
ldr pc, _impossible_asm  
ldr pc, _impossible_asm  
ldr pc, _impossible_asm  
ldr pc, _impossible_asm  
ldr pc, _interrupt_asm  
ldr pc, _impossible_asm
```

```
_impossible_asm:    .word impossible_asm  
_interrupt_asm:    .word interrupt_asm
```

Now we know the constants will follow the code.

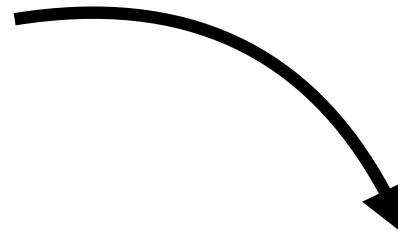
C Code

```
#define RPI_VECTOR_START 0x0
```

...

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

```
000080b8 <_vectors>:
80b8:      e59ff018      ldr    pc, [pc, #24] ; 80d8 <_impossible_asm>
80bc:      e59ff014      ldr    pc, [pc, #20] ; 80d8 <_impossible_asm>
80c0:      e59ff010      ldr    pc, [pc, #16] ; 80d8 <_impossible_asm>
80c4:      e59ff00c      ldr    pc, [pc, #12] ; 80d8 <_impossible_asm>
80c8:      e59ff008      ldr    pc, [pc, #8] ; 80d8 <_impossible_asm>
80cc:      e59ff004      ldr    pc, [pc, #4] ; 80d8 <_impossible_asm>
80d0:      e59ff004      ldr    pc, [pc, #4] ; 80dc <_interrupt_asm>
80d4:      e51ff004      ldr    pc, [pc, #-4] ; 80d8 <_impossible_asm>
80d8:      000080fc      .word  0x000080fc
80dc:      000080e0      .word  0x000080e0
```



```
00000000 <_vectors>:
0000:      e59ff018      ldr    pc, [pc, #24] ; 80d8 <_impossible_asm>
0004:      e59ff014      ldr    pc, [pc, #20] ; 80d8 <_impossible_asm>
0008:      e59ff010      ldr    pc, [pc, #16] ; 80d8 <_impossible_asm>
000c:      e59ff00c      ldr    pc, [pc, #12] ; 80d8 <_impossible_asm>
0010:      e59ff008      ldr    pc, [pc, #8] ; 80d8 <_impossible_asm>
0014:      e59ff004      ldr    pc, [pc, #4] ; 80d8 <_impossible_asm>
0018:      e59ff004      ldr    pc, [pc, #4] ; 80dc <_interrupt_asm>
001c:      e51ff004      ldr    pc, [pc, #-4] ; 80d8 <_impossible_asm>
0020:      000080fc      .word  0x000080fc
0024:      000080e0      .word  0x000080e0
```

Summary

Interrupts allow external events to trigger code to run with very little delay: responsiveness despite long-running functions

- They bring together everything you've learned so far

Running code at arbitrary points is dangerous!

- Copies of `lr` and `sp`, use separate stack

Interrupt vectors are at `0x0-0x1c`

- Have to copy them there at boot time
- Generating safe assembly requires explicitly embedding addresses

Next time: making interrupts happen (the return of GPIO)