Interrupts and Concurrency

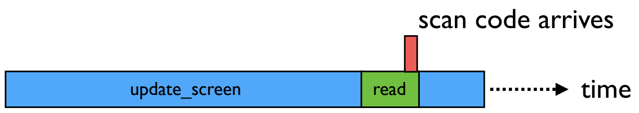
+

=

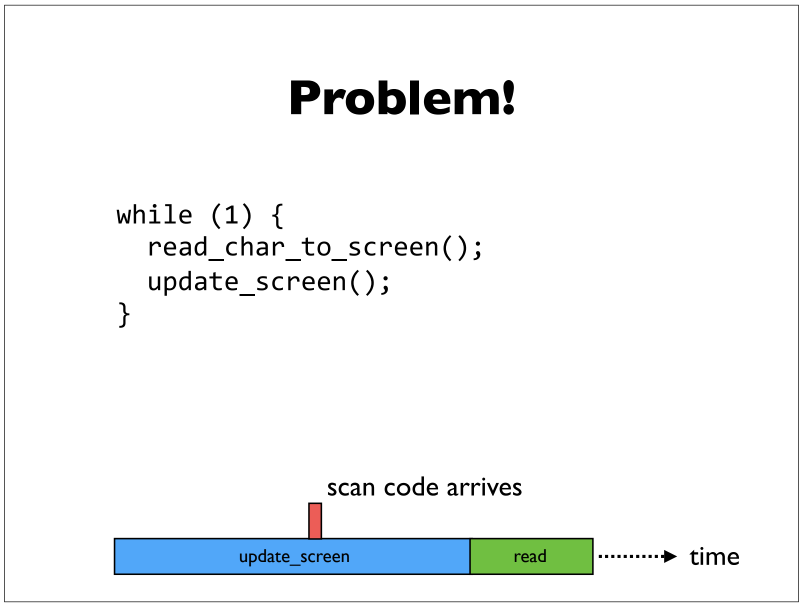
Now, we're cooking with gas

Synchronous I/O

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

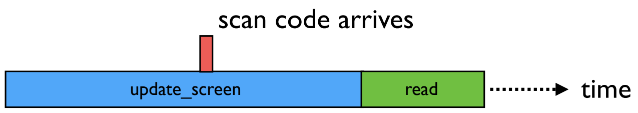


Currently, your console driver works like this. You read a character from the keyboard (which boils down to reading PS/2 scan codes from GPIO pins), then update the screen. The call to read a character spins, waiting for the clock line to change. Updating the screen takes a long time, much longer than a key press or the interval between key presses.

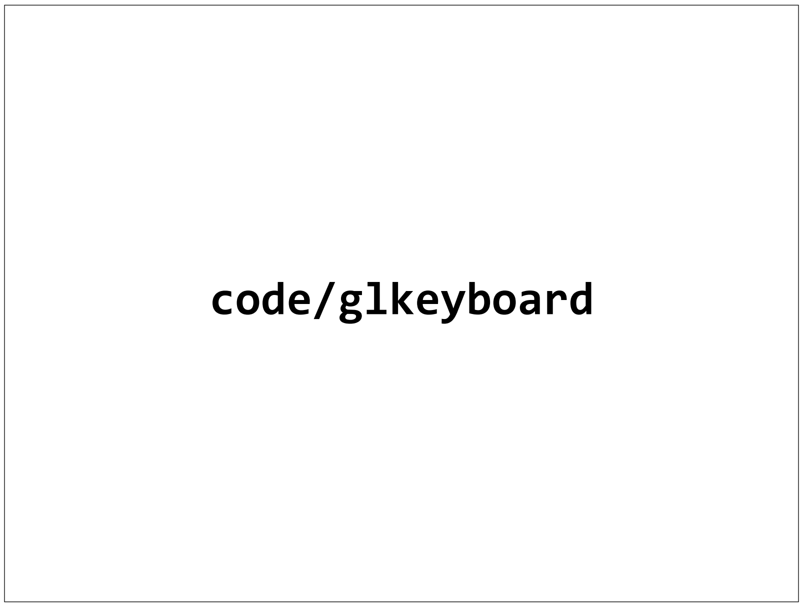


Problem!

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

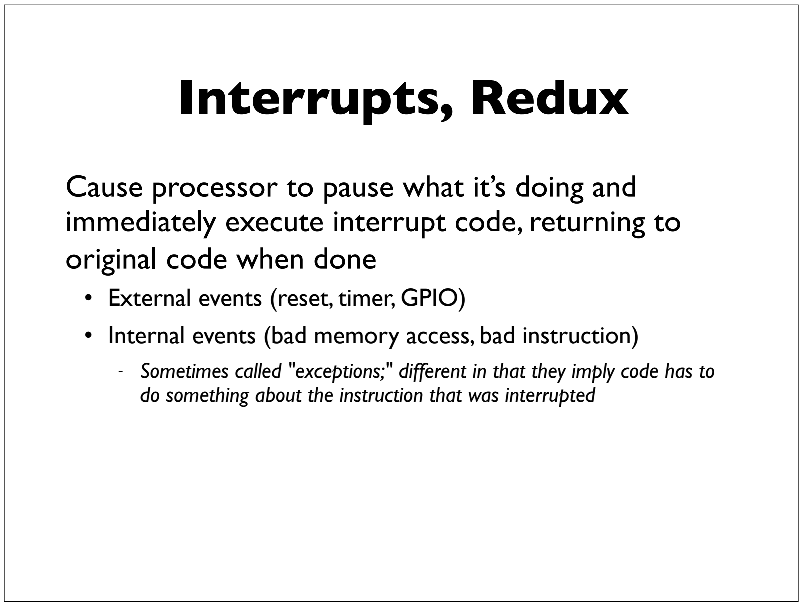


The problem with this approach is that if a scan code arrives while the screen is being updated, it'll be lost. You'll see key presses lost. This can be more than just characters: you can miss a shift-up and suddenly your keyboard is locked in shift mode!



`code/glkeyboard`

glkeyboard demonstrates this happening. Run it, and type fast at your ps/2 keyboard.

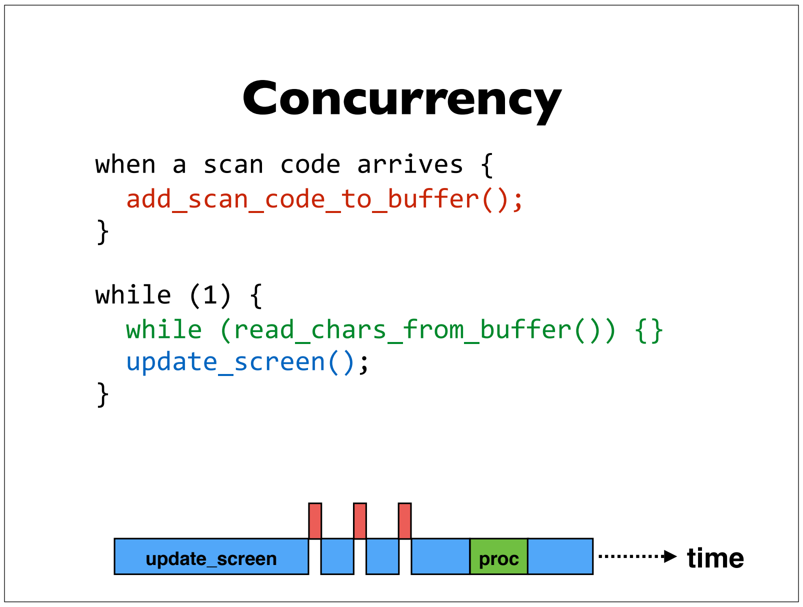


Interrupts, Redux

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

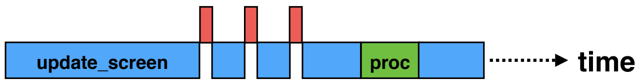
- External events (reset, timer, GPIO)
- Internal events (bad memory access, bad instruction)
  - *Sometimes called "exceptions;" different in that they imply code has to do something about the instruction that was interrupted*

Interrupts are the hardware mechanism to deal with this sort of challenge. Interrupts allow you to configure a processor such that when certain events happen, the processor stops its current instruction stream and jumps to other code, called *interrupt handlers.* When the interrupt handler completes, code returns to running the original instructions exactly where they left off.



Concurrency

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

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



Using interrupts, our code can look like this. Whenever the PS/2 clock line falls, it triggers an interrupt on the processor. The processor can then read in the PS/2 scan code by reading each data bit in the interrupt handler. When a scan code is received successfully, the handler appends it to a buffer. The main loop, rather than reading scan codes, just pulls scan codes out of this buffer. Now, even if update_screen takes a long time, the system won't lose scan codes, as long as the buffer is big enough.

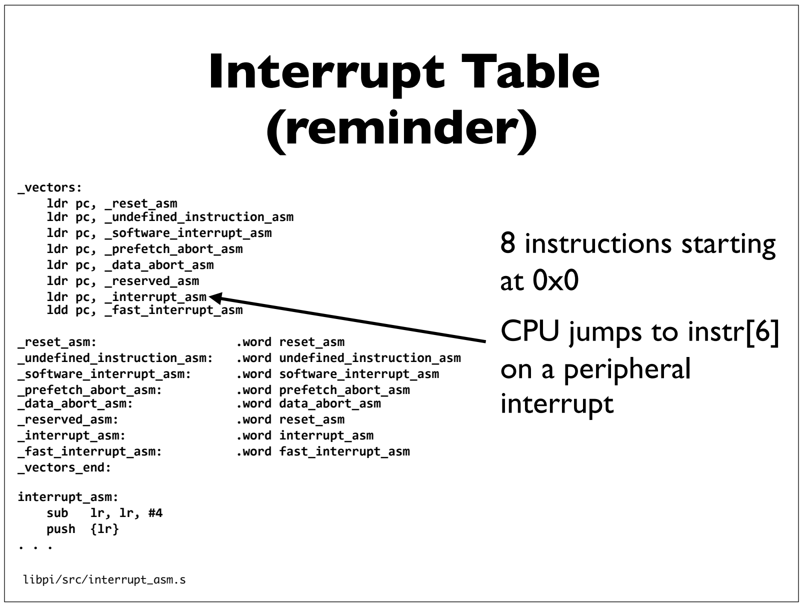
Last Lecture

8 different interrupts (we only care about one)

Processor specifies location (0x0) where it expects table of instructions, one per interrupt

- When an interrupt occurs, processor jumps to the corresponding instruction
- At that point, everything is software's responsibility

Interrupts are extremely valuable and seem simple, but getting them right requires using everything you've learned: assembly, linking, C, memory



Interrupt Table (reminder)

```
_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, _reserved_asm
    ldr pc, _interrupt_asm
    ldd pc, _fast_interrupt_asm

_reset_asm:                  .word reset_asm
_undefined_instruction_asm:  .word undefined_instruction_asm
_software_interrupt_asm:     .word software_interrupt_asm
_prefetch_abort_asm:         .word prefetch_abort_asm
_data_abort_asm:             .word data_abort_asm
_reserved_asm:               .word reset_asm
_interrupt_asm:              .word interrupt_asm
_fast_interrupt_asm:         .word fast_interrupt_asm
_vectors_end:

interrupt_asm:
    sub   lr, lr, #4
    push  {lr}
. . .
```

libpi/src/interrupt_asm.s

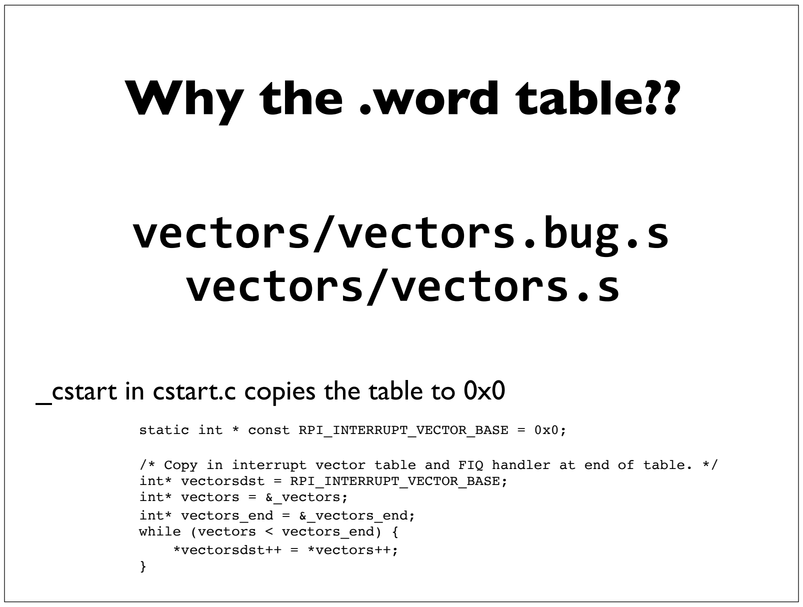
8 instructions starting at 0x0

CPU jumps to instr[6] on a peripheral interrupt

The ARMv6 architecture implements interrupts in this way. When an interrupt occurs, the processor jumps to an instruction in a table starting at address 0x0. The exact instruction depends on the type of interrupt, shown above. For peripherals, what we'll be caring about in this class, it jumps to the 7th element, or instruction[6]. Because the next instruction is what the processor should do when a fast interrupt occurs (something we won't cover, it's a way to write slightly lower overhead handlers), this single instruction needs to make the processor jump to the actual handler code. So this instruction loads the address of the interrupt handler code (written in assembly) into the PC:

```
ldr pc, _interrupt_asm
```

_interrupt_asm is defined to be a word of data (a variable) that stores the address of the interrupt_asm assembly function.



Why the .word table??

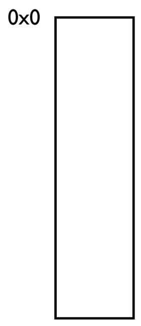
`vectors/vectors.bug.s`
`vectors/vectors.s`

_cstart in cstart.c copies the table to 0x0

```
static int * const RPI_INTERRUPT_VECTOR_BASE = 0x0;

/* Copy in interrupt vector table and FIQ handler at end of table. */
int* vectorsdst = RPI_INTERRUPT_VECTOR_BASE;
int* vectors = &_vectors;
int* vectors_end = &_vectors_end;
while (vectors < vectors_end) {
    *vectorsdst++ = *vectors++;
}
```

This seems a bit weird -- why is there a table of function addresses after the interrupt table, storing the addresses of the assembly functions to call? The reason has to do with the unique nature of interrupts, that they exist at address 0x0. Since our binary is loaded into the processor at address 0x8000, we can't send a binary to the boot loader that will put our interrupt table at 0x0. Instead, when our program boots, we need to copy our interrupt code to 0x0. If you look at _cstart in cstart.c, it does exactly this, copying everything between _vectors and _vectors_end to address 0x0.

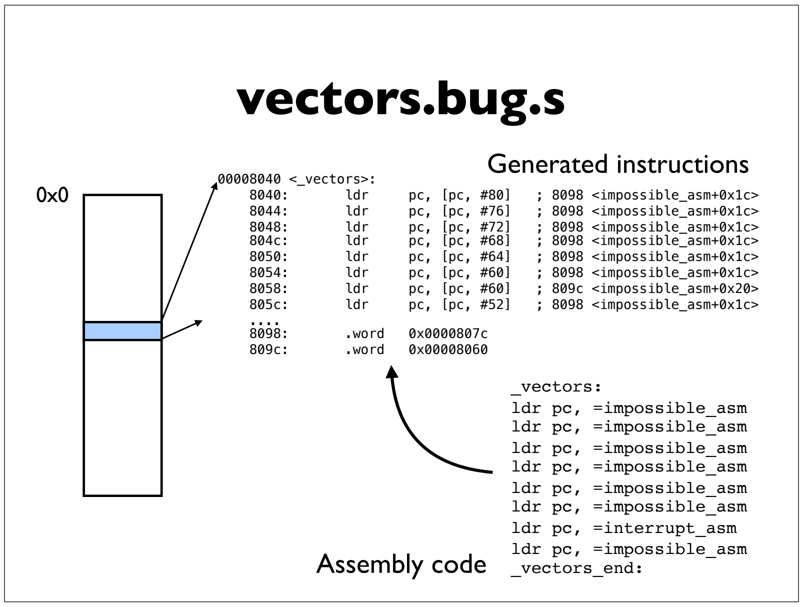
vectors.bug.s



Assembly code

```
_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
_vectors_end:
```

Let's walk through what happens if you don't have this table of function addresses after the interrupt instructions. Say, for example, that we wrote our interrupt table like this. These instructions just directly load the address of the function into the program counter.



vectors.bug.s

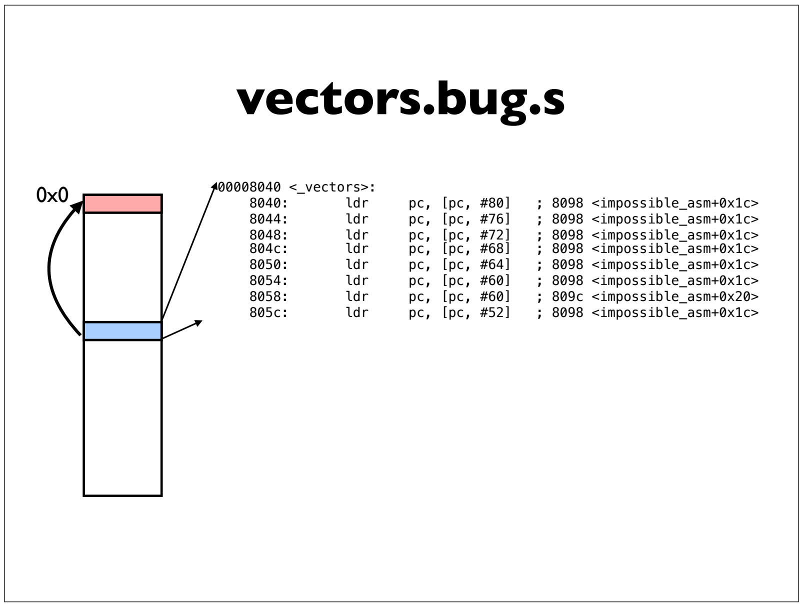
Generated instructions

```
00008040 <_vectors>:
    8040:       ldr     pc, [pc, #80]   ; 8098 <impossible_asm+0x1c>
    8044:       ldr     pc, [pc, #76]   ; 8098 <impossible_asm+0x1c>
    8048:       ldr     pc, [pc, #72]   ; 8098 <impossible_asm+0x1c>
    804c:       ldr     pc, [pc, #68]   ; 8098 <impossible_asm+0x1c>
    8050:       ldr     pc, [pc, #64]   ; 8098 <impossible_asm+0x1c>
    8054:       ldr     pc, [pc, #60]   ; 8098 <impossible_asm+0x1c>
    8058:       ldr     pc, [pc, #60]   ; 809c <impossible_asm+0x20>
    805c:       ldr     pc, [pc, #52]   ; 8098 <impossible_asm+0x1c>
    ....
    8098:       .word   0x0000807c
    809c:       .word   0x00008060
```

Assembly code

```
_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
_vectors_end:
```

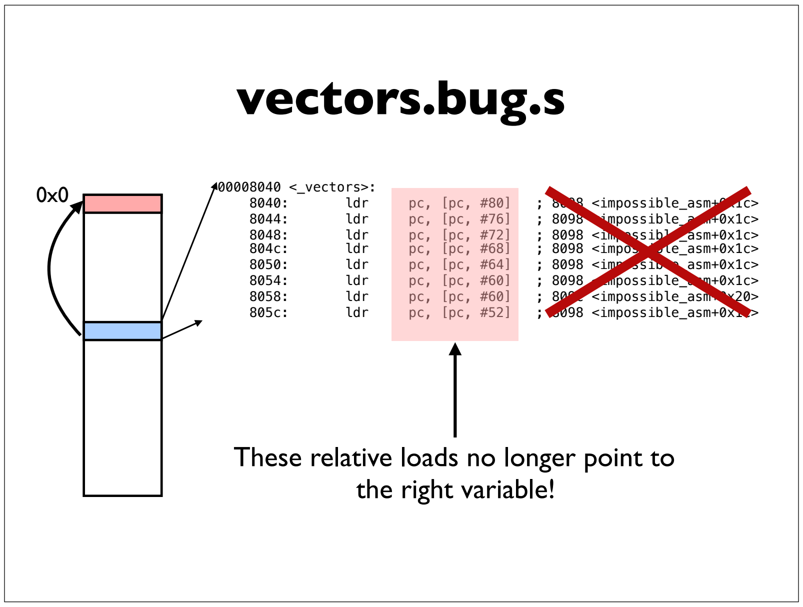
When the assembler assembles this code, it generates these instructions. These instructions have turned the immediate loads into loading a value relative to the PC. For the generated code, this is correct. It's stored the address of impossible_asm at 0x8098 and the address of interrupt_asm at 0x809c. But note that those values are kind of far away from the interrupt table itself: the assembler is allowed to put them wherever it wants.



vectors.bug.s

```
00008040 <_vectors>:
    8040:       ldr     pc, [pc, #80]   ; 8098 <impossible_asm+0x1c>
    8044:       ldr     pc, [pc, #76]   ; 8098 <impossible_asm+0x1c>
    8048:       ldr     pc, [pc, #72]   ; 8098 <impossible_asm+0x1c>
    804c:       ldr     pc, [pc, #68]   ; 8098 <impossible_asm+0x1c>
    8050:       ldr     pc, [pc, #64]   ; 8098 <impossible_asm+0x1c>
    8054:       ldr     pc, [pc, #60]   ; 8098 <impossible_asm+0x1c>
    8058:       ldr     pc, [pc, #60]   ; 809c <impossible_asm+0x20>
    805c:       ldr     pc, [pc, #52]   ; 8098 <impossible_asm+0x1c>
```

So what happens when we copy this code to 0x0? Now, those relative loads are no longer accessing the right addresses. pc + 80 at 0x0 won't load what's stored at 0x8098: it'll load what's stored at 0x58!

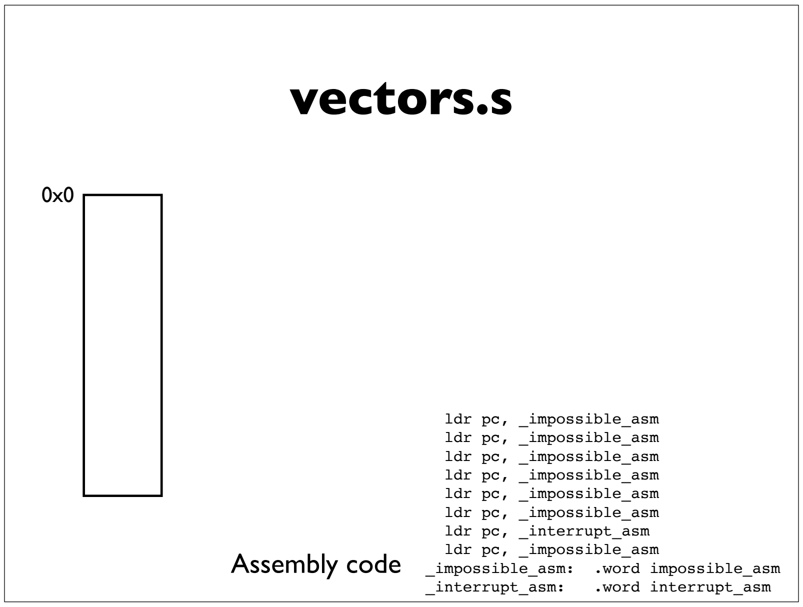


vectors.bug.s

```
00008040 <_vectors>:
    8040:       ldr     pc, [pc, #80]   ; 8098 <impossible_asm+0x1c>
    8044:       ldr     pc, [pc, #76]   ; 8098 <impossible_asm+0x1c>
    8048:       ldr     pc, [pc, #72]   ; 8098 <impossible_asm+0x1c>
    804c:       ldr     pc, [pc, #68]   ; 8098 <impossible_asm+0x1c>
    8050:       ldr     pc, [pc, #64]   ; 8098 <impossible_asm+0x1c>
    8054:       ldr     pc, [pc, #60]   ; 8098 <impossible_asm+0x1c>
    8058:       ldr     pc, [pc, #60]   ; 809c <impossible_asm+0x20>
    805c:       ldr     pc, [pc, #52]   ; 8098 <impossible_asm+0x1c>
```

These relative loads no longer point to the right variable!

So these relative loads won't load the right address to jump to: they'll load whatever is at 0x58 or 0x5c, which is likely 0x0. So you'll have an infinite loop!



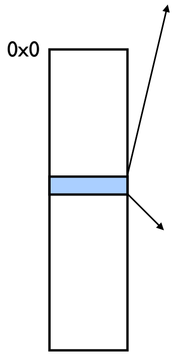
vectors.s

Assembly code

```
  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
```

So the right way to do this is to explicitly embed the function addresses with the table. That way, when you copy the table, you copy not only the assembly instructions, but also the data they depend on. If you do this, then the relative loads will work, because the functions addresses are stored (relatively) at the expected spot.

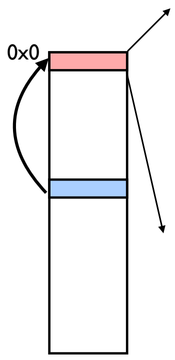
vectors.s



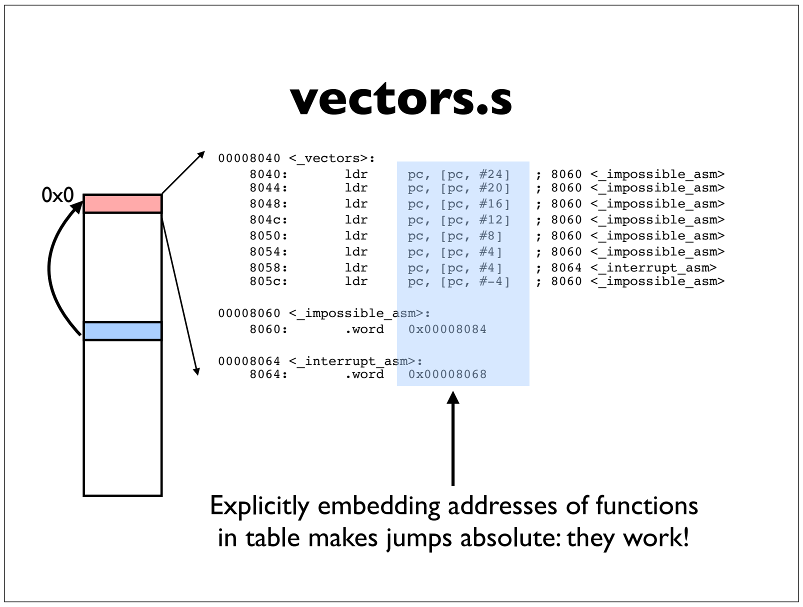
```
00008040 <_vectors>:
    8040:       ldr     pc, [pc, #24]   ; 8060 <_impossible_asm>
    8044:       ldr     pc, [pc, #20]   ; 8060 <_impossible_asm>
    8048:       ldr     pc, [pc, #16]   ; 8060 <_impossible_asm>
    804c:       ldr     pc, [pc, #12]   ; 8060 <_impossible_asm>
    8050:       ldr     pc, [pc, #8]    ; 8060 <_impossible_asm>
    8054:       ldr     pc, [pc, #4]    ; 8060 <_impossible_asm>
    8058:       ldr     pc, [pc, #4]    ; 8064 <_interrupt_asm>
    805c:       ldr     pc, [pc, #-4]   ; 8060 <_impossible_asm>

00008060 <_impossible_asm>:
    8060:       .word   0x00008084

00008064 <_interrupt_asm>:
    8064:       .word   0x00008068
```

```
  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
```

vectors.s



```
00008040 <_vectors>:
    8040:       ldr     pc, [pc, #24]   ; 8060 <_impossible_asm>
    8044:       ldr     pc, [pc, #20]   ; 8060 <_impossible_asm>
    8048:       ldr     pc, [pc, #16]   ; 8060 <_impossible_asm>
    804c:       ldr     pc, [pc, #12]   ; 8060 <_impossible_asm>
    8050:       ldr     pc, [pc, #8]    ; 8060 <_impossible_asm>
    8054:       ldr     pc, [pc, #4]    ; 8060 <_impossible_asm>
    8058:       ldr     pc, [pc, #4]    ; 8064 <_interrupt_asm>
    805c:       ldr     pc, [pc, #-4]   ; 8060 <_impossible_asm>

00008060 <_impossible_asm>:
    8060:       .word   0x00008084

00008064 <_interrupt_asm>:
    8064:       .word   0x00008068
```



vectors.s

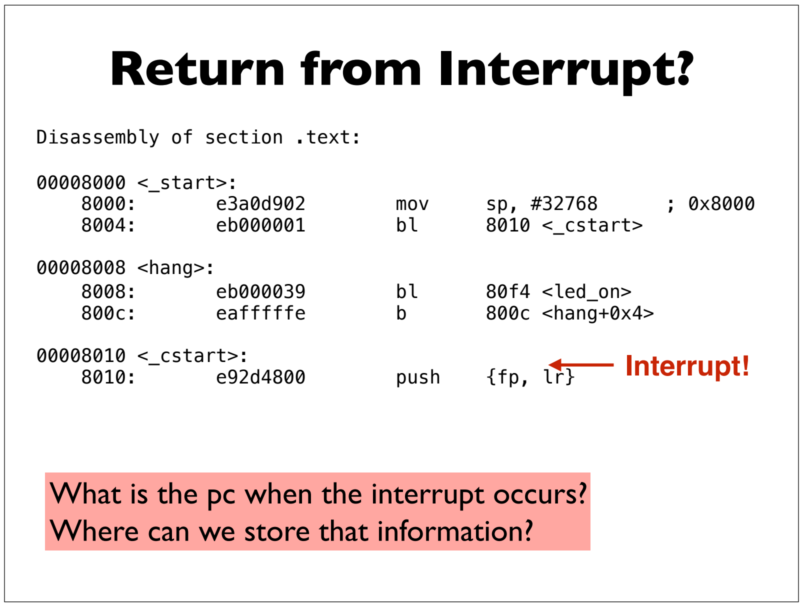
```
00008040 <_vectors>:
    8040:       ldr     pc, [pc, #24]   ; 8060 <_impossible_asm>
    8044:       ldr     pc, [pc, #20]   ; 8060 <_impossible_asm>
    8048:       ldr     pc, [pc, #16]   ; 8060 <_impossible_asm>
    804c:       ldr     pc, [pc, #12]   ; 8060 <_impossible_asm>
    8050:       ldr     pc, [pc, #8]    ; 8060 <_impossible_asm>
    8054:       ldr     pc, [pc, #4]    ; 8060 <_impossible_asm>
    8058:       ldr     pc, [pc, #4]    ; 8064 <_interrupt_asm>
    805c:       ldr     pc, [pc, #-4]   ; 8060 <_impossible_asm>

00008060 <_impossible_asm>:
    8060:       .word   0x00008084

00008064 <_interrupt_asm>:
    8064:       .word   0x00008068
```

Explicitly embedding addresses of functions in table makes jumps absolute: they work!

So this is why we have to explicitly embed the variables containing the desired function addresses. Writing this code, and understanding it, requires understanding not only ARM assembly, but also the linker and how one can define symbols and variables such that the table is copied as well.

Interrupt Handler

So now that we've revisited how you can install a correct interrupt table, let's look at the interrupt handlers themselves.

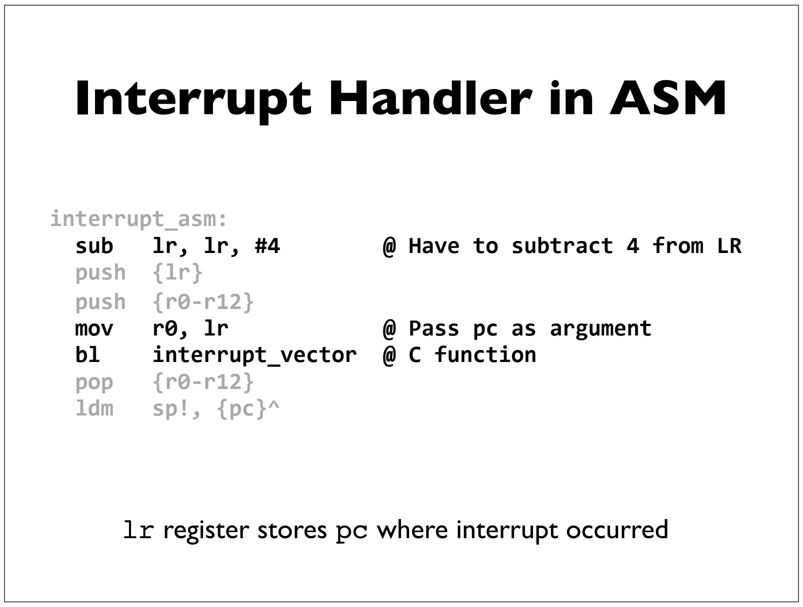


Return from Interrupt?

```
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}
```

What is the pc when the interrupt occurs?
Where can we store that information?

So let's say we're executing, and suddenly there's an interrupt right before the first instruction of _cstart. Normally this wouldn't happen (we haven't installed the interrupt table), but let's suppose we did that earlier, and use _cstart because it's such as simple function you always use. _cstart needs to push the link register and frame pointer. The link register contains the address 0x8008, which is where your code will return to after _cstart returns.



Interrupt Handler in ASM

```
interrupt_asm:
  sub   lr, lr, #4       @ Have to subtract 4 from LR
  push  {lr}
  push  {r0-r12}
  mov   r0, lr           @ Pass pc as argument
  bl    interrupt_vector  @ C function
  pop   {r0-r12}
  ldm   sp!, {pc}^
```

`lr` register stores `pc` where interrupt occurred

Our interrupt handler code looks like this. You can ignore the greyed instructions for now. The first thing the handler does is subtract 4 from the LR. The link register normally tells you the instruction *after* a branch. But in the case of an interrupt, we don't want to return to the instruction *after* the one interrupted, but instead the interrupted instruction itself. So we need to subtract 4.

Hold on -- the LR now contains the address of the instruction at which the interrupt occurred. But that instruction was going to operate on the LR. Does this mean that, when we return to _cstart, our link register is corrupted, and so when _cstart returns, it will return to itself?

Processor Modes

User - unprivileged mode

IRQ - interrupt mode

FIQ - fast interrupt mode

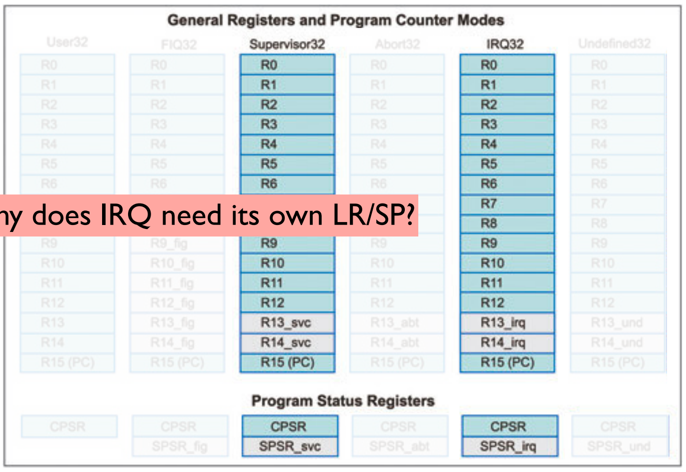
Supervisor - privileged mode, entered on reset

Abort - memory access violation

Undefined - undefined instruction

System - privileged mode that shares user regs

Shared / Private Registers



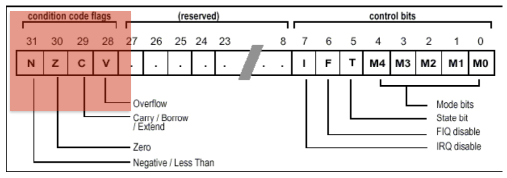
Why does IRQ need its own LR/SP?

Why does an interrupt need a separate context that has its own link register and stack pointer?

In the case of the link register, imagine what would happen if, right after a bl instruction (which puts the old PC in the LR), an interrupt occurs. The value that was in the LR before the instruction is needed: it's the return address the function should go to when it completes. But the interrupt handler needs to also know the address it should return to when the interrupt handler returns. The function hasn't had a chance to save the LR anywhere. If an interrupt overwrites the LR, then the function's return address will be lost. Therefore, the processor has a separate LR for when in interrupt context. When the interrupt handler tells the processor to exit out of IRQ mode (when it returns), then the original function gets to use its own (supervisor mode) LR, unmodified.

So why not have a *full* set of interrupt-private registers? Complexity and speed. Having more registers takes up space and could make accessing them slower. Some processors do something like this. The assumption is that interrupts often do very little, so don't need to save all of the registers. It could therefore be faster (in terms of overall speed) to have only one set of registers and required the interrupt handler to save them. Our interrupt handlers save all of the registers for simplicity and safety, so you're not tracking down weird bugs due to register corruption. But if you're optimizing the lowest levels of the system and know exactly what the interrupt handler does, you can often get away with doing much less.

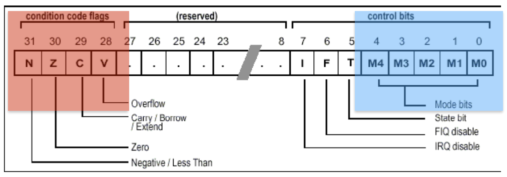
CPSR



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

```
msr psr, Rm      <- Store Rm into psr
mrs Rd, psr      <- Load Rd with psr

msr cpsr_c, r0   <- Store CPSR with r0
mrs r0, cpsr_c   <- Load r0 with CPSR
```

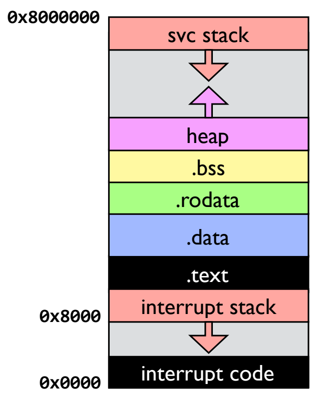
CPSR



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

```
msr psr, Rm      <- Store Rd into psr
mrs Rd, psr      <- Load Rd with psr

msr cpsr_c, r0   <- Store CPSR with r0
mrs r0, cpsr_c   <- Load r0 with CPSR
```

Set up Interrupt Stack



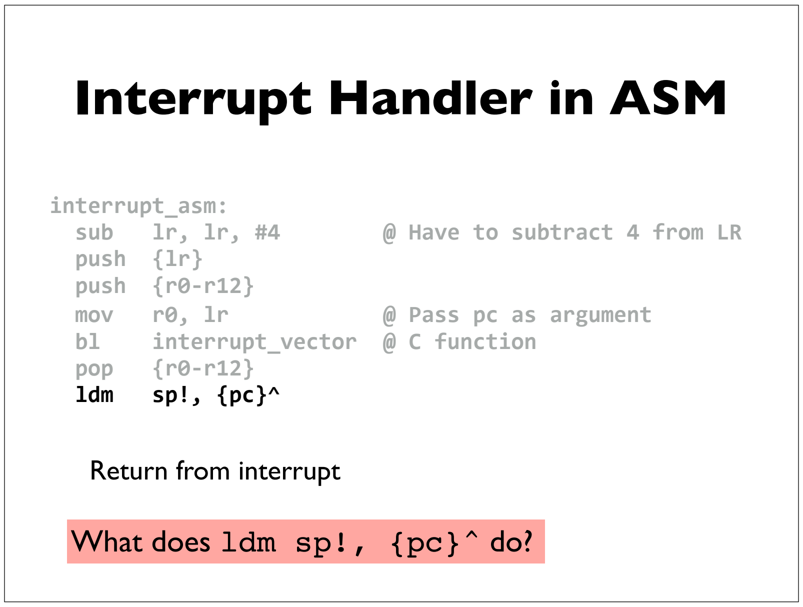
start.s

```
_start:
    mov r0, #0xD2  // Interrupts
    msr cpsr_c, r0
    mov sp, #0x8000
    mov r0, #0xD1  // Fast interrupts
    msr cpsr_c, r0
    mov sp, #0x4000
    mov fp, #0     // FIQ has fp
    mov r0, #0xD3  // Supervisor
    msr cpsr_c, r0
    mov sp, #0x8000000
    mov fp, #0
    bl _cstart
```

Interrupt Handler in ASM

```
interrupt_asm:
  sub   lr, lr, #4        @ Have to subtract 4 from LR
  push  {lr}
  push  {r0-r12}
  mov   r0, lr            @ Pass pc as argument
  bl    interrupt_vector  @ C function
  pop   {r0-r12}
  ldm   sp!, {pc}^
```

Save all registers on the stack: don't want the code that was running to mysteriously have different values in registers!



Interrupt Handler in ASM

```
interrupt_asm:
  sub   lr, lr, #4        @ Have to subtract 4 from LR
  push  {lr}
  push  {r0-r12}
  mov   r0, lr            @ Pass pc as argument
  bl    interrupt_vector  @ C function
  pop   {r0-r12}
  ldm   sp!, {pc}^
```

Return from interrupt

What does `ldm sp!, {pc}^` do?

When running inside an interrupt handler you are using an alternative set of registers in order to preserve the "user" register values that were in place when the interrupt was triggered.

IIRC putting the caret ^ at the end of stm/ldm instructions copies the SPSR to the CPSR, so it returns you from interrupt context.

Enabling Interrupts

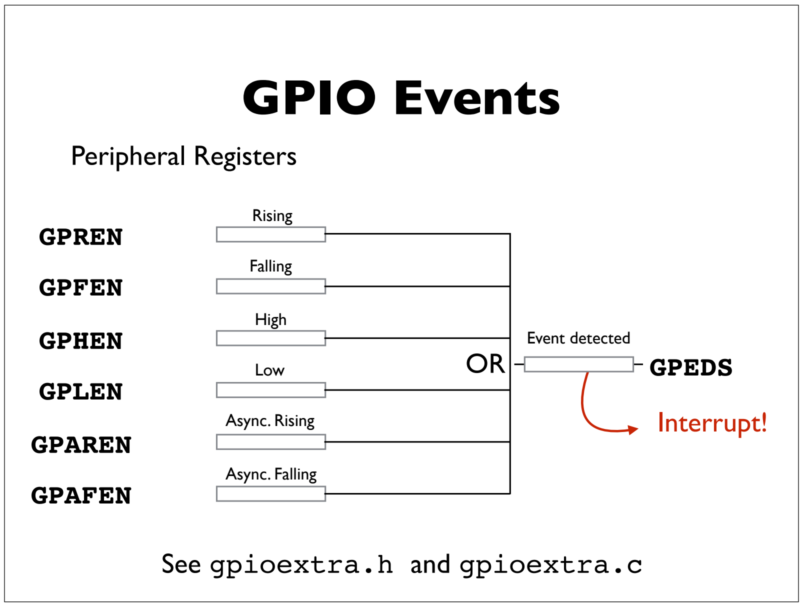
Three Layers

1. Enable/disable a specific interrupt source
   - For example, when we detect a falling clock edge on GPIO_PIN23 (PS/2 CLK)
2. Enable/disable type of interrupts
   - E.g., GPIO interrupts
3. Global interrupt enable/disable

Interrupt fires if and only all three are enabled

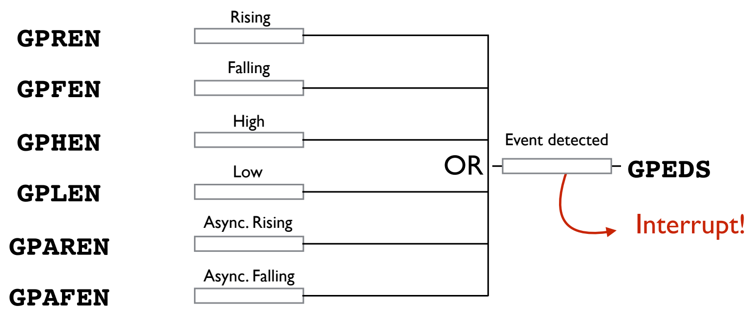
Forgetting to enable one is a common bug

`armtimer/blink.c`



GPIO Events

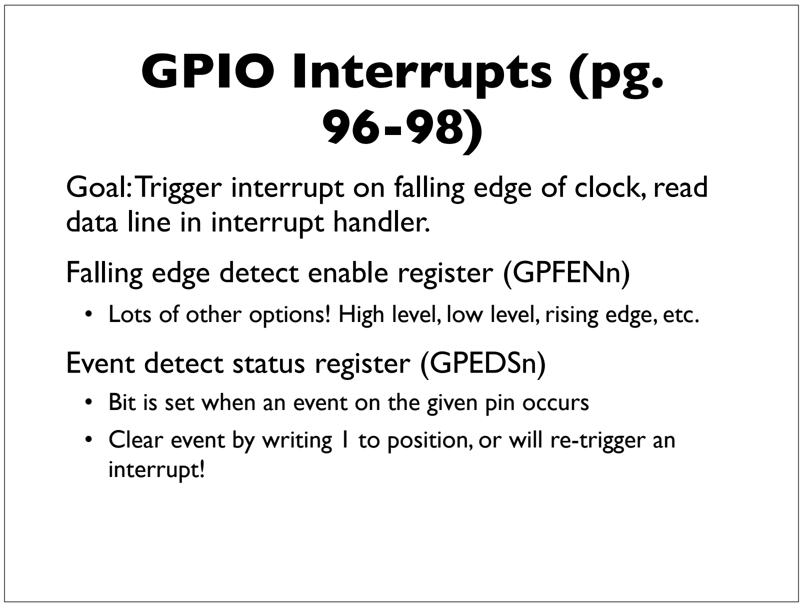
Peripheral Registers



See `gpioextra.h` and `gpioextra.c`

GPFEN0 and GPFEN1 - GPIO pin falling edge enable

GPEDS0 and GPEDS1 - pin event detect status



GPIO Interrupts (pg. 96-98)

Goal: Trigger interrupt on falling edge of clock, read data line in interrupt handler.

Falling edge detect enable register (GPFENn)

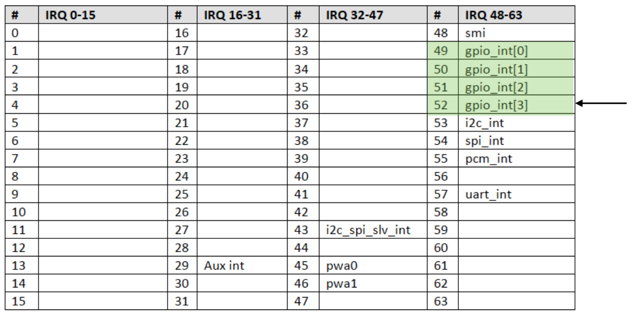
- Lots of other options! High level, low level, rising edge, etc.

Event detect status register (GPEDSn)

- Bit is set when an event on the given pin occurs
- Clear event by writing 1 to position, or will re-trigger an interrupt!

The event detect status registers are used to record level and edge events on the GPIO pins. The relevant bit in the event detect status registers is set whenever: 1) an edge is detected that matches the type of edge programmed in the rising/falling edge detect enable registers, or 2) a level is detected that matches the type of level programmed in the high/low level detect enable registers. The bit is cleared by writing a “1” to the relevant bit.
The interrupt controller can be programmed to interrupt the processor when any of the status bits are set. The GPIO peripheral has three dedicated interrupt lines. Each GPIO bank can generate an independent interrupt. The third line generates a single interrupt whenever any bit is set.

BCM2835, Sec 7.5

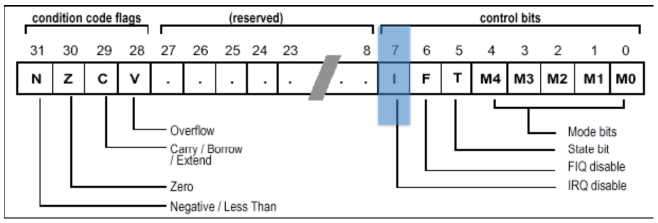


#	IRQ 0-15	#	IRQ 16-31	#	IRQ 32-47	#	IRQ 48-63
0		16		32		48	smi
1		17		33		49	gpio_int[0]
2		18		34		50	gpio_int[1]
3		19		35		51	gpio_int[2]
4		20		36		52	gpio_int[3]
5		21		37		53	i2c_int
6		22		38		54	spi_int
7		23		39		55	pcm_int
8		24		40		56	
9		25		41		57	uart_int
10		26		42		58	
11		27		43	i2c_spi_slv_int	59	
12		28		44		60	
13		29	Aux int	45	pwa0	61	
14		30		46	pwa1	62	
15		31		47		63	

From the "internet"

```
GPIO pin:              4    17   30   31   47
 gpio_irq[0] (49)      Y    Y    Y    Y    N
 gpio_irq[1] (50)      N    N    Y    Y    N
 gpio_irq[2] (51)      N    N    N    N    Y
 gpio_irq[3] (52)      Y    Y    Y    Y    Y
```

Enabling Global Interrupts



```
.global interrupts_global_enable
interrupts_global_enable:
    mrs r0,cpsr
    bic r0,r0,#0x80
    // I=0 enables interrupts
    msr cpsr_c,r0
    bx lr
```

```
.global interrupts_global_disable
interrupts_global_disable:
    mrs r0,cpsr
    orr r0,r0,#0x80
    // I=1 disables interrupts
    msr cpsr_c,r0
    bx lr
```

code/button-interrupts

We’re done!

We now can write correct and safe interrupt code

- Assembly to save all registers
- Call into C code
- Assembly to restore registers, return to interrupted code

We can install the interrupt code table to 0x0

- Embed addresses of assembly routines so jumps are absolute
- Copy interrupt table to 0x0 in cstart

Enable and disable interrupts

- Specific interrupts, per-peripheral interrupts, global interrupts

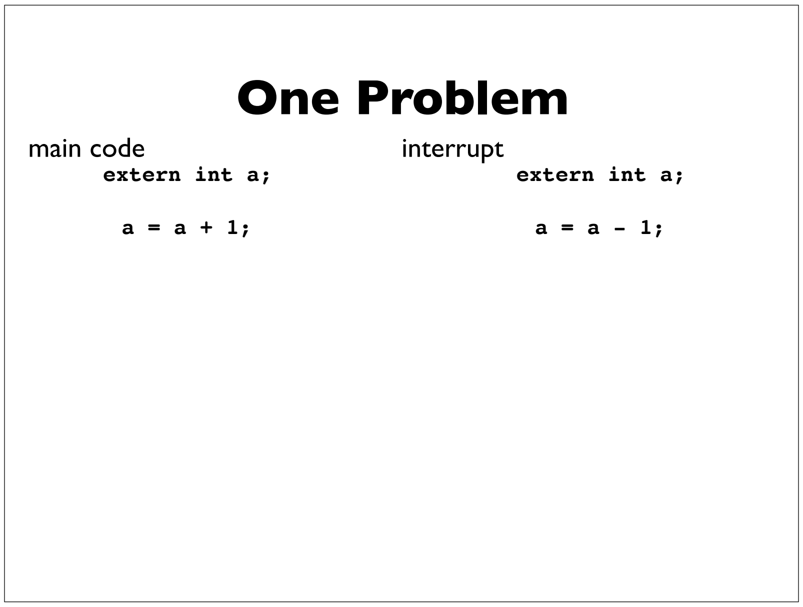
Not Quite

An interrupt can fire at any time

- Interrupt handler may put a PS/2 scan code in a buffer
- Could do so in the middle of when main() code is trying to pull a scan code out of the buffer
- Need to make sure the interrupt doesn't corrupt the buffer

<u>Need to write code that can be safely interrupted</u>

code/race



One Problem

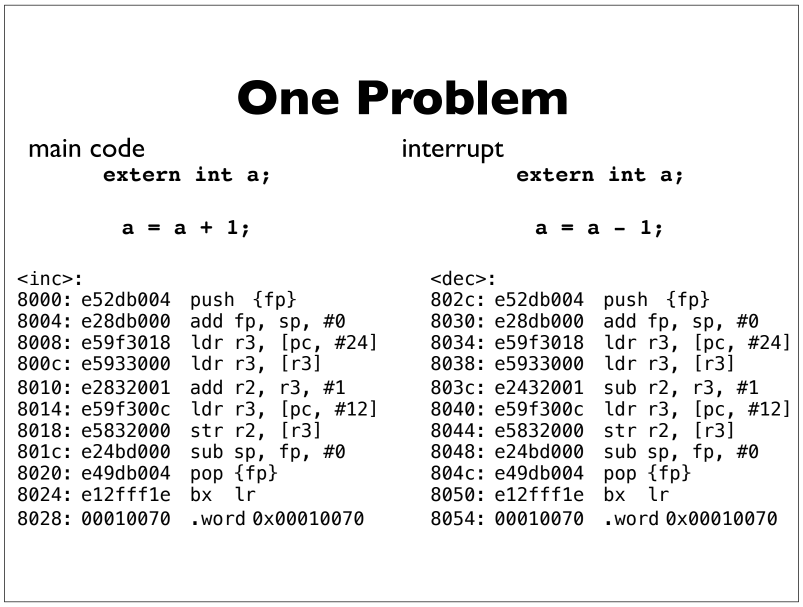
main code

```
extern int a;

  a = a + 1;
```

interrupt

```
extern int a;

  a = a - 1;
```

atomic



One Problem

main code

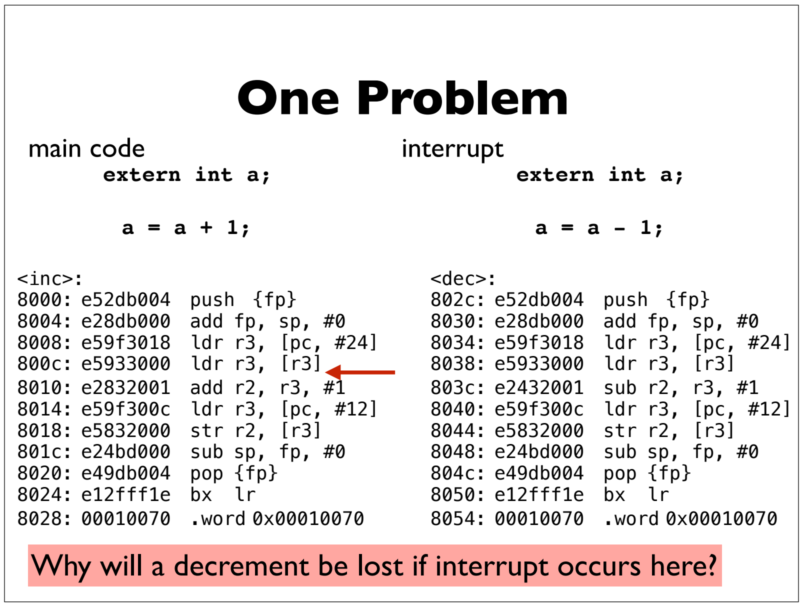
```
      extern int a;

        a = a + 1;
```

interrupt

```
      extern int a;

        a = a - 1;
```

```
<inc>:                                      <dec>:
8000: e52db004  push  {fp}                  802c: e52db004  push  {fp}
8004: e28db000  add fp, sp, #0              8030: e28db000  add fp, sp, #0
8008: e59f3018  ldr r3, [pc, #24]           8034: e59f3018  ldr r3, [pc, #24]
800c: e5933000  ldr r3, [r3]                8038: e5933000  ldr r3, [r3]
8010: e2832001  add r2, r3, #1              803c: e2432001  sub r2, r3, #1
8014: e59f300c  ldr r3, [pc, #12]           8040: e59f300c  ldr r3, [pc, #12]
8018: e5832000  str r2, [r3]                8044: e5832000  str r2, [r3]
801c: e24bd000  sub sp, fp, #0              8048: e24bd000  sub sp, fp, #0
8020: e49db004  pop {fp}                    804c: e49db004  pop {fp}
8024: e12fff1e  bx  lr                      8050: e12fff1e  bx  lr
8028: 00010070  .word 0x00010070            8054: 00010070  .word 0x00010070
```

atomic



One Problem

main code

```
extern int a;

a = a + 1;
```

interrupt

```
extern int a;

a = a - 1;
```

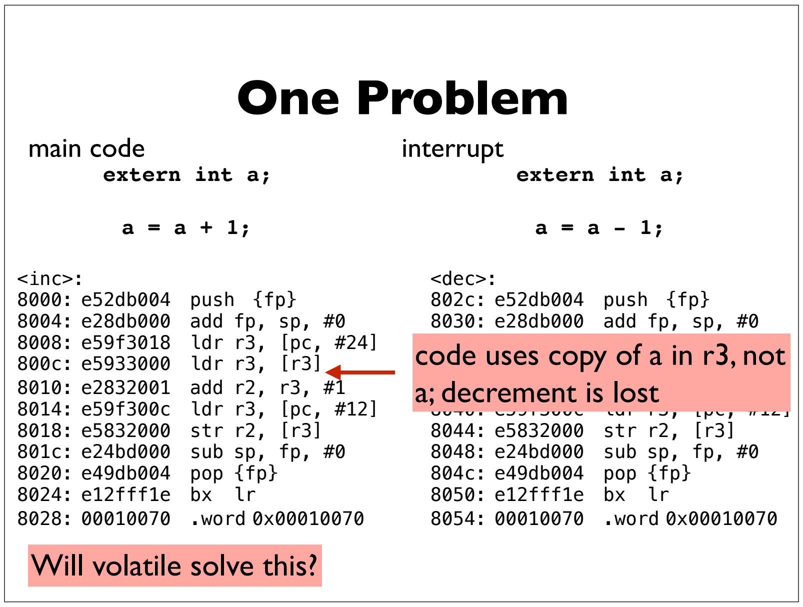
```
<inc>:
8000: e52db004  push  {fp}
8004: e28db000  add fp, sp, #0
8008: e59f3018  ldr r3, [pc, #24]
800c: e5933000  ldr r3, [r3]  <-
8010: e2832001  add r2, r3, #1
8014: e59f300c  ldr r3, [pc, #12]
8018: e5832000  str r2, [r3]
801c: e24bd000  sub sp, fp, #0
8020: e49db004  pop {fp}
8024: e12fff1e  bx  lr
8028: 00010070  .word 0x00010070
```

```
<dec>:
802c: e52db004  push  {fp}
8030: e28db000  add fp, sp, #0
8034: e59f3018  ldr r3, [pc, #24]
8038: e5933000  ldr r3, [r3]
803c: e2432001  sub r2, r3, #1
8040: e59f300c  ldr r3, [pc, #12]
8044: e5832000  str r2, [r3]
8048: e24bd000  sub sp, fp, #0
804c: e49db004  pop {fp}
8050: e12fff1e  bx  lr
8054: 00010070  .word 0x00010070
```

Why will a decrement be lost if interrupt occurs here?

The arrow shows the point at which a has been loaded into r3, but before it has been incremented.

The decrement will be lost because r3 stores the value of a before the decrement. E.g., suppose a is 5. The main loop should increment it to 6. So it loads a (5) into r3. Then, the interrupt fires and decrements a to 4. The main code results, increments r3 to 6, and then stores that result back to a. The decrement was lost.



One Problem

main code

```
extern int a;

  a = a + 1;

<inc>:
8000: e52db004  push  {fp}
8004: e28db000  add fp, sp, #0
8008: e59f3018  ldr r3, [pc, #24]
800c: e5933000  ldr r3, [r3]
8010: e2832001  add r2, r3, #1
8014: e59f300c  ldr r3, [pc, #12]
8018: e5832000  str r2, [r3]
801c: e24bd000  sub sp, fp, #0
8020: e49db004  pop {fp}
8024: e12fff1e  bx  lr
8028: 00010070  .word 0x00010070
```

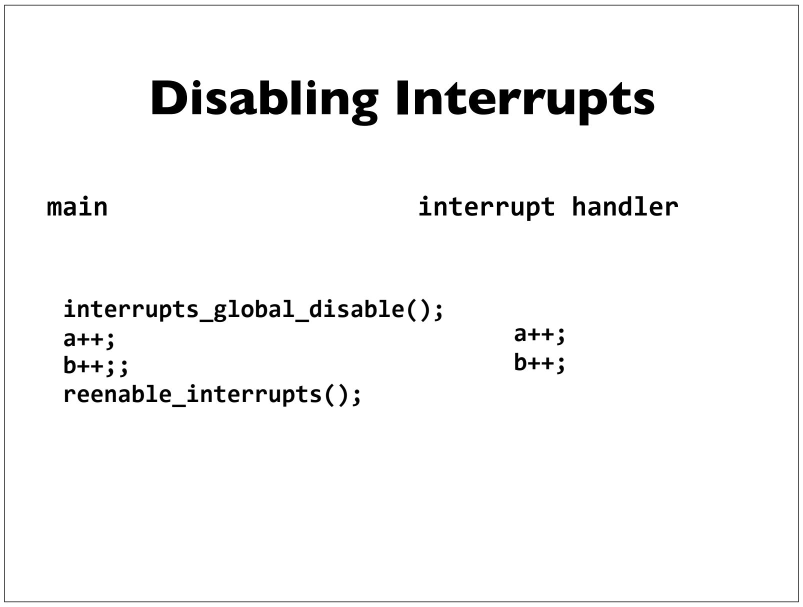
interrupt

```
extern int a;

  a = a - 1;

<dec>:
802c: e52db004  push  {fp}
8030: e28db000  add fp, sp, #0
8044: e5832000  str r2, [r3]
8048: e24bd000  sub sp, fp, #0
804c: e49db004  pop {fp}
8050: e12fff1e  bx  lr
8054: 00010070  .word 0x00010070
```

code uses copy of a in r3, not a; decrement is lost

Will volatile solve this?

Volatile won't solve this! The issue is that the code needs to atomically (in a way that cannot be divided, either executes fully or not at all) read, modify, and write the variable in memory.



Disabling Interrupts

```
main                          interrupt handler


 interrupts_global_disable();
 a++;                               a++;
 b++;;                              b++;
 reenable_interrupts();
```

To solve this problem, we can just disable interrupts. Now, we're sure that the interrupt handler won't run while the main loop is incrementing a and b, and so won't see partial results.

Preemption and Safety

Very hard, lots of bugs.

You'll learn more in CS110/CS140.

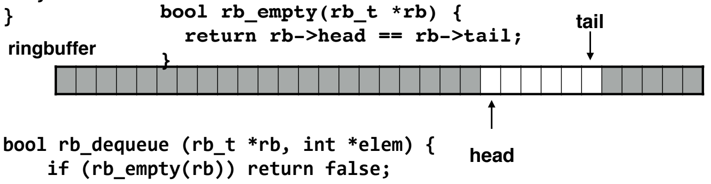
Two simple answers

1. Use simple, safe data structures
   - write once, but not always possible
2. Otherwise, temporarily disable interrupts
   - always works, but easy to forget

Safe Ring Buffer

```
int rb_enqueue(rb_t *rb, int elem) {
    if (rb_full(rb)) {
        return 1;
    } else {
      rb->entries[rb->tail] = elem;
      rb->tail = (rb->tail + 1) % LENGTH; // only writes tail
      return 0;
   }
}
```

```
bool rb_empty(rb_t *rb) {
  return rb->head == rb->tail;
}
```



```
bool rb_dequeue (rb_t *rb, int *elem) {
    if (rb_empty(rb)) return false;
    *elem = rb->entries[rb->head];
    rb->head = (rb->head + 1) % LENGTH; // only writes head
    return true;
}
```

Ringbuffer (rb)

```
void interrupt_handler(void) {
  read_data_bit();
  if (scancode_complete) {
    rb_enqueue(rb, scancode);
  }
}


keyboard_read_scancode(void) {
  while (rb empty(rb) {}
  if( !rb_empty(rb) )
    rb_dequeue(rb, &scancode);
}
```

This Lecture

Writing the code that runs in interrupts

- Assembly code needed to change to processor models and special registers
- Interrupt table copied to `0x0` in `cstart.c`

Setting up the CPU to issue interrupts

- 3 levels: cause, type, global

Writing code that can be safely interrupted

- Race conditions though interrupt-safe ring buffer

Summary

Interrupts allow external events to preempt what's executing and run code immediately

- Needed for responsiveness, e.g., do not missing PS/2 scan codes from keyboard when drawing
- Without interrupts, most computers do nothing: they deliver keystrokes, network packets, disk reads, timers, etc.

Simple goal, but working correctly is very tricky!

- Deals with many of the hardest issues in systems

Assignment 7: update keyboard to use interrupts