Gammon Forum

See www.mushclient.com/spam for dealing with forum spam. Please read the MUSHclient FAQ!

📁 **Entire forum**
  L 📁 **Electronics**
    L 📁 **Microprocessors**
      L 📁 **Interrupts**

Interrupts

Postings by administrators only.

🔁 Refresh page

Posted by	**Nick Gammon** Australia (21,241 posts) 🖼 bio *Forum Administrator*
Date	Sun 08 Jan 2012 03:19 AM (UTC)

Amended on Tue 25 Oct 2016 06:38 AM (UTC) by Nick Gammon

Message   This article discusses interrupts on the Arduino Uno (Atmega328) and similar processors, using the Arduino IDE. The concepts however are very general. The code examples provided should compile on the Arduino IDE (Integrated Development Environment).

> This page can be quickly reached from the link: http://www.gammon.com.au/interrupts

---

TL;DR :

When writing an Interrupt Service Routine (ISR):

- Keep it *short*
- Don't use delay ()
- Don't do serial prints
- Make variables shared with the main code **volatile**
- Variables shared with main code may need to be protected by "critical sections" (see below)
- Don't try to turn interrupts off or on

What are interrupts?

Most processors have interrupts. Interrupts let you respond to "external" events while doing something else. For example, if you are cooking dinner you may put the potatoes on to cook for 20 minutes. Rather than staring at the clock for 20 minutes you might set a timer, and then go watch TV. When the timer rings you "interrupt" your TV viewing to do something with the potatoes.

What interrupts are NOT

Interrupts are not for simply changing your mind and doing something different. For example:

- I interrupted my television viewing to take a bath.
- I was reading a book when visitors arrived and interrupted me. We then went to the movies.
- The user pressed a red button to interrupt the robot walking around.

The above are examples of **doing something different**. They should not be done in an interrupt routine. The most that an interrupt (service) routine might do would be to remember that the red button was pressed. This fact would then be tested in the main program loop.

Example of interrupts

```
const byte LED = 13;
const byte BUTTON = 2;

// Interrupt Service Routine (ISR)
void switchPressed ()
{
  if (digitalRead (BUTTON) == HIGH)
    digitalWrite (LED, HIGH);
  else
    digitalWrite (LED, LOW);
}  // end of switchPressed

void setup ()
{
  pinMode (LED, OUTPUT);  // so we can update the LED
  digitalWrite (BUTTON, HIGH);  // internal pull-up resistor
  attachInterrupt (digitalPinToInterrupt (BUTTON), switchPressed, CHANGE);  // attach interrupt handler
}  // end of setup

void loop ()
{
  // loop doing nothing
}
```

This example shows how, even though the main loop is doing nothing, you can turn the LED on pin 13 on or off, if the switch on pin D2 is pressed.

To test this, just connect a wire (or switch) between D2 and Ground. The internal pullup (enabled in setup) forces the pin HIGH normally. When grounded, it becomes LOW. The change in the pin is detected by a CHANGE interrupt, which causes the Interrupt Service Routine (ISR) to be called.

In a more complicated example, the main loop might be doing something useful, like taking temperature readings, and allow the interrupt handler to detect a button being pushed.

digitalPinToInterrupt function

To simplify converting interrupt vector numbers to pin numbers you can call the function **digitalPinToInterrupt**, passing a pin number. It returns the appropriate interrupt number, or NOT_AN_INTERRUPT (-1).

For example, on the Uno, pin D2 on the board is interrupt 0 (INT0_vect from the table below).

Thus these two lines have the same effect:

```
    attachInterrupt (0, switchPressed, CHANGE);     // that is, for pin D2
    attachInterrupt (digitalPinToInterrupt (2), switchPressed, CHANGE);
```

However the second one is easier to read and more portable to different Arduino types.

Available interrupts

Below is a list of interrupts, in priority order, for the Atmega328:

```
 1  Reset
 2  External Interrupt Request 0  (pin D2)          (INT0_vect)
 3  External Interrupt Request 1  (pin D3)          (INT1_vect)
 4  Pin Change Interrupt Request 0 (pins D8 to D13) (PCINT0_vect)
 5  Pin Change Interrupt Request 1 (pins A0 to A5)  (PCINT1_vect)
 6  Pin Change Interrupt Request 2 (pins D0 to D7)  (PCINT2_vect)
 7  Watchdog Time-out Interrupt                     (WDT_vect)
 8  Timer/Counter2 Compare Match A                  (TIMER2_COMPA_vect)
 9  Timer/Counter2 Compare Match B                  (TIMER2_COMPB_vect)
10  Timer/Counter2 Overflow                         (TIMER2_OVF_vect)
```

```
11  Timer/Counter1 Capture Event           (TIMER1_CAPT_vect)
12  Timer/Counter1 Compare Match A         (TIMER1_COMPA_vect)
13  Timer/Counter1 Compare Match B         (TIMER1_COMPB_vect)
14  Timer/Counter1 Overflow                (TIMER1_OVF_vect)
15  Timer/Counter0 Compare Match A         (TIMER0_COMPA_vect)
16  Timer/Counter0 Compare Match B         (TIMER0_COMPB_vect)
17  Timer/Counter0 Overflow                (TIMER0_OVF_vect)
18  SPI Serial Transfer Complete           (SPI_STC_vect)
19  USART Rx Complete                      (USART_RX_vect)
20  USART, Data Register Empty             (USART_UDRE_vect)
21  USART, Tx Complete                     (USART_TX_vect)
22  ADC Conversion Complete                (ADC_vect)
23  EEPROM Ready                           (EE_READY_vect)
24  Analog Comparator                      (ANALOG_COMP_vect)
25  2-wire Serial Interface  (I2C)         (TWI_vect)
26  Store Program Memory Ready             (SPM_READY_vect)
```

Internal names (which you can use to set up ISR callbacks) are in brackets.

*Warning: If you misspell the interrupt vector name, even by just getting the capitalization wrong (an easy thing to do) the interrupt routine **will not be called**, and you will not get a compiler error.*

Summary of interrupts

The main reasons you might use interrupts are:

- To detect pin changes (eg. rotary encoders, button presses)
- Watchdog timer (eg. if nothing happens after 8 seconds, interrupt me)
- Timer interrupts - used for comparing/overflowing timers
- SPI data transfers
- I2C data transfers
- USART data transfers
- ADC conversions (analog to digital)
- EEPROM ready for use
- Flash memory ready

The "data transfers" can be used to let a program do something else while data is being sent or received on the serial port, SPI port, or I2C port.

Wake the processor

External interrupts, pin-change interrupts, and the watchdog timer interrupt, can also be used to wake the processor up. This can be very handy, as in sleep mode the processor can be configured to use a lot less power (eg. around 10 microamps). A rising, falling, or low-level interrupt can be used to wake up a gadget (eg. if you press a button on it), or a "watchdog timer" interrupt might wake it up periodically (eg. to check the time or temperature).

Pin-change interrupts could be used to wake the processor if a key is pressed on a keypad, or similar.

The processor can also be awoken by a timer interrupt (eg. a timer reaching a certain value, or overflowing) and certain other events, such as an incoming I2C message.

Enabling / disabling interrupts

The "reset" interrupt cannot be disabled. However the other interrupts can be temporarily disabled by clearing the interrupt flag.

Enable interrupts

You can enable interrupts with the function call "interrupts" or "sei" like this:

```
interrupts ();  // or ...
sei ();         // set interrupts flag
```

Disable interrupts

If you need to disable interrupts you can "clear" the interrupt flag like this:

```
noInterrupts ();  // or ...
cli ();           // clear interrupts flag
```

Either method has the same effect, using "interrupts" / "noInterrupts" is a bit easier to remember which way around they are.

The default in the Arduino is for interrupts to be enabled. Don't disable them for long periods or things like timers won't work properly.

Why disable interrupts?

There may be time-critical pieces of code that you don't want interrupted, for example by a timer interrupt.

Also if multi-byte fields are being updated by an ISR then you may need to disable interrupts so that you get the data "atomically". Otherwise one byte may be updated by the ISR while you are reading the other one.

For example:

```
noInterrupts ();
long myCounter = isrCounter;  // get value set by ISR
interrupts ();
```

Temporarily turning off interrupts ensures that isrCounter (a counter set inside an ISR) does not change while we are obtaining its value.

Warning: if you are not sure if interrupts are already on or not, then you need to save the current state and restore it afterwards. For example, the code from the millis() function does this:

```
unsigned long millis()
{
  unsigned long m;
  uint8_t oldSREG = SREG;

  // disable interrupts while we read timer0_millis or we might get an
  // inconsistent value (e.g. in the middle of a write to timer0_millis)
  cli();
  m = timer0_millis;
  SREG = oldSREG;

  return m;
}
```

Note the lines in bold save the current SREG (status register) which includes the interrupt flag. After we have obtained the timer value (which is 4 bytes long) we put the status register back how it was.

> **Tips**
>
> **Function names**
> The functions cli/sei and the register SREG are specific to the AVR processors. If you are using other processors such as the ARM ones the functions may be slightly different.
>
> **Disabling globally vs disabling one interrupt**
> If you use cli() you disable **all** interrupts (including timer interrupts, serial interrupts, etc.).
>
> However if you just want to disable a **particular** interrupt then you should clear the interrupt-enable flag for that particular interrupt source. For example, for external interrupts, call detachInterrupt().

What is interrupt priority?

Since there are 25 interrupts (other than reset) it is possible that more than one interrupt event might occur at once, or at least, occur before the previous one is processed. Also an interrupt event might occur while interrupts are disabled.

The priority order is the sequence in which the processor checks for interrupt events. The higher up the list, the higher the priority. So, for

example, an External Interrupt Request 0 (pin D2) would be serviced before External Interrupt Request 1 (pin D3).

Can interrupts occur while interrupts are disabled?

Interrupts **events** (that is, noticing the event) can occur at any time, and most are remembered by setting an "interrupt event" flag inside the processor. If interrupts are disabled, then that interrupt will be handled when they are enabled again, in priority order.

What are "volatile" variables?

Variables **shared** between ISR functions and normal functions should be declared "volatile". This tells the compiler that such variables might change at any time, and thus the compiler must reload the variable whenever you reference it, rather than relying upon a copy it might have in a processor register.

For example:

```
volatile boolean flag;

// Interrupt Service Routine (ISR)
void isr ()
{
 flag = true;
}  // end of isr

void setup ()
{
  attachInterrupt (digitalPinToInterrupt (2), isr, CHANGE);  // attach interrupt handler
}  // end of setup

void loop ()
{
  if (flag)
    {
    // interrupt has occurred
    }
}  // end of loop
```

How do you use interrupts?

- You write an ISR (interrupt service routine). This is called when the interrupt occurs.
- You tell the processor when you want the interrupt to fire.

Writing an ISR

Interrupt Service Routines are functions with no arguments. Some Arduino libraries are designed to call your own functions, so you just supply an ordinary function (as in the examples above), eg.

```
// Interrupt Service Routine (ISR)
void pinChange ()
{
 flag = true;
}  // end of pinChange
```

However if a library has not already provided a "hook" to an ISR you can make your own, like this:

```
volatile char buf [100];
volatile byte pos;

// SPI interrupt routine
```

```
ISR (SPI_STC_vect)
{
byte c = SPDR;  // grab byte from SPI Data Register

  // add to buffer if room
  if (pos   si eof buf)
    {
    buf [pos  ] = c;
    }  // end of room available
}  // end of interrupt routine SPI_STC_vect
```

In this case you use the "ISR" define, and supply the name of the relevant interrupt vector (from the table earlier on). In this case the ISR is handling an SPI transfer completing. (Note, some old code uses SIGNAL instead of ISR, however SIGNAL is deprecated).

Connecting an ISR to an interrupt

For interrupts already handled by libraries, you just use the documented interface. For example:

```
void receiveEvent (int howMany)
 {
   while (Wire.available ()   0)
     {
     char c = Wire.receive ();
     // do something with the incoming byte
     }
}  // end of receiveEvent

void setup ()
  {
  Wire.onReceive(receiveEvent);
  }
```

In this case the I2C library is designed to handle incoming I2C bytes internally, and then call your supplied function at the end of the incoming data stream. In this case receiveEvent is not strictly an ISR (it has an argument) but it is called by an inbuilt ISR.

Another example is the "external pin" interrupt.

```
// Interrupt Service Routine (ISR)
void pinChange ()
{
   // handle pin change here
}  // end of pinChange

void setup ()
{
   attachInterrupt (digitalPinToInterrupt (2), pinChange, CHANGE);  // attach interrupt handler for D2
}  // end of setup
```

In this case the attachInterrupt function adds the function pinChange to an internal table, and in addition configures the appropriate interrupt flags in the processor.

Configuring the processor to handle an interrupt

The next step, once you have an ISR, is to tell the processor that you want this particular condition to raise an interrupt.

As an example, for External Interrupt 0 (the D2 interrupt) you could do something like this:

```
EICRA  = 3;  // clear existing flags
EICRA  = 2;   // set wanted flags (falling level interrupt)
EIMS   = 1;   // enable it
```

More readable would be to use the defined names, like this:

```
EICRA = (bit(ISC00)  bit (ISC01));  // clear existing flags
EICRA = bit (ISC01);    // set wanted flags (falling level interrupt)
EIMS  = bit (INT0);     // enable it
```

EICRA (External Interrupt Control Register A) would be set according to this table from the Atmega328 datasheet (page 1). That defines the exact type of interrupt you want:

- 0: The low level of INT0 generates an interrupt request (LOW interrupt).
- 1: Any logical change on INT0 generates an interrupt request (CHANGE interrupt).
- 2: The falling edge of INT0 generates an interrupt request (FALLING interrupt).
- 3: The rising edge of INT0 generates an interrupt request (RISING interrupt).

EIMSK (External Interrupt Mask Register) actually enables the interrupt.

Fortunately you don't need to remember those numbers because attachInterrupt does that for you. However that is what is actually happening, and for other interrupts you may have to "manually" set interrupt flags.

Low-level ISRs vs. library ISRs

To simplify your life some common interrupt handlers are actually inside library code (for example INT0_vect and INT1_vect) and then a more user-friendly interface is provided (eg. attachInterrupt). What attachInterrupt actually does is save the address of your wanted interrupt handler into a variable, and then call that from INT0_vect / INT1_vect when needed. It also sets the appropriate register flags to call the handler when required.

Can ISRs be interrupted?

In short, no, not unless you want them to be.

When an ISR is entered, **interrupts are disabled**. Naturally they must have been enabled in the first place, otherwise the ISR would not be entered. However to avoid having an ISR itself be interrupted, the processor turns interrupts off.

When an ISR exits, then **interrupts are enabled again**. The compiler also generates code inside an ISR to save registers and status flags, so that whatever you were doing when the interrupt occurred will not be affected.

However you **can** turn interrupts on inside an ISR if you absolutely must, eg.

```
// Interrupt Service Routine (ISR)
void pinChange ()
{
  // handle pin change here
  interrupts ();  // allow more interrupts

}  // end of pinChange
```

Normally you would need a pretty good reason to do this, as another interrupt now could result in a recursive call to pinChange, with quite possibly undesirable results.

How long does it take to execute an ISR?

According to the datasheet, the minimal amount of time to service an interrupt is 4 clock cycles (to push the current program counter onto the stack) followed by the code now executing at the interrupt vector location. This normally contains a jump to where the interrupt routine really is, which is another 3 cycles.

Then an ISR routine (declared with the ISR define) does something like this:

```
// SPI interrupt routine
ISR (SPI_STC_vect)
 118  1f 92       push  r1  (2)    // save R1 - the " ero" register
```

```
11a  0f 92        push  r0   (2)    // save register R0
11c  0f b6        in  r0, 0x3f (1)  // get SREG (status register)
11e  0f 92        push  r0   (2)    // save SREG
120  11 24        eor r1, r1 (1)    // ensure R1 is  ero
122  8f 93        push  r24  (2)
124  9f 93        push  r25  (2)
126  ef 93        push  r30  (2)
128  ff 93        push  r31  (2)
{
```

That's another 1   cycles (the cycles are in brackets). So from the moment the interrupt occurs, to the first line of code being executed, would be 1   cycles (23 cycles), at   2.5 nS per clock cycle, that would be 1.43 5   S. That's assuming a 1   MH clock.

Then to leave the ISR we have this code:

```
}  // end of interrupt routine SPI_STC_vect
152  ff 91        pop r31 (2)
154  ef 91        pop r30 (2)
156  9f 91        pop r25 (2)
158  8f 91        pop r24 (2)
15a  0f 90        pop r0 (2)     // get old SREG
15c  0f be        out 0x3f, r0 (1)  // restore SREG
15e  0f 90        pop r0 (2)     // now put old R0 register bac
160  1f 90        pop r1 (2)     // restore old value of R1
162  18 95        reti (4)       // return from interrupt, turn interrupts bac   on
```

That's another 1   clock cycles (1.18 5   S). So in total, an ISR using the ISR define will take you 2. 25   S to execute, plus whatever the code itself does.

However the external interrupts (where you use attachInterrupt) do a bit more, as follows:

```
SIGNAL(INT0_vect) {
182  1f 92        push  r1   (2)
184  0f 92        push  r0  (2)
186  0f b6        in  r0, 0x3f (1)
188  0f 92        push  r0  (2)
18a  11 24        eor r1, r1 (1)
18c  2f 93        push  r18 (2)
18e  3f 93        push  r19 (2)
190  4f 93        push  r20 (2)
192  5f 93        push  r21 (2)
194  6f 93        push  r22 (2)
196  7f 93        push  r23 (2)
198  8f 93        push  r24 (2)
19a  9f 93        push  r25 (2)
19c  af 93        push  r26 (2)
19e  bf 93        push  r27 (2)
1a0  ef 93        push  r30 (2)
1a2  ff 93        push  r31 (2)
 if(intFunc[EXTERNAL_INT_0])
1a4  80 91 00 01  lds r24, 0x0100 (2)
1a8  90 91 01 01  lds r25, 0x0101 (2)
1ac  89 2b        or  r24, r25 (2)
1ae  29 f0        breq  . 10   (2)
   intFunc[EXTERNAL_INT_0]();
1b0  e0 91 00 01  lds r30, 0x0100 (2)
1b4  f0 91 01 01  lds r31, 0x0101 (2)
1b8  09 95        icall (3)
}
1ba  ff 91        pop r31 (2)
1bc  ef 91        pop r30 (2)
1be  bf 91        pop r27 (2)
1c0  af 91        pop r26 (2)
1c2  9f 91        pop r25 (2)
1c4  8f 91        pop r24 (2)
1c6  7f 91        pop r23 (2)
1c8  6f 91        pop r22 (2)
1ca  5f 91        pop r21 (2)
1cc  4f 91        pop r20 (2)
```

```
1ce  3f 91        pop r19 (2)
1d0  2f 91        pop r18 (2)
1d2  0f 90        pop r0 (2)
1d4  0f be        out 0x3f, r0 (1)
1d6  0f 90        pop r0 (2)
1d8  1f 90        pop r1 (2)
1da  18 95        reti (4)
```

I count 82 cycles there (5.125 S in total at 1 MH) as overhead plus whatever is actually done in the supplied interrupt routine. That is, 2. 3 5 S **before** entering your interrupt handler, and another 2.18 5 S after it returns.

How long before the processor starts entering an ISR?

This various somewhat. The figures quoted above are the ideal ones where the interrupt is immediately processed. A few factors may delay that:

- If the processor is asleep, there are designated "wake-up" times, which could be quite a few milliseconds, while the clock is spooled back up to speed. This time would depend on fuse settings, and how deep the sleep is.

- If an interrupt service routine is already executing, then further interrupts cannot be entered until it either finishes, or enables interrupts itself. This is why you should keep each interrupt service routine short, as every microsecond you spend in one, you are potentially delaying the execution of another one.

- Some code turns interrupts off. For example, calling millis() briefly turns interrupts off. Therefore the time for an interrupt to be serviced would be extended by the length of time interrupts were turned off.

- Interrupts can only be serviced at the end of an instruction, so if a particular instruction takes three clock cycles, and has just started, then the interrupt will be delayed at least a couple of clock cycles.

- An event that turns interrupts back on (eg. returning from an interrupt service routine) is guaranteed to execute at least one more instruction. So even if an ISR ends, and your interrupt is pending, it still has to wait for one more instruction before it is serviced.

- Since interrupts have a priority, a higher-priority interrupt might be serviced before the interrupt you are interested in.

Performance considerations

Interrupts can increase performance in many situations because you can get on with the "main work" of your program without having to constantly be testing to see if switches have been pressed. Having said that, the overhead of servicing an interrupt, as discussed above, would actually be more than doing a "tight loop" polling a single input port. If you absolutely need to respond to an event within, say, a microsecond, then an interrupt would be too slow. In that case you might disable interrupts (eg. timers) and just loop looking for the pin to change.

How are interrupts queued?

There are two sorts of interrupts:

- Some set a flag and they are handled in priority order, even if the event that caused them has stopped. For example, a rising, falling, or changing level interrupt on pin D2.

- Others are only tested if they are happening "right now". For example, a low-level interrupt on pin D2.

The ones that set a flag could be regarded as being queued, as the interrupt flag remains set until such time as the interrupt routine is entered, at which time the processor clears the flag. Of course, since there is only one flag, if the same interrupt condition occurs again before the first one is processed, it won't be serviced twice.

Something to be aware of is that these flags can be set before you attach the interrupt handler. For example, it is possible for a rising or falling level interrupt on pin D2 to be "flagged", and then as soon as you do an attachInterrupt the interrupt immediately fires, even if the event occurred an hour ago. To avoid this you can manually clear the flag. For example:

```
EIFR = 1;  // clear flag for interrupt 0 (D2 on Uno)
EIFR = 2;  // clear flag for interrupt 1 (D3 on Uno)
```

Or, for readability:

```
EIFR = bit (INTF0);  // clear flag for interrupt 0
EIFR = bit (INTF1);  // clear flag for interrupt 1
```

However the "low level" interrupts are continuously checked, so if you are not careful they will keep firing, even after the interrupt has been called. That is, the ISR will exit, and then the interrupt will immediately fire again. To avoid this, you should do a detachInterrupt immediately after you know that the interrupt fired. An example of this is going into sleep mode:

```
 include  avr/sleep.h

// interrupt service routine in sleep mode
void wa e ()
{
  sleep_disable ();         // first thing after wa ing from sleep
}  // end of wa e

void sleepNow ()
{
  set_sleep_mode (SLEEP_MODE_PWR_DOWN);
  sleep_enable ();          // enables the sleep bit in the mcucr register
  attachInterrupt (digitalPinToInterrupt (2), wa e, LOW);  // wa e up on low level on D2
  sleep_mode ();            // here the device is actually put to sleep
  detachInterrupt (digitalPinToInterrupt (2));     // stop LOW interrupt on D2
}  // end of sleepNow
```

Note that the interrupt is detached immediately after we wake up. However we should be cautious about putting the detachInterrupt actually inside the "wake" ISR, because it is possible the interrupt might fire between attaching the interrupt and going to sleep, in which case we would never wake up.

[EDIT] (14th uly 2013)

Correction:

The improved version below solves this problem by disabling interrupts before doing the attachInterrupt call. Since you are guaranteed that one instruction will be executed after re-enabling interrupts, we are sure that the sleep_cpu call will be done before the interrupt occurs.

```
 include  avr/sleep.h

// interrupt service routine in sleep mode
void wa e ()
{
  sleep_disable ();         // first thing after wa ing from sleep
  detachInterrupt (digitalPinToInterrupt (2));     // stop LOW interrupt on D2
}  // end of wa e

void sleepNow ()
{
  set_sleep_mode (SLEEP_MODE_PWR_DOWN);
  noInterrupts ();          // ma e sure we don t get interrupted before we sleep
  sleep_enable ();          // enables the sleep bit in the mcucr register
  attachInterrupt (digitalPinToInterrupt (2), wa e, LOW);  // wa e up on low level on D2
  interrupts ();            // interrupts allowed now, next instruction WILL be executed
  sleep_cpu ();             // here the device is put to sleep
}  // end of sleepNow
```

These examples illustrates how you can save power with a LOW interrupt, because the LOW interrupt is activated even if the processor is asleep ~~(whereas the other ones - RISING, FALLING, CHANGE - are not)~~. (*Amended 15 Dec 2013* : We now believe all four external interrupt types will wake the processor).

Tip:

> I received this confirmation from Atmel that the datasheet is wrong about only LOW interrupts waking the processor.

> **Quote:**
>
> Commented by Manoraj Gnanadhas (Atmel)
> 2015-01-20 0 :23:3 GMT
> Recipients: Nick Gammon
>
> Hello Nick,
>
> Our design team has confirmed that Note-3 mentioned under Table 10-1 is a datasheet bug. So you can use any type of interrupt (Rising edge/ Falling edge / Low level / Any logical change) to wake up from sleep mode. Sorry for the inconvenience caused.
>
> Best Regards,
> Manoraj Gnanadhas

> Thus, all interrupt types will wake the processor.

Hints for writing ISRs

In brief, keep them short While an ISR is executing other interrupts cannot be processed. So you could easily miss button presses, or incoming serial communications, if you try to do too much. In particular, you should not try to do debugging "prints" inside an ISR. The time taken to do those is likely to cause more problems than they solve.

A reasonable thing to do is set a single-byte flag, and then test that flag in the main loop function. Or, store an incoming byte from a serial port into a buffer. The inbuilt timer interrupts keep track of elapsed time by firing every time the internal timer overflows, and thus you can work out elapsed time by knowing how many times the timer overflowed.

Remember, inside an ISR interrupts are disabled. Thus hoping that the time returned by millis() function calls will change, will lead to disappointment. It is valid to **obtain** the time that way, just be aware that the timer is not incrementing. And if you spend too long in the ISR then the timer may miss an overflow event, leading to the time returned by millis() becoming incorrect.

A test shows that, on a 1 MH Atmega328 processor, a call to micros() takes 3.5 25   S. A call to millis() takes 1. 3 5   S. Recording (saving) the current timer value is a reasonable thing to do in an ISR. Finding the elapsed milliseconds is faster than the elapsed microseconds (the millisecond count is just retrieved from a variable). However the microsecond count is obtained by adding the current value of the Timer 0 timer (which will keep incrementing) to a saved "Timer 0 overflow count".

Warning: Since interrupts are disabled inside an ISR, and since the latest version of the Arduino IDE uses interrupts for Serial reading and writing, and also for incrementing the counter used by "millis" and "delay" you should not attempt to use those functions inside an ISR. To put it another way:

- Don't attempt to delay, eg: `delay (100);`
- You can get the time from a call to millis, however it won't increment, so don't attempt to delay by waiting for it to increase.
- Don't do serial prints (eg. `Serial.println ("ISR entered"); )`
- Don't try to do serial reading.

Pin change interrupts

There are two ways you can detect external events on pins. The first is the special "external interrupt" pins, D2 and D3. These general discrete interrupt events, one per pin. You can get to those by using attachInterrupt for each pin. You can specify a rising, falling, changing or low-level condition for the interrupt.

However there are also "pin change" interrupts for all pins (on the Atmega328, not necessarily all pins on other processors). These act on groups of pins (D0 to D , D8 to D13, and A0 to A5). They are also lower priority than the external event interrupts. You could make an interrupt handler to handle changes on pins D8 to D13 like this:

```
ISR (PCINT0_vect)
 {
  // one of pins D8 to D13 has changed
 }
```

Clearly extra code is needed to work out exactly which pin changed (eg. comparing it to a "before" value).

Pin change interrupts each have an associated "mask" byte in the processor, so you could actually configure them to only react to (say) D8, D10 and D12, rather than a change to D8 to D13. However you still then need to work out which of those changed.

Example of watchdog timer interrupt

```
 include  avr/sleep.h
 include  avr/wdt.h

 define LED 13

// interrupt service routine for when button pressed
void wa e ()
{
  wdt_disable();  // disable watchdog
}  // end of wa e

// watchdog interrupt
ISR (WDT_vect)
{
  wa e ();
}  // end of WDT_vect

void myWatchdogEnable (const byte interval)
{
  noInterrupts ();   // timed sequence below

  MCUSR = 0;                        // reset various flags
  WDTCSR  = 0b00011000;             // see docs, set WDCE, WDE
  WDTCSR = 0b01000000   interval;   // set WDIE, and appropriate delay
  wdt_reset();

  byte adcsra_save = ADCSRA;
  ADCSRA = 0;  // disable ADC
  power_all_disable ();   // turn off all modules
  set_sleep_mode (SLEEP_MODE_PWR_DOWN);   // sleep mode is set here
  sleep_enable();
  attachInterrupt (digitalPinToInterrupt (2), wa e, LOW);   // allow grounding pin D2 to wa e us
  interrupts ();
  sleep_cpu ();            // now goes to Sleep and waits for the interrupt
  detachInterrupt (digitalPinToInterrupt (2));     // stop LOW interrupt on pin D2

  ADCSRA = adcsra_save;  // stop power reduction
  power_all_enable ();   // turn on all modules
}  // end of myWatchdogEnable

void setup ()
{
  digitalWrite (2, HIGH);    // pull-up on button
}  // end of setup

void loop()
{
  pinMode (LED, OUTPUT);
  digitalWrite (LED, HIGH);
  delay (5000);
  digitalWrite (LED, LOW);
  delay (5000);

  // sleep bit patterns
  //  1 second   0b000110
  //  2 seconds  0b000111
  //  4 seconds  0b100000
  //  8 seconds  0b100001

  // sleep for 8 seconds
  myWatchdogEnable (0b100001);  // 8 seconds

}  // end of loop
```

The above code, running on a "bare bones" board (that is, without voltage regulator, USB interface etc.) uses the following:

- With LED lit: 1 .5 mA
- Awake, with LED off: 1 .5 mA

- Asleep: uA (0.00 mA)

You can see that using the watchdog timer combined with sleep mode, you can save a considerable amount of power during times when the processor might not be needed.

It also illustrates how you can recover from sleep in two different ways. One is with a button press (ie. you ground pin D2), the other is waking up periodically (every 8 seconds), although you could make it every 1, 2 4, or 8 seconds (or even shorter if you consult the data sheet).

Warning: after being woken the processor might need a short time to stabili e its clock. For example, you may see "garbage" on a serial comms port while things are synchroni ing. If this is a problem you may want to build in a short delay after being woken.

Note re brownout detection: The figures measured above are with brownout detection disabled. To generate a reference voltage for the brownout detection takes a bit of current. With it enabled, the sleep mode uses closer to 0 uA (not uA). One way of turning off the brownout detection is to use avrdude from the command line, like this:

```
avrdude -c usbtiny -p m328p -U efuse w 0x07 m
```

That sets the "efuse" (extended fuse) to be , which disables brownout detection. This example assumes you are using the USBtinyISP programmer board.

You are probably using interrupts anyway ...

A "normal" Arduino environment already is using interrupts, even if you don't personally attempt to. The millis() and micros() function calls make use of the "timer overflow" feature. One of the internal timers (timer 0) is set up to interrupt roughly 1000 times a second, and increment an internal counter which effectively becomes the millis() counter. There is a bit more to it than that, as adjustment is made for the exact clock speed.

Also the hardware serial library uses interrupts to handle incoming serial data (and in the more recent library versions, outgoing serial data as well). This is very useful as your program can be doing other things while the interrupts are firing, and filling up an internal buffer. Then when you check Serial.available() you can find out what, if anything, has been placed in that buffer.

Executing the next instruction after enabling interrupts

After a bit of discussion and research on the Arduino forum, we have clarified exactly what happens after you enable interrupts. There are three main ways I can think of that you can enable interrupts, which were previously not enabled:

```
sei ();  // set interrupt enable flag
SREG = 0x80;  // set the high-order bit in the status register
reti  ;   // assembler instruction "return from interrupt"
```

In all cases, the processor guarantees that the **next** instruction after interrupts are enabled (if they were previously disabled) will always be executed, even if an interrupt event is pending. (By "next" I mean the next one in program sequence, not necessarily the one physically following. For example, a RETI instruction jumps back to where the interrupt occurred, and then executes one more instruction).

This lets you write code like this:

```
sei ();
sleep_cpu ();
```

If not for this guarantee, the interrupt might occur **before** the processor slept, and then it might never be awoken.

Empty interrupts

If you merely want an interrupt to wake the processor, but not do anything in particular, you can use the EMPTY_INTERRUPT define, eg.

```
EMPTY_INTERRUPT (PCINT1_vect);
```

This simply generates a "reti" (return from interrupt) instruction. Since it doesn't try to save or restore registers this would be the fastest way to get an interrupt to wake it up.

Read the data sheet

More information about interrupts, timers, etc. can be obtained from the data sheet for the processor.

http //atmel.com/dyn/resources/prod_documents/8271S.pdf

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Nick Gammon** Australia (21,241 posts) bio *Forum Administrator*

Date Reply 1 on Mon 09 Jan 2012 08:03 AM (UTC)

Amended on Tue 25 Oct 2016 06:41 AM (UTC) by Nick Gammon

Message

Example code of a pump timer

This was posted on the Arduino forum (by me) of an example of how to make a pump timer. The hardware consisted of a switch (the on/cancel switch) connected to D2 and a "pump time" rotary switch (not a rotary encoder) connected to pins 3 to 11. For example, if the switch was in the 3rd position you would get the pump running for 10 minutes.

The pump was connected to pin 13 via a transistor/relay.

```
// Pump timer
// Author  Nic  Gammon
// Date  7th  anuary 2012
// Released into the public domain.

 include  avr/sleep.h

const int pumpPin = 13;       // output pin for the pump and solenoid (goes to the relay)
const int buttonPin = 2;      // input pin (for a pushbutton switch)
const int firstSwitch = 3;    // first switch pin for time rotary button
const int lastSwitch = 11;    // last switch pin
const int debounceTime = 20;  // debounce in milliseconds
const unsigned long delayTime [] = { 2, 5, 10, 15, 20, 30, 45, 46, 120};   // Pump run times in minutes
unsigned long startPumpTime = 0;         // when pump started
unsigned long pumpTime;                  // how long to run pump
volatile boolean buttonPressed;     // set when button pressed

// interrupt service routine in sleep mode
void wa e ()
{
  sleep_disable ();        // first thing after wa ing from sleep
}  // end of wa e

// interrupt service routine when awa e and button pressed
void buttonDown ()
{
  buttonPressed = true;
}  // end of buttonDown

void sleepNow ()
{
  set_sleep_mode (SLEEP_MODE_PWR_DOWN);
  sleep_enable ();         // enables the sleep bit in the mcucr register
  attachInterrupt (digitalPinToInterrupt (buttonPin), wa e, LOW);
  sleep_mode ();           // here the device is actually put to sleep
```

```
      detachInterrupt (digitalPinToInterrupt (buttonPin));     // stop LOW interrupt
}  // end of sleepNow

void deBounce ()
{
  unsigned long now = millis ();
  do
  {
    // on bounce, reset time-out
    if (digitalRead (buttonPin) == LOW)
      now = millis ();
  }
  while (digitalRead (buttonPin) == LOW
    (millis () - now)  = debounceTime);

}  // end of deBounce

void setup()
{
  pinMode(pumpPin, OUTPUT);
  digitalWrite (buttonPin, HIGH);     // pull-up on button
  for (int i = firstSwitch; i  = lastSwitch; i  )
    digitalWrite (i, HIGH);            // pull-up on switch
} // end of setup

void loop ()
{

  // if pump is running, see if time to turn it off
  if (digitalRead (pumpPin) == HIGH)
  {
    if ((millis () - startPumpTime)  = pumpTime    buttonPressed)
    {
      digitalWrite (pumpPin, LOW);
      deBounce ();
      buttonPressed = false;
    }
    return;  // not time to sleep yet
  }  // end of pump running


  // ------ here if pump not running -----

  // pump not running  sleep then
  sleepNow ();

  // pump is not running and we wo e up, wor  out how long to run it

  deBounce ();
  pumpTime = 0;

  EIFR = 1;      // cancel any existing falling interrupt (interrupt 0)
  attachInterrupt (digitalPinToInterrupt (buttonPin), buttonDown, FALLING);   // ready for button press

  for (int i = firstSwitch; i  = lastSwitch; i  )
    if (digitalRead (i) == LOW)
      pumpTime = delayTime [i - firstSwitch];

  if (pumpTime == 0)
    return;  // no time selected

  // start pump
  startPumpTime = millis ();
  digitalWrite (pumpPin, HIGH);

  pumpTime  = 60000UL;  // convert minutes to milliseconds

  // pumpTime = 5000;  // FOR TESTING - 5 seconds

}  // end of loop
```
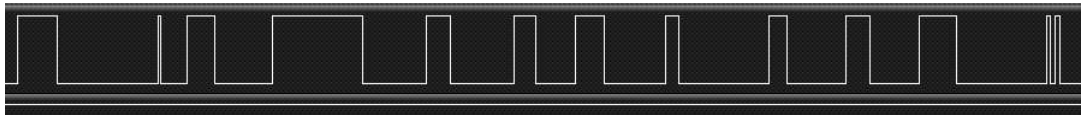
Of interest is the deBounce function - that handles the inevitable "bounces" of switch contacts as you open or close them. Without some sort of debouncing handler you might interpret a single button press as 20 presses, making it very hard to actually do anything.

Above is an image of a switch bouncing, captured on the logic analy er. You can see from the above, that a simple keypress might result in a do en or so transitions. On the image each bounce is around 5 mS apart. So we really need to wait for a longer interval to elapse, in which the switch doesn't open/close again.

The debounce handler above waits for 20 milliseconds for the switch to stop bouncing, and if the switch closes again, resets the time interval so it waits another 20 milliseconds. This seemed to work quite well.

The code also demonstrates putting the processor to sleep if it isn't needed (eg. at midnight) so it uses less power. It also shows how the "low" interrupt can be used to wake it up, and the falling interrupt to notice if the switch is pressed while the pump is running (eg., you have watered your plants enough already).

- Nick Gammon

www.gammon.com.au, www.mushclient.com

Posted by [Nick Gammon](#) Australia (21,241 posts) [bio](#) *Forum Administrator*

Date [Reply   2](#) on Tue 10 Jan 2012 03:20 AM (UTC)

Amended on ed 04 Sep 2013 04:33 AM (UTC) by [Nick Gammon](#)

Message

Timer interrupts

The "normal" Arduino IDE uses timer 0 to provide a "clock" being a number that is incremented, and adjusted, to give you a count per millisecond. You can read that by doing something like this:

```
unsigned long nowMs = millis ();
```

You can also find a more accurate timer value by using micros (), which takes the current value from the millis counter, and adds in the current reading from timer 0 (thus correcting for "right now"). eg.

```
unsigned long nowUs = micros ();
```

However you can make your own interrupts, say using Timer 1, like this:

```
const byte LED = 13;

ISR(TIMER1_COMPA_vect)
{
static boolean state = false;
  state =  state;  // toggle
  digitalWrite (LED, state   HIGH   LOW);
}

void setup() {
  pinMode (LED, OUTPUT);

  // set up Timer 1
  TCCR1A = 0;           // normal operation
  TCCR1B = bit(WGM12)   bit(CS10);   // CTC, no pre-scaling
  OCR1A =  999;       // compare A register value (1000   cloc   speed)
  TIMS 1 = bit (OCIE1A);            // interrupt on Compare A Match
}  // end of setup

void loop() { }
```

This uses CTC (Clear Timer on Compare), so it counts up to the specified value, clears the timer, and starts again. This toggles pin D13

faster than you can see (once every 2.5 S). (It turns <u>on</u> every second toggle, that is every 125 S, giving a frequency of 1/0.000125 8 KH).

Note: The counter is *zero-relative*. So if the counter is 1, then it actually counts two times the clock_speed/pre-scaler (0 and 1). So for an interval of 1000 slower than the internal clock, we need to set the counter to .

If you change the prescaler it toggles every 4 mS, slow enough for you to see it flashing. (That is, the LED lights every 128 mS, being 1/0.128 .8125 H).

```
const byte LED = 13;

ISR(TIMER1_COMPA_vect)
{
static boolean state = false;
  state =  state;  // toggle
  digitalWrite (LED, state   HIGH   LOW);
}

void setup() {
  pinMode (LED, OUTPUT);

  // set up Timer 1
  TCCR1A = 0;           // normal operation
  TCCR1B = bit(WGM12) | bit(CS10) | bit (CS12);   // CTC, scale to clock / 1024
  OCR1A = 999;          // compare A register value (1000   cloc   speed / 1024)
  TIMS 1 = bit (OCIE1A);             // interrupt on Compare A Match
} // end of setup

void loop() { }
```

And even more simply, you can output a timer result without even using interrupts:

```
const byte LED = 9;

void setup() {
  pinMode (LED, OUTPUT);

  // set up Timer 1
  TCCR1A = bit (COM1A0);  // toggle OC1A on Compare Match
  TCCR1B = bit(WGM12)   bit(CS10)   bit (CS12);   // CTC, scale to cloc  / 1024
  OCR1A =  4999;         // compare A register value (5000   cloc   speed / 1024)
}  // end of setup

void loop() { }
```

Plug an LED (and resistor in series, say 4 0 ohms) between D and Gnd, and you will see it toggle every 320 mS. (1/(1 000000 / 1024) 5000).

More information on timers here:

http //www.avrfrea s.net/index.php name=PNphpBB2 file=viewtopic t=50106

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Nick Gammon** Australia (21,241 posts) bio *Forum Administrator*

Date Reply 3 on Fri 13 Jan 2012 01:5 AM (UTC)

Amended on Sat 18 Oct 2014 09:55 PM (UTC) by Nick Gammon

Message

Ignition timing with timers and interrupts

Below is another example of interrupts and timers from the Arduino forum. The basic problem was to turn something on when an

interrupt occurred (eg. on pin D2) but after a delay. The code below uses an interrupt to start the process, sets a timer for 0.5 mS (as an initial delay). Then when that timer fires, the spark is turned on for a different interval (2 mS) and then when that time is up, the spark is turned off, and the process resumes.

It's an interesting example of pin interrupts, timers and timer interrupts.

```
// Timer with an ISR example
// Author  Nic  Gammon
// Date  13th  anuary 2012
// Modified  19 October 2014

 include  digitalWriteFast.h

const byte FIRE_SENSOR = 2;  // note this is interrupt 0
const byte SPAR PLUG = 9;

volatile unsigned int spar DelayTime = 500;    // microseconds
volatile unsigned int spar OnTime = 2000;      // microseconds

// allow for time ta en to enter ISR (determine empirically)
const unsigned int isrDelayFactor = 4;         // microseconds

// is spar  currently on
volatile boolean spar On;

void activateInterrupt0 ()
  {
  EICRA  =  (bit(ISC00)   bit (ISC01));  // clear existing flags
  EICRA  =  bit (ISC01);    // set wanted flags (falling level interrupt)
  EIFR   =  bit (INTF0);    // clear flag for interrupt 0
  EIMS   =  bit (INT0);     // enable it
  }  // end of activateInterrupt0

void deactivateInterrupt0 ()
  {
  EIMS   =  bit (INT0);     // disable it
  }  // end of deactivateInterrupt0

// interrupt for when time to turn spar  on then off again
ISR (TIMER1_COMPA_vect)
  {

  // if currently on, turn off
  if (spar On)
    {
    digitalWriteFast (SPAR PLUG, LOW);  // spar  off
    TCCR1B = 0;                         // stop timer
    TIMS 1 = 0;                         // cancel timer interrupt
    activateInterrupt0 ();              // re-instate interrupts for firing time
    }
  else
    // hold-off time must be up
    {
    digitalWriteFast (SPAR PLUG, HIGH); // spar  on
    TCCR1B = 0;                         // stop timer
    TCNT1 = 0;                          // count bac  to  ero
    TCCR1B = bit(WGM12)   bit(CS11);    // CTC, scale to cloc  / 8
    // time before timer fires ( ero relative)
    // multiply by two because we are on a prescaler of 8
    OCR1A = (spar OnTime   2) - (isrDelayFactor   2) - 1;
    }

  spar On =  spar On;                   // toggle

  }  // end of TIMER1_COMPA_vect

// ISR for when to fire
ISR (INT0_vect)
  {
  spar On = false;                   // ma e sure flag off  ust in case

  // set up Timer 1
  TCCR1A = 0;                        // normal mode
  TCNT1 = 0;                         // count bac  to  ero
  TCCR1B = bit(WGM12)   bit(CS11);   // CTC, scale to cloc  / 8
  // time before timer fires -  ero relative
  // multiply by two because we are on a prescaler of 8
```

```
    OCR1A = (spar DelayTime   2) - (isrDelayFactor   2) - 1;
    TIMS 1 = bit (OCIE1A);              // interrupt on Compare A Match

    deactivateInterrupt0 ();            // no more interrupts yet
  } // end of ISR (INT0_vect)

void setup()
  {
  TCCR1A = 0;  // normal mode
  TCCR1B = 0;  // stop timer
  TIMS 1 = 0;  // cancel timer interrupt

  pinMode (SPAR PLUG, OUTPUT);
  pinMode (FIRE_SENSOR, INPUT_PULLUP);

  activateInterrupt0 ();
  }  // end of setup

void loop()
  {
  // read sensors, compute time to fire spar

  if (false)  // if we need to change the time, insert condition here ...
    {
    noInterrupts ();        // atomic change of the time amount
    spar DelayTime = 500;   // delay before spar  in microseconds
    spar OnTime = 2000;     // spar  on time in microseconds
    interrupts ();
    }
  }  // end of loop
```
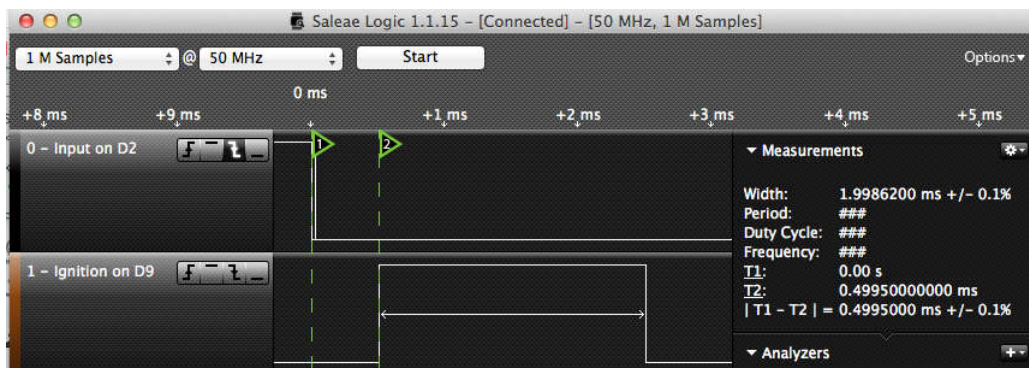
The graphic shows (ignore the switch bounce) that after the switch is closed there is a delay of 500   S (between the green flags) and then a delay of 1   S while the spark is "on".

[EDIT] Changes made on 1 th October 2014.

The sketch uses the digitalwritefast library to do quick writes to the spark pin. It also uses the ISR (INT0_vect) function rather than attachInterrupt, because attachInterrupt takes longer to run.

https://code.google.com/p/digitalwritefast/

There is a tweaking figure of 4 S (isrDelayFactor) which you can determine empirically (by measuring) to compensate for the few microseconds that it takes to enter the ISR. By subtracting that from the required timer interval, the timing is more accurate.

- Nick Gammon

www.gammon.com.au, www.mushclient.com

Posted by **Nick Gammon** Australia (21,241 posts) bio *Forum Administrator*

Date Reply   4 on Sat 14 Jan 2012 09:02 PM (UTC)

Amended on Tue 25 Oct 2016 06:42 AM (UTC) by Nick Gammon

Message

Camera shutter speed example

This example detects how long a camera shutter is open by using a change interrupt. At the first transition it gets the time, and at the second one it gets the new time.

Then the main loop shows the difference.

Of course this general concept could be applied to anything where you want to time a brief pulse. Tested down to a 50 S pulse, but it could probably go a bit shorter, as it takes around 5 S to enter and leave an ISR.

```cpp
// Camera shutter speed timer
// Author  Nic  Gammon
// Date  15th  anuary 2012

volatile boolean started;
volatile unsigned long startTime;
volatile unsigned long endTime;

// interrupt service routine
void shutter ()
{
  if (started)
    endTime = micros ();
  else
    startTime = micros ();

  started =  started;

} // end of shutter

void setup ()
{
  Serial.begin (115200);
  Serial.println ("Shutter test ...");
  attachInterrupt (digitalPinToInterrupt (2), shutter, CHANGE);
}  // end of setup


void loop ()
{

  if (endTime)
  {
    Serial.print ("Shutter open for ");
    Serial.print (endTime - startTime);
    Serial.println (" microseconds.");
    endTime = 0;
  }

} // end of loop
```

Robot race timer example

Another example uses a laser beam to shine on a LDR (light dependent resistor). The laser is so powerful this simple circuit seemed to work OK:
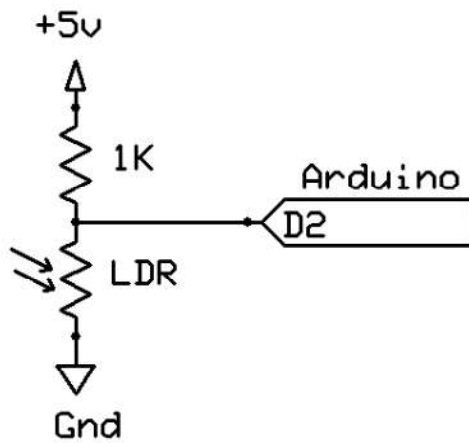
The sketch below times intervals between RISING interrupts (the resistance goes up when less light shines on the LDR):

```
// Robot race timer
// Author  Nic  Gammon
// Date  1st February 2012

unsigned long lastTriggerTime;
volatile unsigned long triggerTime;
volatile boolean triggered;

void isr ()
{
  // wait until we noticed last one
  if (triggered)
    return;

  triggerTime = micros ();
  triggered = true;
}  // end of isr

void setup()
{
  digitalWrite (2, HIGH);  // pull-up
  attachInterrupt(digitalPinToInterrupt (2), isr, RISING);
  Serial.begin (115200);
  Serial.println ("Started timing ...");
}  // end of setup

void loop()
{
  if ( triggered)
    return;

  unsigned long elapsed = triggerTime - lastTriggerTime;

  if (elapsed    1000000L)
    {
    triggered = false;
    return;  // ignore if less than a second
    }

  lastTriggerTime = triggerTime;
  triggered = false;  // re-arm for next time

  Serial.print ("Too    ");
  Serial.print (elapsed);
  Serial.print (" microseconds. ");

  unsigned long minutes, seconds, ms;

  minutes = elapsed / (1000000L    60);
  elapsed -= minutes   (1000000L    60);

  seconds = elapsed / 1000000L;
  elapsed -= seconds    1000000L;

  ms = elapsed / 1000;
```

```
    elapsed -= ms   1000;

    Serial.print (minutes);
    Serial.print ("m ");
    Serial.print (seconds);
    Serial.print ("s ");
    Serial.print (ms);
    Serial.println ("ms.");
}  // end of loop
```

Example output:

```
Started timing ...
Too   3375112 microseconds. 0m 3s 375ms.
Too   1577852 microseconds. 0m 1s 577ms.
Too   1267868 microseconds. 0m 1s 267ms.
Too   1293936 microseconds. 0m 1s 293ms.
Too   2013680 microseconds. 0m 2s 13ms.
Too   3785196 microseconds. 0m 3s 785ms.
Too   12562240 microseconds. 0m 12s 562ms.
```

I shone a cheap 1 mW laser pointer onto the LDR. The resistance is low with the light on it (I read about 1. V on pin D2) which registers as a LOW input. When you break the beam the resistance is high and the pin gets about 3V (being a HIGH). Thus the beam is broken on a RISING interrupt.

The code times the intervals between rising interrupts and reports them to the serial monitor. Of course you could use a LCD or something. There is a test for short intervals (which might be the light passing through a part of the robot) so that if the time is less than a second it doesn't start timing again.

- Nick Gammon

www.gammon.com.au, www.mushclient.com

 top

Posted by **Nick Gammon** Australia (21,241 posts) bio *Forum Administrator*

Date   Reply   5 on   ed 18 Apr 2012 09:00 PM (UTC)

Amended on   ed 04 Sep 2013 04:33 AM (UTC) by Nick Gammon

Message

Read the Analog-to-Digital converter asynchronously

The code example below demonstrates reading the ADC converter (pin A0 in this case) whilst doing something else at the same time.

Each ADC conversion takes around 104 uS, and since that would be 1   4 clock cycles (something like 832 instructions) it would be useful to do something (like process the previous reading) while you are doing it.

```
const byte adcPin = 0;
volatile int adcReading;
volatile boolean adcDone;
boolean adcStarted;

void setup ()
{
  Serial.begin (115200);
  // set the analog reference (high two bits of ADMUX) and select the
  // channel (low 4 bits).  this also sets ADLAR (left-ad ust result)
  // to 0 (the default).
  ADMUX = bit (REFS0)   (adcPin   0x07);

}  // end of setup

// ADC complete ISR
ISR (ADC_vect)
  {
  byte low, high;

  // we have to read ADCL first; doing so loc s both ADCL
```

```
     // and ADCH until ADCH is read.  reading ADCL second would
     // cause the results of each conversion to be discarded,
     // as ADCL and ADCH would be loc ed when it completed.
     low = ADCL;
     high = ADCH;

     adcReading = (high    8)    low;
     adcDone = true;
     }  // end of ADC_vect

void loop ()
{
  // if last reading finished, process it
  if (adcDone)
     {
     adcStarted = false;

     // do something with the reading, for example, print it
     Serial.println (adcReading);
     delay (500);

     adcDone = false;
     }

  // if we aren t ta ing a reading, start a new one
  if ( adcStarted)
     {
     adcStarted = true;
     // start the conversion
     ADCSRA  = bit (ADSC)    bit (ADIE);
     }

  // do other stuff here

}  // end of loop
```

Once the reading has started you continue to execute the main loop, checking to see if it completed in the ISR. Once it does you can process it, print it, or whatever.

Sleep during ADC conversion

Another approach is to go to sleep during the reading. In this case, of course, you cannot do other things, but being asleep reduces digital noise inside the chip, leading to a better analog conversion. The code below is similar to the above, but adds in a "SLEEP_MODE_ADC" sleep for a better conversion.

```
// Example of ta ing ADC reading while asleep
// Author  Nic  Gammon
// Date  23  une 2012

 include  avr/sleep.h

const byte adcPin = 0;
volatile int adcReading;
volatile boolean adcDone;
boolean adcStarted;

void setup ()
{
  Serial.begin (115200);
  // set the analog reference (high two bits of ADMUX) and select the
  // channel (low 4 bits).  this also sets ADLAR (left-ad ust result)
  // to 0 (the default).
  ADMUX = bit (REFS0)    (adcPin   0x07);

}  // end of setup

// ADC complete ISR
ISR (ADC_vect)
   {
   byte low, high;

   // we have to read ADCL first; doing so loc s both ADCL
   // and ADCH until ADCH is read.  reading ADCL second would
```

```
    // cause the results of each conversion to be discarded,
    // as ADCL and ADCH would be loc ed when it completed.
    low = ADCL;
    high = ADCH;

    adcReading = (high     8)    low;
    adcDone = true;
    }  // end of ADC_vect

void loop ()
{
  // if last reading finished, process it
  if (adcDone)
    {
    adcStarted = false;

    // do something with the reading, for example, print it
    Serial.println (adcReading);
    delay (500);

    adcDone = false;
    }

  // if we aren t ta ing a reading, start a new one
  if ( adcStarted)
    {
    adcStarted = true;
    // start the conversion
    ADCSRA  = bit (ADSC)   bit (ADIE);

    set_sleep_mode (SLEEP_MODE_ADC);    // sleep during sample
    sleep_mode ();

    }

  // do other stuff here

}  // end of loop
```

Also consider putting a 0.1 uF capacitor between the AREF and GND pins (adjacent to each other on the Uno board) to reduce noise in the analog reference voltage.

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Nick Gammon** Australia (21,241 posts) bio *Forum Administrator*

Date   Reply   6 on Sun 29 Apr 2012 06:42 AM (UTC)

Amended on Sat 30 Aug 2014 05:11 AM (UTC) by Nick Gammon

Message

Pin Change Interrupts

Pin change interrupts let you detect changes to any of the Arduino pins. However they are a bit more fiddly to use than the external interrupts because they are grouped into batches. So if the interrupt fires you have to work out in your own code exactly which pin caused the interrupt.

Example code:

```
ISR (PCINT0_vect)
 {
 // handle pin change interrupt for D8 to D13 here
 }  // end of PCINT0_vect

ISR (PCINT1_vect)
 {
 // handle pin change interrupt for A0 to A5 here
 }  // end of PCINT1_vect

ISR (PCINT2_vect)
 {
 // handle pin change interrupt for D0 to D7 here
```

```
  }  // end of PCINT2_vect


void setup ()
  {
  // pin change interrupt (example for D9)
  PCMS 0  = bit (PCINT1);  // want pin 9
  PCIFR   = bit (PCIF0);   // clear any outstanding interrupts
  PCICR   = bit (PCIE0);   // enable pin change interrupts for D8 to D13
  }
```

To handle a pin change interrupt you need to:

- Specify which pin in the group. This is the PCMSKn variable (where n is 0, 1 or 2 from the table below). You can have interrupts on more than one pin.
- Enable the appropriate group of interrupts (0, 1 or 2)
- Supply an interrupt handler as shown above

Table of pins -   pin change names / masks

```
D0      PCINT16 (PCMS 2 / PCIF2 / PCIE2)
D1      PCINT17 (PCMS 2 / PCIF2 / PCIE2)
D2      PCINT18 (PCMS 2 / PCIF2 / PCIE2)
D3      PCINT19 (PCMS 2 / PCIF2 / PCIE2)
D4      PCINT20 (PCMS 2 / PCIF2 / PCIE2)
D5      PCINT21 (PCMS 2 / PCIF2 / PCIE2)
D6      PCINT22 (PCMS 2 / PCIF2 / PCIE2)
D7      PCINT23 (PCMS 2 / PCIF2 / PCIE2)
D8      PCINT0  (PCMS 0 / PCIF0 / PCIE0)
D9      PCINT1  (PCMS 0 / PCIF0 / PCIE0)
D10     PCINT2  (PCMS 0 / PCIF0 / PCIE0)
D11     PCINT3  (PCMS 0 / PCIF0 / PCIE0)
D12     PCINT4  (PCMS 0 / PCIF0 / PCIE0)
D13     PCINT5  (PCMS 0 / PCIF0 / PCIE0)
A0      PCINT8  (PCMS 1 / PCIF1 / PCIE1)
A1      PCINT9  (PCMS 1 / PCIF1 / PCIE1)
A2      PCINT10 (PCMS 1 / PCIF1 / PCIE1)
A3      PCINT11 (PCMS 1 / PCIF1 / PCIE1)
A4      PCINT12 (PCMS 1 / PCIF1 / PCIE1)
A5      PCINT13 (PCMS 1 / PCIF1 / PCIE1)
```

Interrupt handler processing

The interrupt handler would need to work out which pin caused the interrupt if the mask specifies more than one (eg. if you wanted interrupts on D8/D /D10). To do this you would need to store the previous state of that pin, and work out (by doing a digitalRead or similar) if this particular pin had changed.

Example of waking from sleep with a Pin Change Interrupt

```
  include  avr/sleep.h

const byte LEDLOOP = 8;
const byte LEDWA E = 9;

ISR (PCINT1_vect)
  {
  // handle pin change interrupt for A0 to A5 here

  // toggle LED
  digitalWrite (LEDWA E,  digitalRead (LEDWA E));
```

```
  }  // end of PCINT1_vect

void setup ()
  {
  pinMode (LEDWA E, OUTPUT);
  pinMode (LEDLOOP, OUTPUT);
  digitalWrite (A0, HIGH);  // enable pull-up

  // pin change interrupt
  PCMS 1  = bit (PCINT8);  // want pin A0
  PCIFR   = bit (PCIF1);   // clear any outstanding interrupts
  PCICR   = bit (PCIE1);   // enable pin change interrupts for A0 to A5

  }  // end of setup

void loop ()
{

  set_sleep_mode (SLEEP_MODE_PWR_DOWN);
  sleep_mode ();

  // flash to indicate we got out of sleep
  digitalWrite (LEDLOOP, HIGH);
  delay (100);
  digitalWrite (LEDLOOP, LOW);
  delay (100);

  } // end of loop
```

- Nick Gammon

www.gammon.com.au, www.mushclient.com

Posted by [Nick Gammon](#) Australia (21,241 posts)   bio *Forum Administrator*

Date [Reply](#)   on Fri 01 Jun 2012 11:06 PM (UTC)

Amended on Sat 25 Jul 2015 02:38 AM (UTC) by [Nick Gammon](#)

Message

Critical sections ... accessing volatile variables

There are some subtle issues regarding variables which are shared between interrupt service routines (ISRs) and the main code (that is, the code not in an ISR).

Since an ISR can fire at any time when interrupts are enabled, you need to be cautious about accessing such shared variables, as they may be being updated at the very moment you access them.

First ... when do you use "volatile" variables?

A variable should only be marked volatile if it is used both inside an ISR, and outside one.

- Variables **only** used outside an ISR should **not** be volatile.
- Variables **only** used inside an ISR should **not** be volatile.
- Variables used both inside and outside an ISR **should** be volatile.

eg.

```
  volatile int counter;
```

Marking a variable as volatile tells the compiler to not "cache" the variables contents into a processor register, but always read it from memory, when needed. This may slow down processing, which is why you don't just make every variable volatile, when not needed.

Atomic access

Consider this code:

```
volatile byte count;

ISR (TIMER1_OVF_vect)
  {
  count = 10;
  }

void setup ()
  {
  }

void loop ()
  {
  count  ;
  }
```

The code generated for count   (add 1 to count) is this:

```
 14c   80 91 00 02   lds   r24, 0x0200
 150   8f 5f         subi  r24, 0xFF      ---- problem if interrupt occurs before this is executed
 152   80 93 00 02   sts   0x0200, r24    ---- problem if interrupt occurs before this is executed
```

(Note that it adds 1 by subtracting -1).

There are two danger points here, as marked. If the interrupt fires after the lds (load register 24 with the variable "count"), but before the sts (store back into the variable "count") then the variable might be changed by the ISR (TIMER1_OVF_vect), however **this change is now lost**, because the variable in the register was used instead.

We need to protect the shared variable by turning off interrupts briefly, like this:

```
volatile byte count;

ISR (TIMER1_OVF_vect)
  {
  count = 10;
  }  // end of TIMER1_OVF_vect

void setup ()
  {
  }  // end of setup

void loop ()
  {
  noInterrupts ();
  count  ;
  interrupts ();
  } // end of loop
```

Now the update of "count" is done "atomically" ... that is, it cannot be interrupted.

Multi-byte variables

Let's make "count" a 2-byte variable, and see the other problem:

```
volatile unsigned int count;

ISR (TIMER1_OVF_vect)
```

```
     {
     count  ;
     } // end of TIMER1_OVF_vect

 void setup ()
     {
     pinMode (13, OUTPUT);
     }  // end of setup

 void loop ()
     {
     if (count   20)
        digitalWrite (13, HIGH);
     } // end of loop
```

OK, we are not changing count any more, so is there still a problem? Sadly, yes. Let's look at the generated code for the "if" statement:

```
  172   80 91 10 02   lds    r24, 0x0210
  176   90 91 11 02   lds    r25, 0x0211    ---- problem if interrupt occurs before this is executed
  17a   45 97         sbiw   r24, 0x15
  17c   50 f0         brcs   . 20
```

Imagine that count was 0xFFFF and was about to "wrap around" back to ero. We load 0xFF into one register, but before we load the second 0xFF the variable changes to 0x00. Now we think count is 0x00FF which is neither the value it had before (0xFFFF) or now (0x0000).

So again we have to "protect" the access to the shared variable, like this:

```
 void loop ()
     {
     noInterrupts ();
     if (count   20)
        digitalWrite (13, HIGH);
     interrupts ();
     }  // end of loop
```

What if you are not sure interrupts are on or off?

There is a final "gotcha" here. What if interrupts might be off already? Then turning them back on afterwards is a Bad Idea.

In that case you need to save the processor status register like this:

```
 unsigned int getCount ()
     {
     unsigned int c;
     byte oldSREG = SREG;    // remember if interrupts are on or off

     noInterrupts ();   // turn interrupts off
     c = count;         // access the shared variable
     SREG = oldSREG;    // turn interrupts back on, if they were on before

     return c;          // return copy of shared variable
     }  // end of getCount
```

This "safe" code saves the current status of interrupts, turns them off (they may already be off), gets the shared variable into a temporary variable, turns interrupts back on - if they were on when we entered the function - and then returns the copy of the shared variable.

Summary

Code may "appear to work" even if you don't take the above precautions. That is because the chances of the interrupt occurring at the

exact wrong moment is fairly low (maybe 1 in 100, depending on the code si e). But sooner or later it will happen at the wrong moment, and your code will either crash, or occasionally return the wrong results.

So for reliable code, pay attention to protecting access to shared variables.

- Nick Gammon

www.gammon.com.au, www.mushclient.com

Posted by **Nick Gammon** Australia (21,241 posts) bio *Forum Administrator*

Date Reply 8 on Sat 2 Jul 2013 12:41 AM (UTC)

Message

Interrupt names to pin mappings

The various models of Arduino have somewhat confusing mappings of interrupt numbers to pins, and the correlation between attachInterrupt (0) is not always to INT0.

The table below shows the ways they are mapped in the internal libraries:

Uno			
attachInterrupt	**Name**	**Pin on chip** (PDIP)	**Pin on board**
0	INT0	4	D2
1	INT1	5	D3

Mega2560			
attachInterrupt	**Name**	**Pin on chip** (TQFP)	**Pin on board**
0	INT4	6	D2
1	INT5	7	D3
2	INT0	43	D21
3	INT1	44	D20
4	INT2	45	D19
5	INT3	46	D18

Leonardo			
attachInterrupt	**Name**	**Pin on chip** (TQFP)	**Pin on board**
0	INT0	18	D3
1	INT1	19	D2
2	INT2	20	D0
3	INT3	21	D1
4	INT6	1	D7

Posted by **Nick Gammon** Australia (21,241 posts) bio *Forum Administrator*

Date Reply 9 on Sat 12 Oct 2013 12:50 AM (UTC)

Amended on Sat 12 Oct 2013 05:09 AM (UTC) by Nick Gammon

Message

ATtiny85 sleep mode and wake on pin change

The sketch below illustrates putting the ATtiny85 to sleep, and then waking on a pin-change interrupt on D4 (pin 3 on the chip). The pin is put into input-pullup mode so all you have to do is ground the pin to wake the chip.

Also see http://www.gammon.com.au/forum/?id 114 reply reply for an example of similar code which also wakes up on a watchdog timer event.

```
// ATtiny85 sleep mode, wa e on pin change interrupt demo
// Author  Nic  Gammon
// Date  12 October 2013

// ATMEL ATTINY 25/45/85 / ARDUINO
//
//                    - /-
// Ain0 (D 5) PB5  1      8  Vcc
// Ain3 (D 3) PB3  2      7  PB2 (D 2) Ain1
// Ain2 (D 4) PB4  3      6  PB1 (D 1) pwm1
```

```
//              GND  4      5  PB0 (D 0) pwm0
//                      ----

 include   avr/sleep.h      // Sleep Modes
 include   avr/power.h      // Power management

const byte LED = 3;   // pin 2
const byte SWITCH = 4; // pin 3 / PCINT4

ISR (PCINT0_vect)
 {
 // do something interesting here
 }

void setup ()
  {
  pinMode (LED, OUTPUT);
  pinMode (SWITCH, INPUT);
  digitalWrite (SWITCH, HIGH);  // internal pull-up

  // pin change interrupt (example for D4)
  PCMS    = bit (PCINT4);  // want pin D4 / pin 3
  GIFR    = bit (PCIF);    // clear any outstanding interrupts
  GIMS    = bit (PCIE);    // enable pin change interrupts

  }  // end of setup

void loop ()
  {
  digitalWrite (LED, HIGH);
  delay (500);
  digitalWrite (LED, LOW);
  delay (500);
  goToSleep ();
  }  // end of loop


void goToSleep ()
  {
  set_sleep_mode(SLEEP_MODE_PWR_DOWN);
  ADCSRA = 0;              // turn off ADC
  power_all_disable ();  // power off ADC, Timer 0 and 1, serial interface
  sleep_enable();
  sleep_cpu();
  sleep_disable();
  power_all_enable();     // power everything bac  on
  }  // end of goToSleep
```

With the brownout disable turned off, the above sketch uses 500 nA of current when asleep, as predicted by the datasheet, page 183:

Figure 22-11. Power-down Supply Current vs. V_{CC} (Watchdog Timer Disabled)

POWER-DOWN SUPPLY CURRENT vs. V_{CC}
WATCHDOG TIMER DISABLED
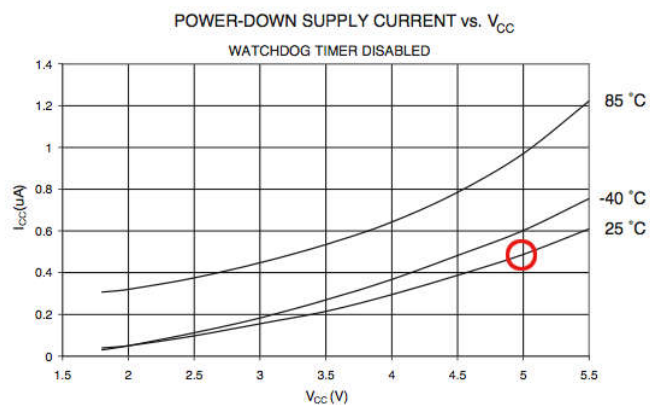
Fuses were:

```
Signature = 0x1E 0x93 0x0B
Processor = ATtiny85
```

```
Flash memory si e = 8192 bytes.
LFuse = 0xE2
HFuse = 0xDF
EFuse = 0xFF
Loc  byte = 0xFF
Cloc  calibration = 0x9A
```

- Nick Gammon

www.gammon.com.au, www.mushclient.com

Posted by **Nick Gammon** Australia (21,241 posts)   bio   *Forum Administrator*

Date Reply   10 on   ed 15 Oct 2014 08:05 PM (UTC)

Amended on Tue 25 Oct 2016 06:43 AM (UTC) by Nick Gammon

Message

Timing an interval

I have seen example code suggesting you count things for a second by turning interrupts off for a second and then on again. I don't recommend this, because for one thing, you can't do serial prints if interrupts are off.

Example (not recommended):

```
volatile unsigned long events;
const unsigned long INTERVAL = 1000;  // 1 second

void eventISR ()
  {
  events  ;
  }   // end of eventISR

void setup ()
  {
  Serial.begin(115200);
  attachInterrupt (digitalPinToInterrupt (2), eventISR, FALLING);
  }   // end of setup

void loop ()
  {
  events = 0;        // reset counter
  interrupts ();     // allow interrupts
  delay (INTERVAL); // wait desired time
  noInterrupts();   // stop interrupts

  Serial.print ("I counted ");
  Serial.println (events);
  } // end of loop
```

That may work for simple situations, but turning interrupts off means that the Serial prints won't work until they are turned on again (which they may not be in a more complex sketch).

It is better to use the millis() result and just detect when the time limit is up.

Improved sketch:

```
volatile bool counting;
volatile unsigned long events;

unsigned long startTime;
const unsigned long INTERVAL = 1000;  // 1 second

void eventISR ()
  {
  if (counting)
    events  ;
  }  // end of eventISR
```

```
void setup ()
  {
  Serial.begin (115200);
  Serial.println ();
  attachInterrupt (digitalPinToInterrupt (2), eventISR, FALLING);
  }  // end of setup

void showResults ()
  {
  Serial.print ("I counted ");
  Serial.println (events);
  }  // end of showResults

void loop ()
  {
  if (counting)
    {
    // is time up
    if (millis () - startTime    INTERVAL)
      return;
    counting = false;
    showResults ();
    }  // end of if

  noInterrupts ();
  events = 0;
  startTime = millis ();
  EIFR = bit (INTF0);  // clear flag for interrupt 0
  counting = true;
  interrupts ();
  }  // end of loop
```

In loop() here we wait for the interval (1 second in this case) to be up, otherwise we return, effectively doing nothing. You could of course do other things instead of just returning.

If the time is up we display the count.

If counting is not currently active (and presuming we want it to be active) we remember the start time, reset the counter to ero, and let it start counting up.

That code seemed to work OK up to 100 kH , although the counts were getting a bit inaccurate. You can do more precise timings by using the hardware counters / timers. Details here:

http://www.gammon.com.au/timers

 top

324,298 iews.

Postings by administrators only.

 Refresh page

Go to topic: [oo   o   ]   [Go]   Search the forum

 top

 Home

Comments to: Gammon Software support
XML Forum RSS feed (https://gammon.com.au/rss/forum. ml)

BEST VIEWED WITH AnyBrowser   FutureQuest