Overview of Interrupts in ATMEL AVR

Kartman

Intro

Interrupts seem to have some confusion and ignorance associated with them. I hope to remove some of this and replace it with understanding so one can use interrupts with confidence.

What is an interrupt?

Interrupts were devised as a means of signalling the processor that a hardware device wanted attention. Without interrupts, the software would have to regularly ask the device(s) if they were ready and then service them if they were. Between this 'polling', you could do some real work. The problem was some devices need fast attention, so if you didn't poll the device fast enough, data would be lost. Enter the 'interrupt'. An interrupt is effectively a subroutine call activated by a hardware signal. The processor at its simplest fetches an instruction then executes it. Repeat ad-infinitum. With interrupts, the processor checks to see if the interrupt signal is active. If it is active, it will execute then means needed to call the interrupt service routine (ISR). Otherwise, the processor will test for interrupt, fetch the instruction, execute it, repeat.....

AVR Interrupts

With the AVR, we have many sources of interrupts - timer, external, uart etc. Each source has a 'vector' associated with it. This is simply the address of the ISR to be called when the requisite interrupt is activated. We also have enable flags and interrupt active flags for each of the interrupt sources. This means we can enable/disable each interrupt source as we see fit. The AVR also has a global interrupt enable flag that is set/reset with the SEI (enable interrupts) and CLI (disable interrupts) instructions.

Multiple Interrupt at one time

Since we have multiple interrupt sources that we can enable, what happens if a number of them are active at one time? Priority. The AVR has a fixed priority scheme to sort out what source it will service. Lower priority sources will have to wait their turn. Many of the devices have an interrupt active flag associated with them. In the case of the timers, each source of interrupt has a flag that gets set when a timer event occurs. This flag stays set until the AVR actually calls the ISR for that interrupt source. so with this, an event won't get missed but the response may be delayed. This is a good reason to make all your ISRs lean and mean. Get the job done and get out - no sitting in loops or wasting time.

In a nutshell, the AVR will scan for all active interrupt sources, the highest priority wins, disable all further interrupts by clearing the I bit in the status register then it will get the vector (address of the ISR) for that source then call that ISR. Simple. When you do a RETI instruction (in 'c' the compiler does this for you at the end of an ISR), the I bit is set so other interrupts can be serviced.

The down side of interrupts

So we've had an interrupt and the AVR has called our ISR, what next. Because we've interrupted other code and the ISR may change registers etc, we need to save the SREG (status register) and any other registers we modify. This is called 'preserving the processor state'. Once we've done this, we can alter the registers as we please. Once our ISR has completed, we need to restore the saved registers and do a RETI (return from interrupt instruction). In 'C' the compiler does this for us.

So far so good.

For most programs, we need to share variables between the ISR and the rest of our code. This is where problems can arise. We need to share our variables carefully, otherwise an ISR might occur at any time and change something when we're not looking.

Atomicity
Certain operations need to be done without interruption - 'atomically' which means 'as one' or 'indivisable'. This needs to happen when:

1. Reading/writing variables that are larger than one byte. The AVR being an 8 bit cpu, reads/writes in 8bit chunks. so to read/write a int variable (in C), we need to r/w in two chunks, the low byte and the high byte. This takes a few processor instructions to achieve this. What happens if an interrupt occurs between these two operations and the ISR modifies the variable we're accessing? We get half the unmodified variable and the other half modified. Of course this occurs randomly and most likely very infrequently as both interrupt and the variable access have to happen at the right time. This causes the worst kinds of bugs - so very hard to track down. So how do we get around this problem? Disable interrupts, if reading a shared variable copy the shared variable into another variable, if writing, perform the write, then re-enable interrupts. If any interrupts occur whilst we have them disabled will be processed after we reenable them. In 'c' the code might look like this:

```
volatile unsigned int shared_var;
```

```
in main code:
cli();
copy_of_shared_var = shared_var;
sei();
// use copy of shared var
```

Note that we have declared the variable 'volatile'. With optimising compilers, they try to keep used variables inside the AVR registers for performance. The problem is if an interrupt comes along and modifies the variable that is stored in ram, next time the main code reads the variable (that the compiler has stashed in the registers), we're not reading the current value of that variable. Declaring a variable 'volatile' tells the compiler not to stash a copy away in the registers but to read/write the variable to/from memory each time it is accessed.

Note that the above applies also to arrays and collections of data that you expect to be cohesive.

2. When testing and modifying a variable. A common operation that needs to be protected is a 'test and set'. If the value is 0, we set the variable to a non-zero value. Here we have two operations that might get interrupted and cause and unwanted condition. Again, we must temporarily disable interrupts, do the test and set and then re-enable interrupts.

So, we need to remember that interrupts can happen at any time and ISRs can modify shared variables whilst the main code is performing operations on them. These are called 'critical sections'. At the assembler level, it is obvious to the programmer that there are a number of instructions involved in a particular operation, whereas in'c' and other higher level languages it is not so obvious that one line of code may generate a number of assembler instructions.

The basic rules are:
1/ Understand the pitfalls of shared variables
2/ minimise the use of shared variables
3/ Implement 'critical sections' where necessary

This article highlights what can go wrong when ignoring the above:
http://courses.cs.vt.edu/~cs3604/lib/Th ... rac_1.html

Pre-emptive RTOS
The term RTOS seems to have some magic associated with it. 'Real time Operating System' is what it stands for. so what is 'real time'? Historically it means that the operating system will activate a task within a given time. The actual time varies, but is usually taken to be 'as fast as possible' - so a faster cpu will respond faster in most instances. The 'operating system' in many cases (especially with AVR) is simply a task switcher not something like Windows(c).

Task Switching
There are two major type of task switching:
1/ co-operative
2/ Pre-emptive

Co-operative is the simplest - each of your tasks must perform its duty then return. Then the next task runs etc. As such there is only one 'thread' of execution.
Each task must 'co-operate' otherwise other tasks won't get a chance of running.

Pre-emptive is a little more complex. Pre-emptive means that a running task may be interrupted and suspended at any point in its execution. This is done with interrupts. The operating system keeps a stack for each of the tasks and when a task swap is needed, it swaps to the required stack and restores the previous processor state. The main downside of this is due to the number of stacks, this consumes an amount of memory which may be in short supply in a small system like the AVR.

Usually a timer is set up to give a regular interrupt that calls the operating system scheduler. The scheduler code determines what task should run next. Interrupts from other sources may call the scheduler. The design of the scheduler determines if the system is 'real time' or not. Windows and Linux are examples of pre-emptive operating systems but they are not real-time as certain tasks may continue to run until they relinguish control back to the scheduler.

There may be some conjecture as to my description of real time as the term has been misused so much that its meaning has been lost in the mire. I base my description on the historical use of the term.

With a pre-emptive operating system real time or not we again have the problem of sharing variables. Many operating systems give you system calls to take care of this - flags, queues etc. Use there where possible as the operating system takes care of any sharing issues (well at least it should!). Sometimes we still need to share variables between tasks so the issue with atomicty is the same - use critical sections to ensure your operations are atomic. With a co-operative system, variable sharing isn't a problem (except with ISRs) as each task runs to completion.

Old tricks
There's usually a question that pops up on the forums a lot on how to create a 'software interrupt'. Some processors have this facility built it but the AVRs don't. There's a variety of ways of making this happen - set up a port pin for external interrupts and toggle the port pin in software is a popular one. What the person is wanting though is a task switcher. In this instance, the person should evaluate the design of the code to eliminate this requirement or go to a pre-emptive o/s like freertos to provide a more general solution to the problem. Personally, I avoid using a pre-emptive task switcher on AVR class projects as most applications don't need it. Clever use of a timer interrupt and careful design can yield a solution using a co-operative method.

Using a co-operative method avoids a lot of potential pitfalls associated with pre-emptive strategies. Call it 'problem avoidance'. By avoiding methods that might introduce tricky problems in debugging means you have a better chance of writing defect free code - which should be a 'good thing'.

Switches and Interrupts
It seems like a perfect marriage, switches activating a pin change or external interrupt - but there are dangers lurking. These are:

1/ Switch bounce - mechanical switches 'bounce' as in giving multiple on/off actions. This usually happens in the range of 5 - 50mS. So hooked up to an interrupt, each press of the switch can give you a random amount of interrupts for each press.

2/ Picking up EMI. Say we had a switch connected by 10M of wire to our AVR with an interrupt. Without protection the wire may pick up unwanted radiation from a mobile phone or other unit. This would give us a burst of interrupts in fast succession - so fast that our poor AVR might be hard pressed to keep up. The net result is our application on the AVR doesn't work as we would like. Even without a long length of wire, we still have the potential to pick up unwanted interference.

Put simply ,we have little control of the interrupts which may introduce some unreliability into our unit. So how do we avoid this?

Use a timer interrupt to read the switch input(s) and apply a debounce algorithm. Even if the input is not a mechanical switch, it always pays to apply some form of filtering to remove transient spikes etc from affecting the rest of our code.

If you want the switch to power up the AVR, use an interrupt here, but disable it once you've powered up then use a timer interrupt to read the switch.

Where possible, minimise or eliminate the use of external interrupts. I'm not saying 'never do it', but if you have to do it, be aware of the potential problems and take steps to minimise them.

If you've got this far hopefully you've been able to understand my waffle. If something is not too clear or you need clarification, drop a message and I'm try to correct/explain it.