ECE 271 – Microcomputer Architecture and Applications Lecture 16

Vince Weaver

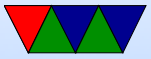
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

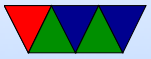
22 March 2022

Announcements

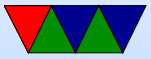
- Read Chapter 11
- Midterm graded
- Lab #7 is happening



Hand back Midterms

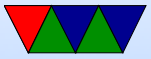


Continued Review of Interrupts on the Cortex-M Processor



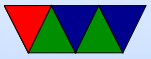
Clocks on the STM32L4Discovery

- HSI16 – high speed internal – 16MHz clock
- MSI – multi-speed internal RC clock – 100kHz to 48MHz
  Note these are RC and best effort and "approximate"
- HSE – high speed external, 4-48MHz
- PLL – phase locked loop, complicated, but in our case essentially a clock multiplier. That's how the board can get up to 80MHz
- Why run at lower speeds? CMOS power equation, power usage is linear with frequency (and square of voltage)



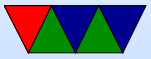
Interrupts Review (from last lecture)

- Cortex-M has 255 interrupts.
  -1 to -15 built-in, 0-240 external
- When something triggers interrupt (traditionally pull a line low) stops execution and jumps to interrupt handler
- With vector interrupt handler, a vector each with an address for each handler, look up in table from interrupt number and jump
- Interrupt handler code needs registers, but before we can use them we need to save the old ones.



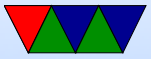
Interrupt Register State Saving

- Cortex-M saves some registers on the stack for you at interrupt time (note: not all processors do this)
- R0,R1,R2,R3, R12,PSR,LR,PC saved to stack (PSR is the process status register, has the NZCV flags)
- Note: by default saves on the "MSP – Main Stack Pointer" To confuse things there's also a special "PSP – Process stack pointer" if you're running applications like with an OS



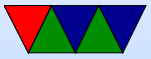
Cortex-M Interrupts Handler

- An interrupt handler in C looks just like a function
- On some platforms you might have to specially annotate the function, or include special assembly code to properly return
- On Cortex-M you don't have to, it looks just like a regular function call

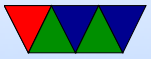


General Interrupts Handler Coding

- You may need to "ACK" the interrupt, let the hardware know you are handling things so it can stop asserting the IRQ line
- If you don't do that, the interrupt might immediately trigger again as soon as you return
- Some architectures you might have to manually disable interrupts at the beginning/restore at end if you don't want your interrupt handler interrupted by other interrupts (we won't need to do that here)

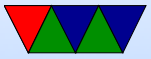


- Then have code that does what you need.
  - Inside of an operating system usually you want you interrupt handler to be as short as possible to avoid perturbing the main code running
  - For small embedded systems you can alternately write your code so it is mostly idle and all the processing happens in interrupt handlers

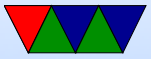


Returning From Interrupt Handler

- Cortex-M we can return just like a regular return (some architectures require a special return-from-irq instruction... not Cortex-M though)
- Cortex-M does weird stuff with Link Register – special value with `FFFF` in high bits that indicates we are returning from an IRQ handler and that the return value is on the stack (more info on this in the textbook/manual)
- Ideally the main code running on the processor doesn't

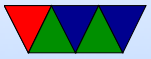


even notice an interrupt happened



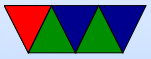
Writing the interrupt handler

- Looks like regular C function
- Our Lab#7 code,just decrements a global variable each time it is called, stopping at zero.
- We call it `SysTick_Handler()`



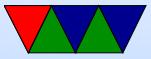
Setting up the Interrupt Handler

- How does its address get in the right slot? (address 0x58)
- Linux, we put its address there in an array that gets put in to the right place by the linker script.
- Keil does another trick, there's an existing `SysTick_Handler()` declared as a WEAK symbol. That means it's lesser priority, so if we define a function with the same name the linker will replace the other one with ours.



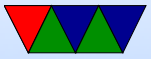
Behind the Scenes – Linker Scripts

- The C compiler outputs assembly language, the assembler makes object code, and a linker then merges that together and makes the final binary image (or executable)
- Often your system/OS expects certain parts of the code to be in certain locations, such as in this case the IRQ vectors need to be first
- There is a script (text file) we pass to the linker that explains what code should go where. Usually someone



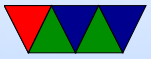
will provide this for you (or it comes with your system) so you don't have to write it yourself

- There are usually compiler directives you can put in front of variables/functions that can force the code to end up in specific "sections" so the linker puts them in the right place



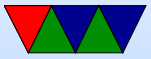
Concurrency issues in IRQ Handler

- Our timer interrupt handler just loads a global variable and decrements it
- Is this safe? Maybe. What if the write is halfway done and gets interrupted by the interrupt? More a problem if you imagine a two-part value being set, like minutes:seconds. Race condition. Locking? Disabling interrupts?
- Does this global value need to be marked volatile?



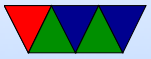
The Delay Function

- The `delay()` function just spins in a tight loop waiting for the global variable being decremented by the timer interrupt to hit zero
- This might seem to be a waste of power
  How could we do it differently?
- One would be to use a WFI instruction to sleep while waiting
- Another would be to put the actual blinking code in the IRQ handler and have the main thread just be sleeping



Setting up/enabling Interrupts

- Note this and SysTick described in Cortex-M4 Devices Generic User Guide DUI0553.pdf not in the STM32L4 manual
- The SysTick and internal interrupts do not have to be specially enabled or acknowledged



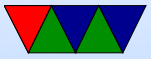
Nested Vector Interrupt Controller (NVIC)

- Used for the non-system interrupts
- Interrupt Set Enable Register – (ISER0–ISER7) note, this is like the BSRR register, 1 means enable, 0 means do nothing
- Interrupt Clear Enable Register (ICER0–ICER7)
- Setting/clearing. Bitmask, so 32-bits

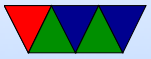
```
word_offset=irq_num>>5;    // why?
bit_offset=irq_num&0x1f;   // why not % 32
NVIC->ISER[word_offset]=(1<<bit_offset);
```

- Assembly – note byte vs word addressing

```
// irq to enable in r0
```

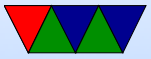


```
ldr r4,=NVIC_BASE

lsr r1,r0,#5     // get word_offset into r1
lsl r1,r1,#2     // change to byte offset, mulx4
add r1,r1,#NVIC_ISER0

and r2,r0,#0x1f // get bit offset in r2
mov r3,#1
lsl r3,r3,r2

str r3,[r4,r1]
```



Interrupt Priority

- Cortex-M supports 16 priorities for interrupts
- Generally the top nibble (4-bits) of a byte you have to set
- Nested interrupts – a higher priority interrupt can interrupt a lower priority one



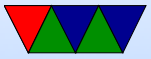
Setting Priority (check textbook for more info)

- SHP (system handler priority) for internal like systick
  - Byte array in the SCB (System Control Block)

```
SCB->SHP[(((uint8_t)irq)&0xf)-4]=(priority<<4)&0xff);
```

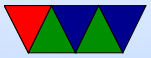
- For the external ones
  - there's the IP (interrupt priority register) in the NVIC structure.

```
NVIC->IP[irq]=(priority<<4)&0xff;
```



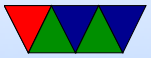
Global Interrupt Enable/Disable

- Even once everything is configured, you can disable/enable interrupts globally in software
- Why might you want to do this?
  - To avoid interrupts before you have handlers installed
  - To turn off interrupts before critical chunks of code you don't want interrupted
  - To turn off interrupts in code that is unsafe (like moving the stack around)



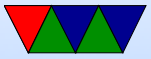
Change Processor State (CPS)

- CPS (change processor state) instruction – pseudo instruction that sets the PRIMASK (priority mask) register
- CPSID i – disable interrupts
- CPSID f – disable fault handlers
- CPSIE i
- CPSID f
- Can also set priority mask manually to disable interrupts above a certain level. Need MSR instruction as it's a



special register

- The way to do this is the `CPSIE I` assembly language instruction.



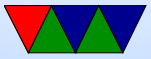
CPS in C

- Can we do this in C? We'll have to use inline assembly.
  - On Keil, you can do this:

```
__asm("CPSIE␣i");
```

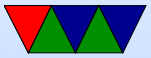
  - On Linux it will look like:

```
asm volatile ( "cpsie␣i" );
```



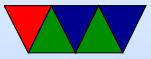
NMI – Non-maskable Interrupts

- An interrupt that cannot be stopped
- What are the useful for?
- Watchdog timers?
- Hacking, performance counters?



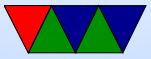
WFI/WFE instructions

- Wait for interrupt or event
- CPU goes into a low-power mode (sleep) waiting for an interrupt
- Can save power when nothing else is going on
- Use SEV to send event to wake from WFE (used on Pi to start secondary processors)



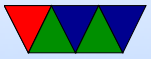
Inline Assembly

- Inline assembly
- Note, gcc/Linux does this a lot more annoyingly
- To work with gcc, you have to let it know the inputs/outputs to the code (what to put into registers), and also list any register/memory locations you change (called "clobbers") so the C compiler doesn't try to optimize/use the things you're changing behind its back

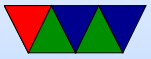


Inline Assembly Examples

```
__asm int sum4(int a, int b, int c ,int d) {
    push {r4,lr}
    mov  r4,r0
    add  r4,r4,r1
    add  r4,r4,r2
    add  r0,r4,r3
    pop  {r4,pc}

int sum4(int a, int b, int c, int d) {
    int t;
    __asm {
        add t,a,b
        add t,c
        add t,d
    }
    return t;
}



\begin{lstlisting}
```



```
long shell_address=current_proc[0]->user_state.pc;
long cp_address=(long)&(current_proc[0]);

        asm volatile(
                "mov␣␣␣␣lr,␣%[shell]\n"
                "mov␣␣␣␣sp,␣%[cp]\n"
                "ldr␣␣␣␣sp,[sp]\n"
                "mov␣␣␣␣r0,␣#0x1f\n"    /* system mode, IRQ enabled */
                "msr␣␣␣␣SPSR_cxsf,␣r0\n"
                "movs␣␣␣pc,lr\n"
                : /* output */
                : [shell] "r"(shell_address),
                  [cp] "r"(cp_address)  /* input */
                : "r0", "lr", "memory");        /* clobbers */
```



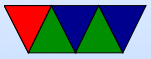
Calling Assembly from C (10.4)

```
/* main.c */

extern int sum4(int a, int b, int c, int d);

int main(int argc, char **argv) {

    x=sum4(1,2,3,4);

    while(1);
}
```

This is where the ABI excels.

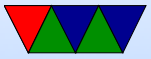
Linux directives are a bit different. `.globl`

```
/* sum4.s */

    EXPORT sum4
sum4    PROC
    push {r4,lr}
    mov  r4,r0
```



```
add   r4,r4,r1
add   r4,r4,r2
add   r0,r4,r3
pop   {r4,pc}
ENDP
```

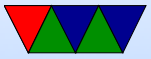
Strong and weak symbols?



Calling C from Assembly (10.4)

```
/* sum.c */

/* note, not declared static */
int sum2(int x, int y) {

    return x+y;

}
```

```
/* main.s */

    IMPORT sum2

    ENTRY

__main  PROC

    mov r0,#1
    mov r1,#2
    bl  sum2

stop    b   stop
```



```
ENDP
```