

# **ECE 531/598 – Advanced Operating Systems Lecture 8**

Vince Weaver

`https://web.eece.maine.edu/~vweaver`

`vincent.weaver@maine.edu`

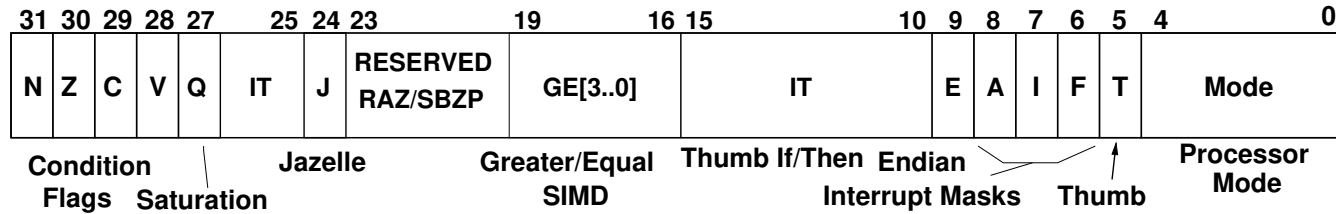
21 September 2023

# Announcements

- Homework #3 Due Friday  
Notes on the USB-serial situation  
Two drivers for MacOS/Windows  
Modern OS device-drivers, code signing, etc, is complicated
- Homework #4 will be posted, still working on Pi4 support



# (Review) ARM CPSR Register



- Current Program Status Register



# (Review) ARM Interrupt Handling

- ARM core saves CPSR to the proper SPSR
- ARM core saves PC to the banked LR (possibly with an offset)
- ARM core sets CPSR to exception mode (disables interrupts)
- ARM core jumps to appropriate offset in vector table



# Vector Table

Type	Type	Offset	LR	Priority
Reset	SVC	0x0	–	1
Undefined Instruction	UND	0x04	lr-4/2	6
Software Interrupt	SVC	0x08	lr	6
Prefetch Abort	ABT	0x0c	lr-4	5
Data Abort	ABT	0x10	lr-8	2
UNUSED	–	0x14	–	–
IRQ	IRQ	0x18	lr-4	4
FIQ	FIQ	0x1c	lr-4	3



- See ARM ARM ARMv7 documentation for details
- NOTE: contains a 4-byte instruction, not an address
- Location defaults to 0x000000  
if SCTL.V is 1 “high-vector” 0xffff0000
- If security mode implemented more complex, separate vectors for secure/nonsecure, and on nonsecure the SCTL.V lets you set it anywhere via VBAR
- Interrupts: IRQ = general purpose hardware,  
FIQ = fast interrupt for really fast response (only 1),  
SWI = syscalls, talk to OS
- FIQ mode auto-saves r8-r12.



# Complications

- What about thumb or endian mode when call into interrupt? Depends on flags in SCTRLR register
- Stack pointer changes when handle interrupt (why?)
- Need to set that up in advance, before interrupts enabled
- Why does kernel have own stack pointer? Why not use the user stack? Does the user stack pointer always have to be valid?



# Ways to return from IRQ

- Regular function return not enough, need to change mode and adjust LR
- `subs pc,lr,#4`  
Sneakily branches and gets the right status register (special case when S in SUBS and PC is destination)
- `sub r14,r14,#4`  
...  
`movs pc,lr (or rfe)`
- Another stores lr and other things to stack, then restores





```
sub lr,lr,#4  
stmbd sp!,{r0-r12,lr}
```

...

```
ldmfd sp!,{r0-r12,pc}^
```

The caret means to load cpsr from spsr

Exclamation point means to update sp after popping.



# IRQ Handlers in C

In gcc for ARM, you can specify the interrupt type with an attribute. Automatically restores to right address.

```
void function () __attribute__((interrupt ("IRQ")));

/* Can be IRQ, FIQ, SWI, ABORT and UNDEF */

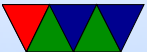
void __attribute__((interrupt("UNDEF"))) undefined_instruction_vector(void) {

    while(1) {
        /* Do Nothing */
    }
}
```



# Getting Interrupt to Happen

- Initialize (set up vectors and stacks)
- Enable Interrupt at Pi Level
- Enable Interrupt at Device Level
- Enable Global interrupts at ARM Level



# Raspberry Pi Interrupts

- See Section 7 of BCM2835 doc (though it's not well written)
- Up to 64 possible, but only subset available to ARM chip (rest belong to GPU)
- MMIO Registers used to configure:
  - Basic pending: 32-bit field with most common IRQ sources
  - Full pending: two 32-bit registers a bit for each IRQ source and whether triggered



- FIQ register: can pick which one is FIQ
- Enable register: to set which interrupts are enabled
- Disable register
- You also have to enable interrupts on the device too
- On Pi4 need to enable gic-400 interrupt controller too



# Initializing

- How do we get the vectors to address 0x0?  
Copy it there after the fact. Hard part is if we want the routines to be C code.
- Clever, have the reset vector point to start of code, so you can have the reset vector of beginning of code and it will jump to the right location.
- `ldr` does a PC-relative load, so as long as we copy the vectors at the same offset will work
- Leave at entry point, and first one is reset, so at boot



# we jump to reset

```
_start:
    ldr pc, reset_addr
    ldr pc, undefined_addr
    ldr pc, software_interrupt_addr
    ldr pc, prefetch_abort_addr
    ldr pc, data_abort_addr
    ldr pc, unused_addr
    ldr pc, interrupt_addr
    ldr pc, fast_interrupt_addr
reset_addr:                .word    reset
undefined_addr:           .word    undefined_instruction
software_interrupt_addr:  .word    software_interrupt
prefetch_abort_addr:     .word    prefetch_abort
data_abort_addr:          .word    data_abort
unused_addr:              .word    reset
interrupt_addr:           .word    interrupt
fast_interrupt_addr:     .word    fast_interrupt

_start:
    ...
reset:
```



```
ldr r3, =_start
mov    r4, #0x0000
ldmia r3!,{r5, r6, r7, r8, r9, r10, r11, r12}
stmia r4!,{r5, r6, r7, r8, r9, r10, r11, r12}
ldmia r3!,{r5, r6, r7, r8, r9, r10, r11, r12}
stmia r4!,{r5, r6, r7, r8, r9, r10, r11, r12}
```





# Setting up the Stacks

- Need chunk of memory for each stack
- Temporarily switch to mode, then set the stack pointer
- You can manually (without getting an interrupt) set the CPSR value with a `msr` instruction (move to status register)
- We start in SVC mode (Well, on pi2+newer HYP mode) but we can get to a mode where we can change CPSR



# Pi2/Pi3 Memory Map

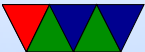
<b>GPU</b> mirrored with different caching rules?	0xffff ffff	(4GB)
<b>Peripherals</b>	0x4000 0000	(1GB)
<b>RAM Reserved for GPU</b>	0x3f00 0000	(1GB-16MB)
		256MB?
<b>Device Tree</b>	0x2f00 0000	
<b>Unused RAM</b>	0x2eff b500	
<b>Our Operating System</b>		
<b>System Stack</b>	0x0000 8000	(32k)
<b>IRQ Stack</b>	0x0000 4000	(16k)
<b>ATAGs</b>	0x0000 0100	(256)
<b>IRQ Vectors</b>	0x0000 0000	



# Setting up the Stacks

```
/* Set up the Interrupt Mode Stack      */
/* First switch to interrupt mode, then update stack pointer */
/* cpsr_c means just change the config (mode) registers */
mov     r3, #(CPSR_MODE_IRQ | CPSR_MODE_IRQ_DISABLE | CPSR_MODE_FIQ_DISABLE )
msr     cpsr_c, r3
mov     sp, #0x4000

/* Switch back to supervisor mode */
mov     r3, #(CPSR_MODE_SVC | CPSR_MODE_IRQ_DISABLE | CPSR_MODE_FIQ_DISABLE )
msr     cpsr_c, r3
```



# Clearing the Interrupt Status Bit

```
/* or use the "cpsid i" instruction? */
```

```
_enable_interrupts:  
    mrs     r0, cpsr  
    bic     r0, r0, #0x80 ; bit clear  
    msr     cpsr_c, r0  
  
    mov     pc, lr
```



# Timer interrupt

- Most OSes have some sort of internal timer keeping things going
- Tracks time of day, triggers scheduler (for context switching), uptime, time accounting
- Ideally triggered as a regular interrupt



# Timer interrupt – Linux

- Traditionally 100Hz, these days 250Hz?
  - Too slow and the delay in context switching is noticeable
  - Too fast and the overhead from each interrupt adds up (have to stop CPU, save/restore state, etc)
- Ancient Linux/UNIX has time value updated every 1s since Jan 1st 1970. On 32-bit machines will overflow in Jan 2038
- Interrupts can waste power, especially if machine is



mostly idle/sleeping

Linux these days has ways to run without timer-ticks

- Trivia, timer tick on Linux called a jiffie



# Timer interrupt – Other Operating Systems

- DOS = 18.2Hz
- Windows = 64Hz





# Configuring a Timer

- Section 14 of BCM2835 Peripheral manual.
- Similar, but not exactly the same, as an ARM SP804
- There are also the system timers (4 timers described in Section 12).
- Note that the timer we use is based on the APB clock which ticks at 250MHz
- Limitations: it scales with the system clock, so frequency might change
- Important registers



- `TIMER_LOAD`: set a value and it will count down on each tick and give interrupt when zero. Automatically re-loaded after interrupt.
- `TIMER_CONTROL`: start/stop, interrupts on/off, scaling
- `TIMER_RELOAD`: queue a different value to be loaded into `TIMER_LOAD` automatically when current hits zero
- `TIMER_IRQ_CLEAR`: clears the interrupt
- `TIMER_PRE_DIVIDE`: another divider, as original design was for 1MHz clock



```

/* Timer is based on the APB bus clock which is 250MHz on Rasp-Pi */

int timer_init(void) {

    uint32_t old;

    /* Disable the clock before changing config */
    old=bcm2835_read(TIMER_CONTROL);
    old&=~(TIMER_CONTROL_ENABLE|TIMER_CONTROL_INT_ENABLE);
    bcm2835_write(TIMER_CONTROL,old);

    /* First we scale this down to 1MHz using the pre-divider */
    /* We want to /250. The pre-divider adds one, so 249 = 0xf9 */
    bcm2835_write(TIMER_PREDIVIDER,0xf9);

    /* We enable the /256 prescalar */
    /* So final frequency = 1MHz/256/61 = 64.04 Hz */

    /* The value is loaded into TIMER_LOAD and then it counts down */
    /* and interrupts once it hits zero. */
    /* Then this value is automatically reloaded and restarted */

    bcm2835_write(TIMER_LOAD,61);

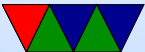
```



```
/* Enable the timer in 32-bit mode, enable interrupts */
/* And pre-scale the clock down by 256 */
bcm2835_write(TIMER_CONTROL,
              TIMER_CONTROL_32BIT | /* In manual 23bit typo */
              TIMER_CONTROL_ENABLE |
              TIMER_CONTROL_INT_ENABLE |
              TIMER_CONTROL_PRESCALE_256);

/* Enable timer interrupt */
bcm2835_write(IRQ_ENABLE_BASIC_IRQ, IRQ_ENABLE_BASIC_IRQ_ARM_TIMER);

return 0;
}
```



# Sample Interrupt Handler

- CPU disables interrupts, switches CPSR to correct mode
- Save registers (no need to save SPSR unless nested)
- Interrupt handler checks and sees which interrupt was triggered (in a register)
- Interrupt Status Routine (ISR) called which services the routine and then acknowledges interrupt
- Handler restores context, returns
- CPU restores execution



# Sample Interrupt Handler

```
void __attribute__((interrupt("IRQ"))) interrupt_vector(void) {
    static int lit = 0;
    int which;

    /* Check to see what interrupt we had */
    which=bcm2835_read(IRQ_BASIC_PENDING);
    if (which&0x1) {

        /* Clear the Timer interrupt */
        bcm2835_write(TIMER_IRQ_CLEAR,0x1);

        /* Flip the LED */
        if( lit ) { led_off(); lit=0; }
        else {led_on(); lit=1; }
    }
}
```



# IRQ Handlers in C

In gcc for ARM, you can specify the interrupt type with an attribute. Automatically restores to right address.

```
void function () __attribute__((interrupt ("IRQ")));

/* Can be IRQ, FIQ, SWI, ABORT and UNDEF */

void __attribute__((interrupt("UNDEF"))) undefined_instruction_vector(void) {

    while(1) {
        /* Do Nothing */
    }
}
```

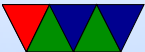


# Enabling Interrupts in C

```
static inline uint32_t get_CPSR(void) {
    uint32_t temp;
    asm volatile ("mrs_␣%0,CPSR":"=r" (temp):) ;
    return temp;
}

static inline void set_CPSR(uint32_t new_cpsr) {
    asm volatile ("msr_␣CPSR_cxsf,%0"::"r"(new_cpsr) );
}

/* enable interrupts */
static inline void enable_interrupts(void){
    uint32_t temp;
    temp = get_CPSR();
    set_CPSR(temp & ~0x80);
}
```





# Writing a Command Parser

- Read in values one character at a time into string buffer.
- Read until Enter (slash r)
- Be sure to NUL terminate! Also be sure to not overflow buffer!
- How to parse? `strtok()`? `strcmp()`?  
Who provides these string routines?
- Simple way to do things is to manually check, like  
`if ((buffer[0]=='l') && (buffer[1]=='s')) somet`



# HW#4 Note

- See the next lecture for some more info on HW#4

