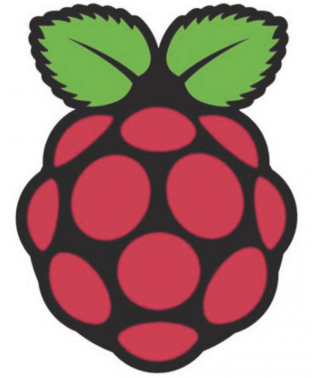


Interrupts



Pardon the Interruption

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

Friday

Using interrupts

Sharing data safely with interrupt code



Synchronous I/O

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

How long does it take to send a scan code?

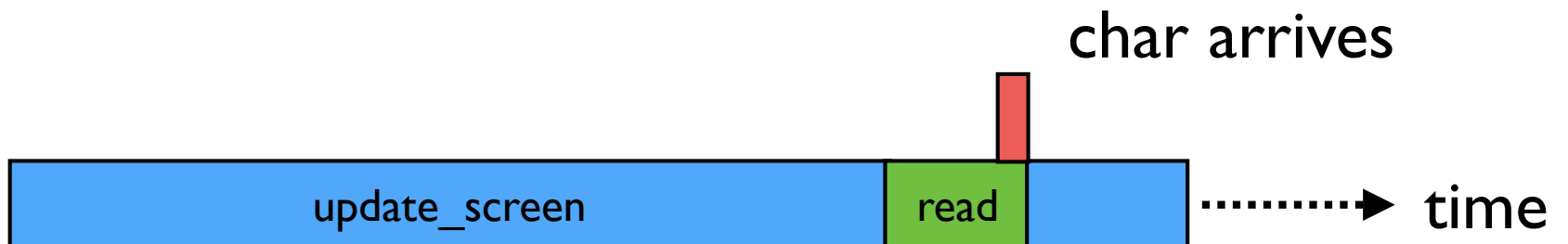
11 bits, clock rate 15kHz

How long does it take to update the screen?

What could go wrong?

Blocking I/O

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



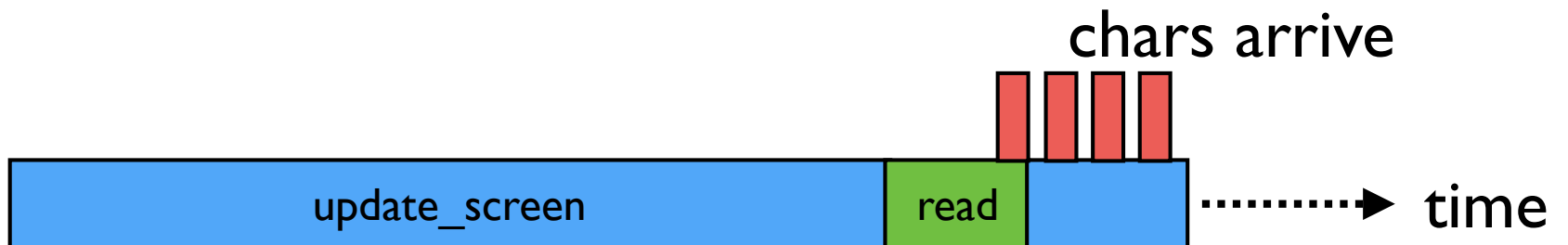
Blocking I/O

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



Blocking I/O

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



code/button-blocking

The Problem

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

Need to respond to external events quickly.

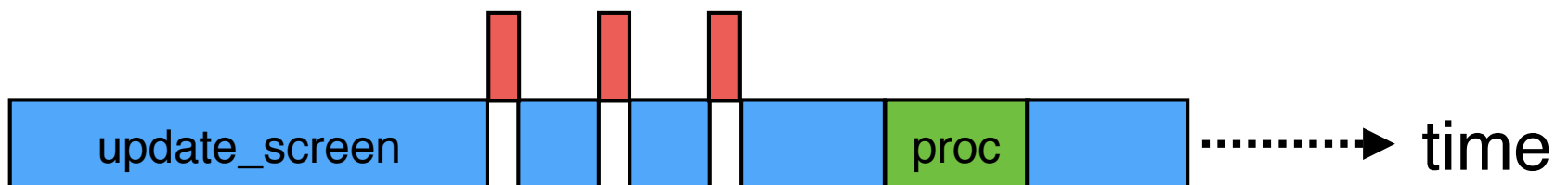
How could we change this code?

```
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();  
}
```

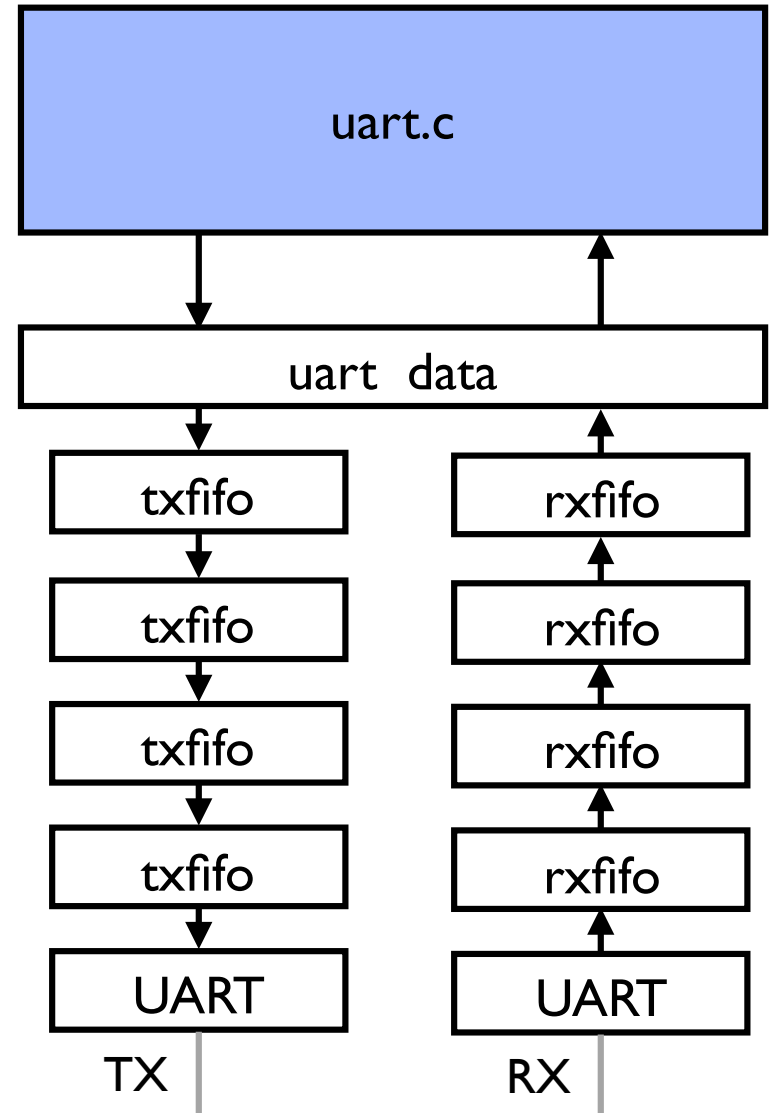


Hardware can help

```
bool uart_haschar(void)
{
    return (uart->lsr & MINI_UART_LSR_RX_READY);
}

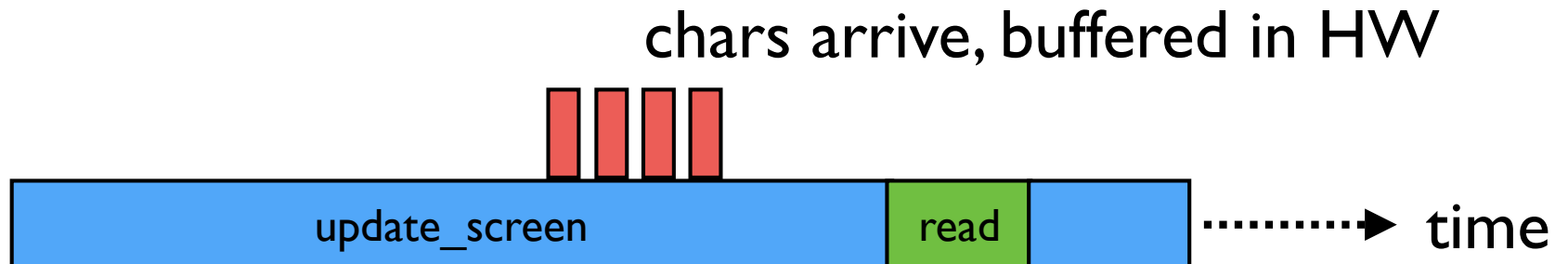
unsigned char uart_recv(void)
{
    while (!uart_haschar()) ;
    return uart->data & 0xFF;
}

void uart_send(unsigned char byte)
{
    while (!(uart->lsr & MINI_UART_LSR_TX_EMPTY)) ;
    uart->data = byte & 0xFF;
}
```



Blocking I/O (HW help)

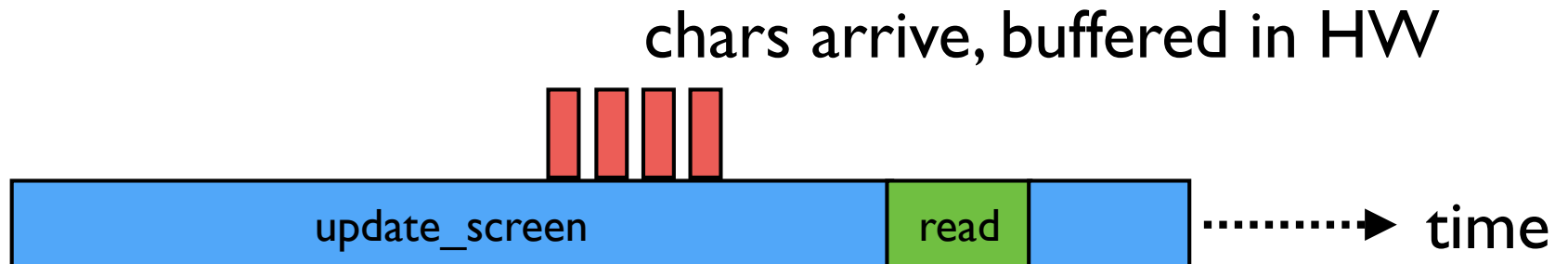
```
while (1) {  
    while (read_chars_to_screen()) {}  
    update_screen();  
}
```



Blocking I/O (HW help)

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

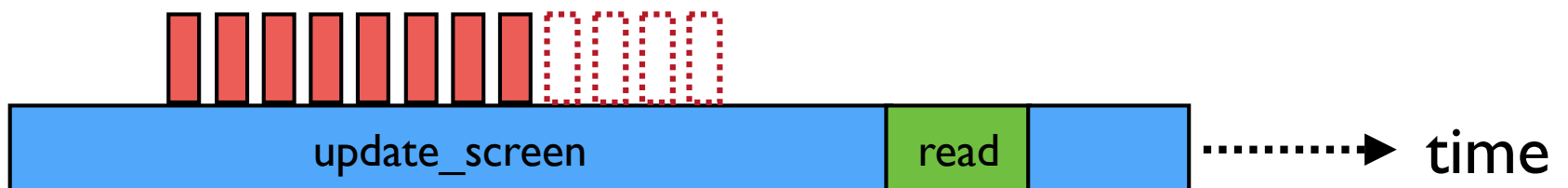
Can we still lose characters?



Blocking I/O (HW help)

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

Yes! Chars overflow FIFO, dropped.



Interrupts to the rescue!

Cause processor to pause what it's doing and immediately execute interrupt code

- 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-interrupt

Example code

Uses a button interrupt to increment a counter

- `interrupt_handler` in `main.c`

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

interrupts_asm.s

```
interrupt_asm:  
    mov     sp, #0x8000  
    sub     lr, lr, #4  
    push    {r0-r12, lr}  
    mov     r0, lr  
    bl      interrupt_dispatch  
    ldm     sp!, {r0-r12, pc}^
```

What is happening in `interrupt_asm`?
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?

We Need To

1. Set up the interrupt stack.

2. Install interrupt handler code.

3. Tell CPU when to trigger interrupts.

- When PS/2 clock line has a falling edge

4. Enable interrupts!

5. Writing safe interrupt handlers.

- How do you share state that can be modified at any time?

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


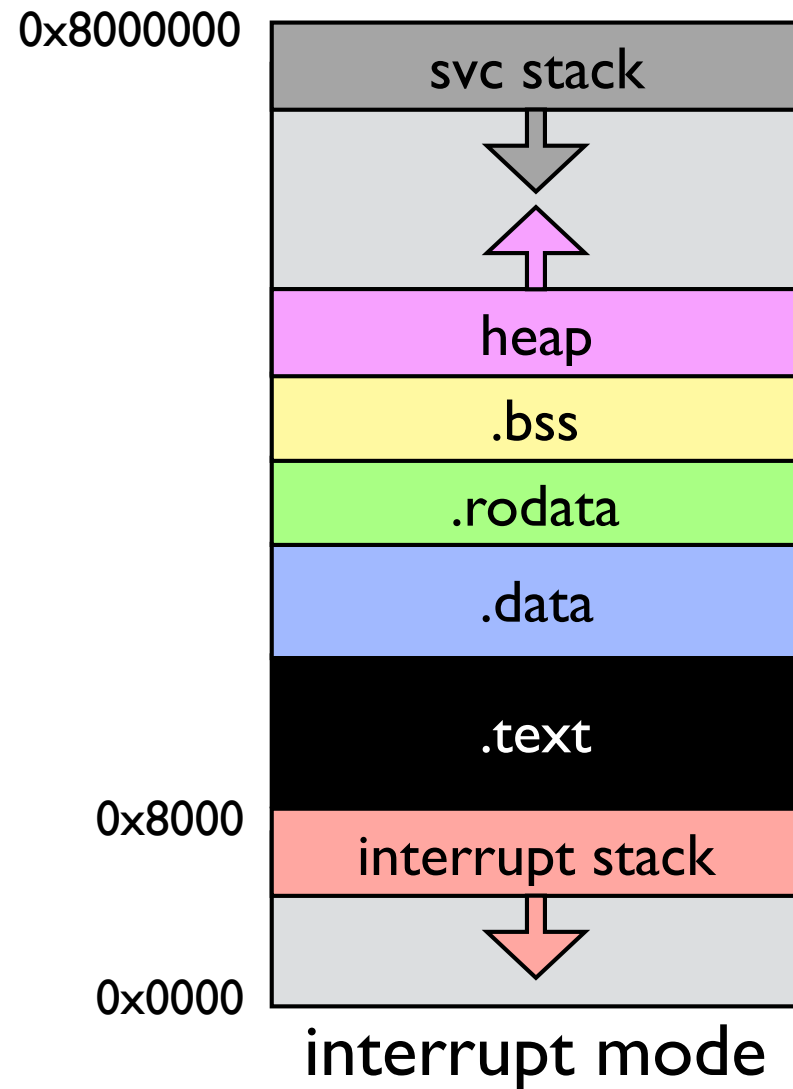
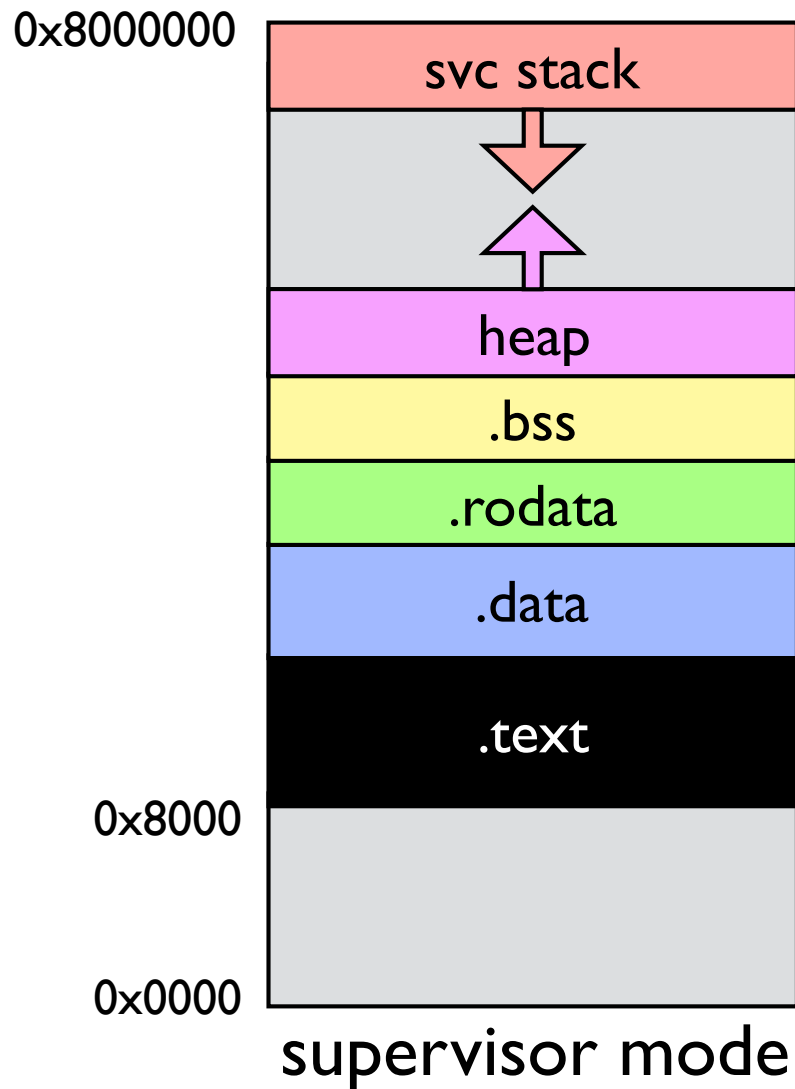
 indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

Figure A2-1 Register organization

Processor Modes, Cont'd



interrupts_asm.s

```
iinterrupt_asm:
    mov     sp, #0x8000
    sub     lr, lr, #4
    push    {r0-r12, lr}
    mov     r0, lr
    bl      interrupt_dispatch
    ldm     sp!, {r0-r12, pc}^
```

- _

How does the processor know to call `interrupt_asm`?

We Need To

1. Set up the interrupt stack.
2. Install interrupt handler code.
3. Tell CPU when to trigger interrupts.
 - When PS/2 clock line has a falling edge
4. Enable interrupts!
5. Writing safe interrupt handlers.
 - How do you share state that can be modified at any time?

cstart.c

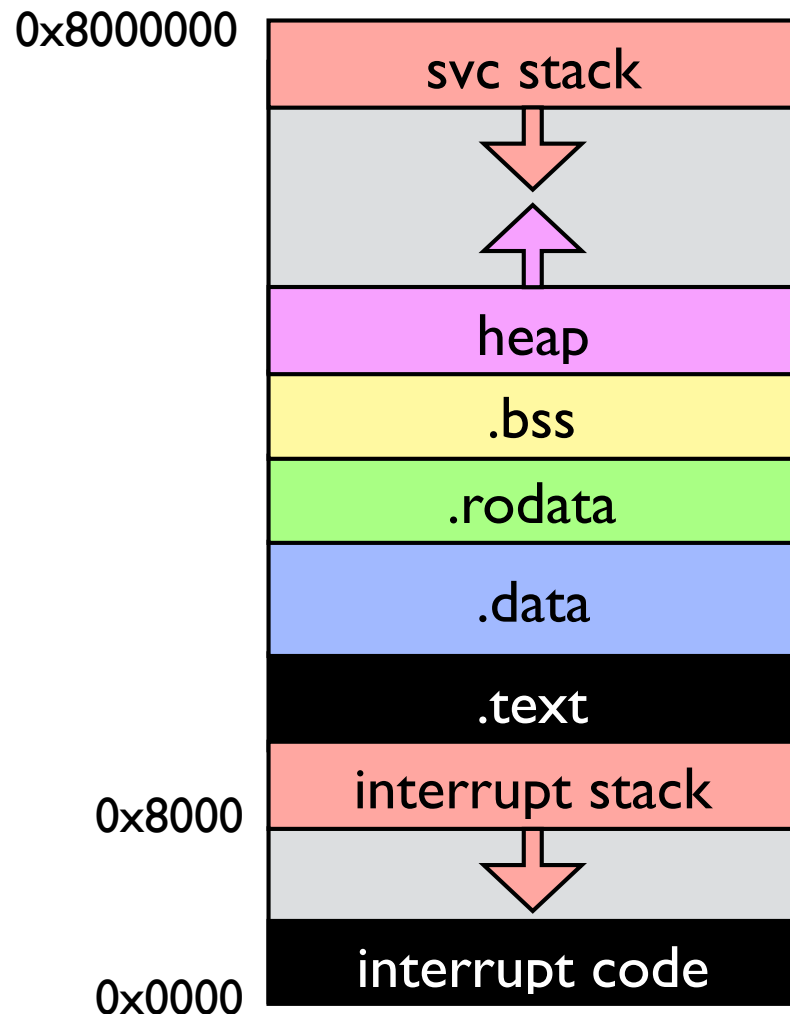
```
#define RPI_VECTOR_START 0x0
```

```
...
```

```
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



code/vectors

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
```

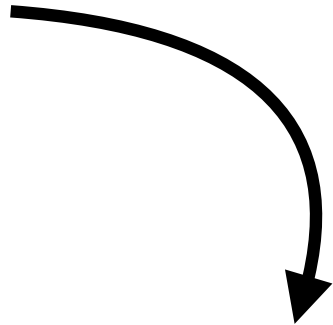
```
  14: b abort_asm
```

```
  18: b interrupt_asm
```

```
 1c: b abort_asm
```

Use Branch Instructions

```
.globl _vectors
_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
```

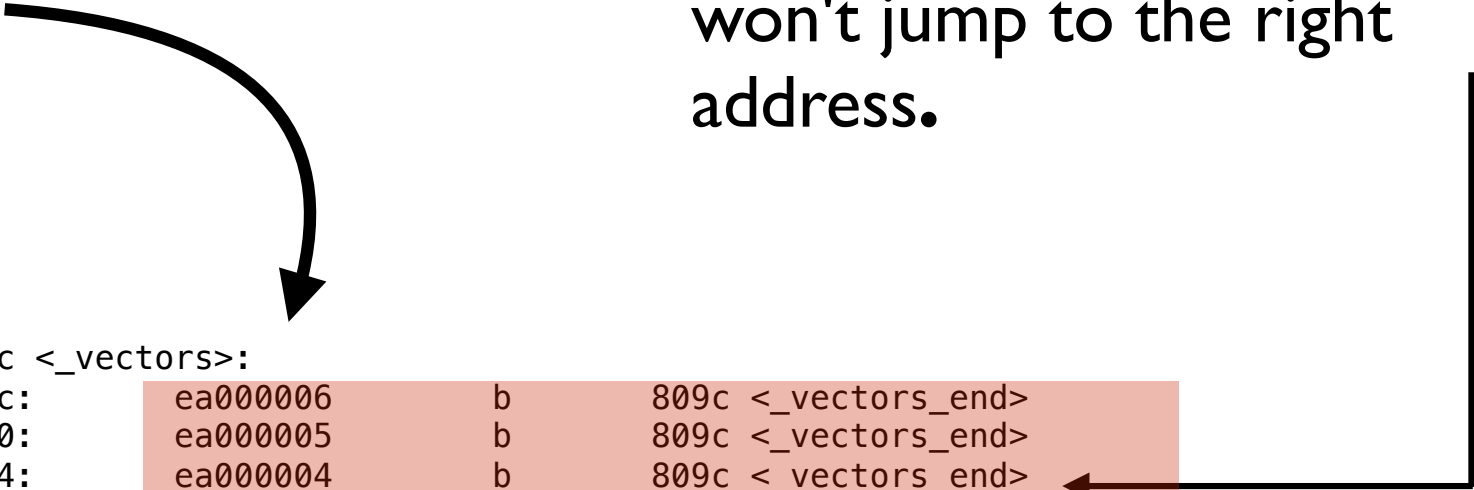


0000807c <_vectors>:			
807c:	ea000006	b	809c <_vectors_end>
8080:	ea000005	b	809c <_vectors_end>
8084:	ea000004	b	809c <_vectors_end>
8088:	ea000003	b	809c <_vectors_end>
808c:	ea000002	b	809c <_vectors_end>
8090:	ea000001	b	809c <_vectors_end>
8094:	ea000001	b	80a0 <interrupt_asm>
8098:	eaffffff	b	809c <_vectors_end>
0000809c <_vectors_end>:			
809c:	eaffffff	b	809c <_vectors_end>
000080a0 <interrupt_asm>:			
80a0:	e3a0d902	mov	sp, #32768 ; 0x8000

Use Branch Instructions

```
.globl _vectors
_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
```

These are relative jumps.
If we move the code, they
won't jump to the right
address.



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

0000809c <_vectors_end>:
 809c: eaffffff      b      809c <_vectors_end>

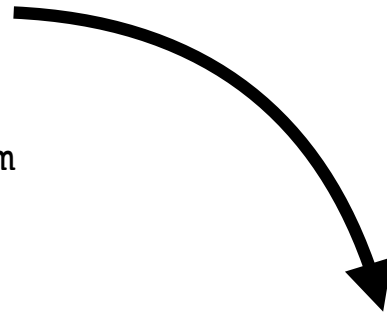
000080a0 <interrupt_asm>:
 80a0: e3a0d902      mov     sp, #32768      ; 0x8000
```

Load Address Explicitly

```
.globl _vectors
_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
_vectors_end:
```

Load Address Explicitly

```
.globl _vectors
_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
_vectors_end:
```



```
0000807c <_vectors>:
    807c:      e59ff034      ldr    pc, [pc, #52]    ; 80b8 <interrupt_asm+0x18>
    8080:      e59ff030      ldr    pc, [pc, #48]    ; 80b8 <interrupt_asm+0x18>
    8084:      e59ff02c      ldr    pc, [pc, #44]    ; 80b8 <interrupt_asm+0x18>
    8088:      e59ff028      ldr    pc, [pc, #40]    ; 80b8 <interrupt_asm+0x18>
    808c:      e59ff024      ldr    pc, [pc, #36]    ; 80b8 <interrupt_asm+0x18>
    8090:      e59ff020      ldr    pc, [pc, #32]    ; 80b8 <interrupt_asm+0x18>
    8094:      e59ff020      ldr    pc, [pc, #32]    ; 80bc <interrupt_asm+0x1c>
    8098:      e59ff018      ldr    pc, [pc, #24]    ; 80b8 <interrupt_asm+0x18>

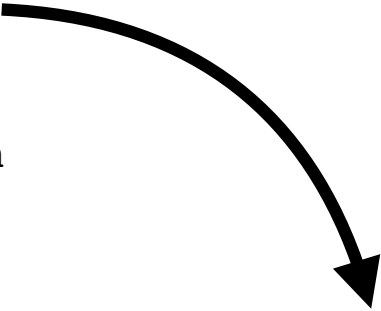
0000809c <_vectors_end>:
    809c:      eaffffff      b      809c <_vectors_end>

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

Load Address Explicitly

```
.globl _vectors
_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
_vectors_end:
```

Also gets turned into a relative load. If we move this code it won't work.



0000807c <_vectors>:				
807c:	e59ff034	ldr	pc, [pc, #52]	; 80b8 <interrupt_asm+0x18>
8080:	e59ff030	ldr	pc, [pc, #48]	; 80b8 <interrupt_asm+0x18>
8084:	e59ff02c	ldr	pc, [pc, #44]	; 80b8 <interrupt_asm+0x18>
8088:	e59ff028	ldr	pc, [pc, #40]	; 80b8 <interrupt_asm+0x18>
808c:	e59ff024	ldr	pc, [pc, #36]	; 80b8 <interrupt_asm+0x18>
8090:	e59ff020	ldr	pc, [pc, #32]	; 80b8 <interrupt_asm+0x18>
8094:	e59ff020	ldr	pc, [pc, #32]	; 80bc <interrupt_asm+0x1c>
8098:	e59ff018	ldr	pc, [pc, #24]	; 80b8 <interrupt_asm+0x18>

0000809c <_vectors_end>:			
809c:	eaaffffe	b	809c <_vectors_end>

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

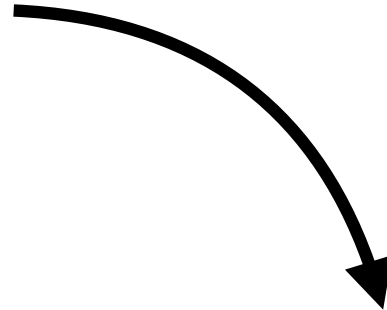
Explicit Address v2

(if functions defined in different file so compiler can't use a relative load since their location is not known)

```
.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
```



```
0000807c <_vectors>:
```

807c:	e59ff018	ldr	pc, [pc, #24]	; 809c <_vectors_end>
8080:	e59ff014	ldr	pc, [pc, #20]	; 809c <_vectors_end>
8084:	e59ff010	ldr	pc, [pc, #16]	; 809c <_vectors_end>
8088:	e59ff00c	ldr	pc, [pc, #12]	; 809c <_vectors_end>
808c:	e59ff008	ldr	pc, [pc, #8]	; 809c <_vectors_end>
8090:	e59ff004	ldr	pc, [pc, #4]	; 809c <_vectors_end>
8094:	e59ff004	ldr	pc, [pc, #4]	; 80a0 <_vectors_end+0x4>
8098:	e51ff004	ldr	pc, [pc, #-4]	; 809c <_vectors_end>

```
0000809c <_vectors_end>:
```

809c:	000080a4	.word	0x000080a4
80a0:	000080a8	.word	0x000080a8

Explicit Address v2

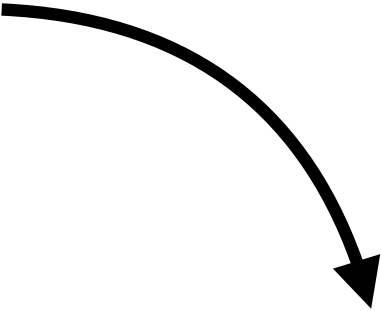
(if functions defined in different file so compiler can't use a relative load since their location is not known)

```
.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.



```
0000807c <_vectors>:
```

807c:	e59ff018	ldr	pc, [pc, #24]	; 809c <_vectors_end>
8080:	e59ff014	ldr	pc, [pc, #20]	; 809c <_vectors_end>
8084:	e59ff010	ldr	pc, [pc, #16]	; 809c <_vectors_end>
8088:	e59ff00c	ldr	pc, [pc, #12]	; 809c <_vectors_end>
808c:	e59ff008	ldr	pc, [pc, #8]	; 809c <_vectors_end>
8090:	e59ff004	ldr	pc, [pc, #4]	; 809c <_vectors_end>
8094:	e59ff004	ldr	pc, [pc, #4]	; 80a0 <_vectors_end+0x4>
8098:	e51ff004	ldr	pc, [pc, #-4]	; 809c <_vectors_end>

```
0000809c <_vectors_end>:
```

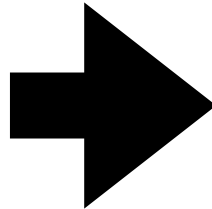
809c:	000080a4	.word	0x000080a4
80a0:	000080a8	.word	0x000080a8

Explicitly Embedded Absolute Addresses

```
.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
```



```
.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.
This works!!!

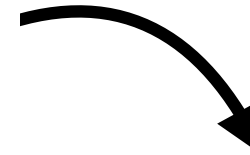
C Code

```
#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++;
```

0000807c <_vectors>:

807c:	e59ff018	ldr	pc, [pc, #24]	; 809c <abort_addr>
8080:	e59ff014	ldr	pc, [pc, #20]	; 809c <abort_addr>
8084:	e59ff010	ldr	pc, [pc, #16]	; 809c <abort_addr>
8088:	e59ff00c	ldr	pc, [pc, #12]	; 809c <abort_addr>
808c:	e59ff008	ldr	pc, [pc, #8]	; 809c <abort_addr>
8090:	e59ff004	ldr	pc, [pc, #4]	; 809c <abort_addr>
8094:	e59ff004	ldr	pc, [pc, #4]	; 80a0 <interrupt_addr>
8098:	e51ff004	ldr	pc, [pc, #-4]	; 809c <abort_addr>
809c:	000080a4	.word	0x000080a4	
80a0:	000080a8	.word	0x000080a8	



00000000 <_vectors>:

0000:	e59ff018	ldr	pc, [pc, #24]	; 809c <abort_addr>
0004:	e59ff014	ldr	pc, [pc, #20]	; 809c <abort_addr>
0008:	e59ff010	ldr	pc, [pc, #16]	; 809c <abort_addr>
000c:	e59ff00c	ldr	pc, [pc, #12]	; 809c <abort_addr>
0010:	e59ff008	ldr	pc, [pc, #8]	; 809c <abort_addr>
0014:	e59ff004	ldr	pc, [pc, #4]	; 809c <abort_addr>
0018:	e59ff004	ldr	pc, [pc, #4]	; 80a0 <interrupt_addr>
001c:	e51ff004	ldr	pc, [pc, #-4]	; 809c <abort_addr>
0020:	000080a4	.word	0x000080a4	
0024:	000080a8	.word	0x000080a8	

Interrupts Overview

Problem: responsive PS2 driver.

Answer: run interrupt code in response to events or inputs, CPU preempts execution, no blocking needed.

Requires setting up CPU to execute code, CPU provides some extra mechanisms and has different execution modes.

Hardware support for interrupts

Processor always executes in a particular "mode"

- Supervisor, interrupt, user, abort, ..
- Reset starts in supervisor mode (that's us!)
- Hardware monitors interrupt sources, when event occurs:

 Pause current mode, switch to interrupt mode

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 has instruction(s) to execute on interrupt

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.

Next Lecture

1. Set up the interrupt stack.
2. Install interrupt handler code.
3. Tell CPU when to trigger interrupts.
 - When PS/2 clock line has a falling edge
4. Enable interrupts!
5. Writing safe interrupt handlers.
 - How do you share state that can be modified at any time?