Newbie's Guide to AVR Interrupts

By Dean Camera, 2010

Hi all. This is a small guide to teach those new to the embedded world about the wonders of processor interrupts. Feedback appreciated.

Part 1: What are Interrupts?

One of the most fundamental and useful principles of modern embedded processors are *interrupts*. There seems to be a lot of confusion about interrupts; what are they, and how do we use them? In short, an interrupt is a way for an external (or, sometimes, internal) event to pause the current processor's activity, so that it can complete a brief task before resuming execution where it left off. Consider the following:

Let's say we are at home, writing an excellent tutorial on how a principle of modern embedded processors works. We are very interested in this topic, so we are devoting all our concentration to our keyboard. However, half way though, the phone rings. Despite not being by the phone waiting for a call, we are able to stop what we are doing, take the call, and go back where we left off once we have hung up.

This is how a processor interrupt works. We can set up the processor so that it is looking for a specific external event (like a pin going low, or a timer overflowing) to become true, while it goes on and performs other tasks. When these events occur, we stop the current task, handle the event, and resume back where we left off. This gives us a great deal of flexibility; rather than having to actively poll our events to see if they have happened, we can instead just ignore them completely and trust our interrupt routines to process them instead.

What we are doing is called *asynchronous* processing - that is, we are processing the interrupt events outside the regular "execution thread" of the main program. It's important to note that we are not doing two things at once (although it may appear that way) - the regular program is stopped while an interrupt routine runs.

When correctly set up, we can link a specific interrupt source to a specific handler routine, called an *Interrupt Service Routine*, or *ISR* for short.

Part 2: What can cause interrupts?

There are two main sources of interrupts:

- **Hardware Interrupts**, which occur in response to a changing external event such as a pin going low, or a timer reaching a preset value
- **Software Interrupts**, which occur in response to a command issued in software

The 8-bit AVRs lack software interrupts, which are usually used for special operating system tasks like switching between user and kernel space, or for handling exceptions. Because of this, we'll only be looking at hardware interrupts.

Each AVR model contains a different set of interrupt sources. We can find out which interrupts our chosen model has by looking in the "Interrupts" chapter of the AVR's datasheet. This chapter contains a table, similar to the following fragment from the AT90USB1287 datasheet:

Code:
```
Vector No. | Address | Source | Interrupt Definition
2          | $0002   | INT0   | External Interrupt Request 0
3          | $0004   | INT1   | External Interrupt Request 1
4          | $0006   | INT2   | External Interrupt Request 2
```

This contains a complete list of all the possible interrupts in our AVR, giving each interrupt's Vector table address, Source Interrupt Name, and Source Interrupt Description. Take a look at your AVR's list - see the variety of interrupt sources, from all sorts of on-chip peripherals of the AVR.

Hardware interrupts can be useful to process sparingly occurring events, such as buttons presses or alarm inputs. They can also be useful for situations where a low, fixed latency is required, for example when processing inputs from a rotary encoder.

Part 3: So how to we define an ISR?

How we do interrupts varies between programming languages, but at the lowest level, we are simply making a general AVR function somewhere in the AVR's FLASH memory space, and "linking" it to a specific interrupt source by placing a JMP or RJMP instruction to this function at the address specified in the table we just looked at. At the start of the AVR's FLASH memory space lies the *Interrupt Vector Table*, which is simply a set of hardwired addresses which the AVR's processor will jump to when each of the interrupts fire. By placing a jump instruction at each interrupt's address in the table, we can make the AVR processor jump to our ISR which lies elsewhere.

For an ISR to be called, we need three conditions to be true:

- **Firstly**, the AVR's global *Interrupts Enable bit* (I) must be set in the MCU control register SREG. This allows the AVR's core to process interrupts via ISRs when set, and prevents them from running when cleared. It defaults to being cleared on power up, so we need to set it.

- **Secondly**, the individual interrupt source's enable bit must be set. Each interrupt source has a separate interrupt enable bit in the related peripheral's control registers, which turns on the ISR for that interrupt. This must also be set, so that when the interrupt event occurs the processor runs the associated ISR.

- **Thirdly**, The condition for the interrupt must be met - for example, for the *USART Receive Complete* (USART RX) interrupt, a character must have been received.

When all three conditions are met, the AVR will fire our ISR each time the interrupt event occurs.

Again, the method used to define an ISR differs between langages and compilers, so I'll just put the AVR-GCC and AVR ASM versions here.

Code:
```
#include <avr/interrupt.h>

ISR({Vector Source}_vect)
```

```
{
   // ISR code to execute here
}
```

Code:
```
.org 0x{Vector Address}
jmp MyISRHandler


MyISRHandler:
  ; ISR code to execute here
  reti
```

Note that in the case of the assembly version, we need to add a "reti" instruction at the end of our interrupt instead of the usual "ret" instruction to return to the main program's execution; this special instruction has the dual function of exiting the ISR, and automatically re-enabling the *Global Interrupt Enable* bit. This happens inside the C version too when the function returns, we just don't see it normally.

This raises the next point; **by default, interrupts are themselves not interruptable**. When an interrupt fires, the AVR CPU will automatically disable the *Global Interrupt Enable* bit, to prevent the ISR from being itself interrupted. This is to prevent stack overflows from too many interrupts occurring at once and to prevent the ISRs from running too long, as most uses of interrupts are to have minimal latency when processing an event. It's perfectly possible to set the *Global Interrupt Enable* bit again as part of the ISR so that nested interrupts can occur, but this is highly not recommended as it is dangerous.

Part 4: Enabling a Specific Interrupt?

If you simply add in an ISR to your existing program, you will find that it appears to do nothing when you try to fire it. This is because while we have defined an ISR, we haven't enabled the interrupt source! As mentioned in part 3, we need to meet all three criteria for the ISR to fire.

Firstly, we need to set the I bit in the SREG register. This is the *Global Interrupt Enable* bit, without which the AVR will simply ignore any and all interrupts. In assembly, we have two special single-cycle instructions for dealing with the I bit:

sei, which SEts the I flag
cli, which CLears the I flag

While in C, we have to use special macros from our libc library's header. In the case of AVR-GCC and its avr-libc library, we just use the **sei()** and **cli()** macro equivalents defined in <*avr/interrupt.h*>:

Code:
```
sei ; Enable Global Interrupts
```

Code:
```
sei(); // Enable Global Interrupts
```

Next, we also need to enable a specific interrupt source, to satisfy the second condition for firing an ISR. The way to do this varies greatly between interrupt sources, but always involves setting a specific flag in one of the peripheral registers. Let's set an interrupt on the *USART Receive Complete* (USART RX) interrupt. According to the datasheet, we want to set the *RXCIE* bit in the *UCSRB* register:

Code:
```
in r16, UCSRB
ori r16, (1 << RXCIE)
out UCSRB, r16
```

Code:
```
UCSRB |= (1 << RXCIE);
```

That should be enough to make the AVR's execution jump to the appropriate ISR when the interrupt occurs.

Part 5: Things you Need to Know

There are a few things to keep in mind when using interrupts in your program:

1) When an interrupt is executing, your main program isn't. That means that if your application isn't robust and your receive many interrupts in a very short space of time, your main program's execution will slow to a halt. Keep this in mind when you decided to attach a 4MHz clock input to an external interrupt pin.

2) Data shared between the ISR and your main program **must be both volatile and global in scope** in the C language. Without the *volatile* keyword, the compiler may optimize out accesses to a variable you update in an ISR, as the C language itself has no concept of different execution threads. Take the following example:

Code:
```
#include <avr/interrupt.h>

int MyValue;

ISR(SomeVector_vect)
```

```
{
   MyValue++;
}

int main(void)
{
   SetupInterrupts();

   while (MyValue == 0); // Wait for interrupt

   TurnOnLED();
}
```

There is a good chance the program will freeze forever in the while loop, as (according to the optimizer) the "MyValue" global's value never changes in the loop. By making the global variable volatile, you are declaring to the compiler that the value may change due to circumstances it is not aware of, forcing it to generate code to reload the value each time.

3) There's another subtle bug in the above code sample; it's possible at the machine code level for an interrupt to occur half way between the fetching of the two bytes of the integer variable *MyValue*, causing corruption if the variable's value is altered inside the ISR. This can be avoided by making the fetching of the variable's value in the main program code "atomic", by disabling the global interrupt enable bit while the fetching is taking place. In avr-libc this can be done in the following manner:

Code:
```
#include <avr/interrupt.h>
#include <util/atomic.h>

volatile int MyValue;

ISR(SomeVector_vect)
{
   MyValue++;
}

int main(void)
{
   SetupInterrupts();

   int MyValue_Local;

   do
   {
      ATOMIC_BLOCK(ATOMIC_RESTORESTATE)
      {
         MyValue_Local = MyValue;
      }
   } while (MyValue_Local == 0) // Wait for interrupt

   TurnOnLED();
}
```

Your compiler may vary.

4) Each interrupt source has exactly **one** flag to indicate that the event occurred. If two such interrupts occur while you are in the middle of processing another interrupt's ISR, you will only receive one ISR call when the current ISR terminates. This means that if you have lots of interrupts occurring in a short space of time, you can miss interrupts.

5) The interrupt source flag is usually cleared when the ISR fires. This isn't always the case (some interrupt flags are cleared when a particular register is read, such as the USART receive complete flag) however this is more of an exception to the rule. The upshot of this is that if you receive another interrupt to the source you are currently processing while in its ISR, you'll get another call to the ISR once the current one terminates. Again, you are still limited to a single "event occurred" flag for each interrupt, so you can miss interrupts if many happen at once before you have a chance to process them.

Part 6: Putting it all together

Rather than waste my efforts by repeating what I've already done before here, I will instead direct readers to my previous tutorial on Interrupt Driven USART Communications for a short, practical example of how to use interrupts in an application.

- Dean