CSE/ECE474 Interrupt Concepts: SP23

This document goes over key interrupt ideas at the conceptual level. Details vary according to your specific chip. Here are the key concepts to understand and work with interrupts (be sure to read ISR writing tips at the end).

Definitions

Interrupt Service Routine (ISR): When an interrupt is generated, some code, called an ISR, must be run immediately to "service" the interrupt. "Service" means to perform tasks needed by the event that caused the interrupt. For an extremely simple example, if a timer is causing regular clock interrupts, an ISR might just increment a variable called "time" and return. A more advanced example would be a disk drive which earlier got a command to read data from a hardware disc sector. After the sector read is completed, the drive generates an interrupt, and the ISR must grab the data from the drive and put it into the right place in memory.

Enabling interrupts: Just like when you concentrate on some reading or writing task, interruptions are costly for the CPU. Furthermore, if there is no ISR defined for the interrupt, then acting on that interrupt could hang (freeze) the system. Therefore there are bits controlled by software that enable or disable interrupts from hardware signals. By default all the interrupts are disabled. In addition to enabling individual interrupts, the CPU has bits that can globally enable or disable **all** interrupts.

Internal Interrupts: Interrupts from components inside the computer itself such as timer/counters.

External interrupts: Interrupts from components outside the CPU such as a digital input bit, a network interface, or an SSD.

Interrupt Priorities: Interrupts can interrupt ISRs(!) ... or not! You might have one interrupt which is a very time-critical control function, and a plane will crash if this ISR does not execute correctly in 2milliseconds. You might also have another ISR that updates a log file. The first ISR is therefore a much higher priority than a log-file update. Yes, we need to guarantee that the log file gets updated, but the speed does not matter. Most processors group the available interrupts into **priority levels** such that lower priority interrupts have to wait until there are no running ISRs of higher priority.

Interrupt "vectors": A powerful computer system could have many different interrupts that are enabled at a given time. A simple example would be a Network-Attached-Storage (NAS) device which is basically a computer with a bunch of disk drives (or SSDs) attached to it to be a file server. If there are four drives attached to the NAS, each one needs a separate interrupt so that they can operate simultaneously. The way to implement this is for each hardware interrupt mechanism to have a pointer to a software ISR function. The address of the ISR for each interrupt is the "Interrupt Vector".

Interrupt Vector Table: This is a hardware-specific region of memory that is specially set aside (and hardware-defined) to hold an interrupt vector for each hardware interrupt the processor can support. If the interrupt is not enabled then it doesn't matter what is in the table, but for every enabled interrupt, there must be a valid pointer to a function that can run and return.

Setting up an Interrupt: In view of the previous definitions, setting up an interrupt to properly respond to an event involves several steps:

Hardware Steps:

- If the interrupt is external, wire it to a specific pin of the chip. Internal interrupts are already "connected" internal to the chip.

Software Steps (do these *in order* at setup time):

1. For internal interrupts, configure the internal device and set it to generate interrupts under the correct conditions by setting bits in its control register(s). For external interrupts, set the chosen pin to act as an input.
2. Write an ISR and make a function pointer for it (note that in C, the function name already **is** a function pointer and can be used directly).
3. Copy the function pointer's value into the slot in the interrupt vector table belonging to the specific bit that will cause the interrupt (pin, timer signal, etc). The Interrupt Vector Table for AtMega2560 is in the datasheet: Table 14-1, page 101.
   IMPORTANT: use the `ISR(vector)` macro to do this.
   [Blog Post] [Authoritative AVR reference]
   Example: to increment a counter in a simple isr:

   ```
   ISR(--vector--) {
       Isr_counter++;
       }
   ```

   "`--vector--`" should be a symbol indicating the correct interrupt. It's a little hard to find these symbols but the ones you will need will be:
   - `TIMER`**`n`**`_COMPA_vect` `Output compare event for Timer` **`n`**
4. Enable the specific interrupt (for example, enable an interrupt when pin 6 goes from logic 0 to logic 1). To enable interrupts for the 16-bit Timer/Counters see the datasheet sections 17-11-33 --- 17.11.36. Enable only the specific interrupt you need.
5. The "Status Register" referred to in docs is the master CPU status register (data sheet page 13) and Interrupts should be enabled there by default. You could check that with
   `if (SREG & (1<<7)) { intenabled = 1} ; else intenabled=0;`
   Enable interrupts globally `with SREG |= (1<<7).`

Now write your application without worrying about the interrupt. While your application is running, as an interrupt comes in, your app will be stopped, the ISR will be started, and when

the ISR returns, your application will resume. For example, consider the simple ISR above for keeping track of elapsed clock time.

ISR Writing Tips: When there are multiple tasks, threads, interrupts, happening all at once, debugging can keep you up at night! The number one tip for writing ISRs that are reliable and easy to debug is to **keep the ISR as short as possible.** That means as few statements as possible and, importantly, call as few external routines. **Do not use delay(x) in an ISR.** The quicker your ISR gets done and returns to the main process, the more understandable your execution will be and the less chance of something going wrong.

Often you have to share a variable between the main code and the ISR. For example, for the time keeping ISR, we need to access the time in the main code by e.g.

```
Time = Isr_counter/1000;  // convert milliseconds to seconds
```

The problem is that modern C-compilers don't see any line in your main code which at any time modifies `Isr_counter` so it thinks it is a constant and gets rid of it in the machine code output (memory optimization). The ISR will have it's own private copy that the main code can't see. To prevent this, use the **volatile** keyword to signal that this value might change due to an ISR:

```
volatile int Isr_counter=0;
```

And define it outside your main (or loop()) functions so that it is global to both the ISR and your main code.

A common beginner mistake to avoid is to try to debug an ISR by printing something (e.g. `printf()` or `Serial.print()`) within the ISR. Compared to simple operations like `x++`; or `digitalWrite(pin, HIGH);`, print statements take a really long time and can have big side effects like blocking a bunch of other interrupts. Flashing LEDs can be very useful here (at least for ISRs that can be tested at low frequencies.). We can debug faster ISRs using the oscilloscope to observe output pins set and cleared by the ISR for debugging purposes.

Follow the 6 TL:DR tips HERE.

474-Specific notes:

attachInterrupt() You may find reference to this Arduino function, but this **does not** simplify your assignment compared to the recommended approach and should not be used.

You may **not** use any external libraries for setting up the interrupt, particularly the **TimerInterrupt** library.