2b. Overview of Interrupts

Jon Zhang edited this page on Nov 23, 2015 · 12 revisions

Some vocabulary:

Registers: PCMSK0, PCMSK1, PCMSK2 :registers that enable or disable pin-change interrupts on individual pins

PCICR : a register where the three least significant bits enable or disable pin change interrupts on a range of pins, i.e. {0,0,0,0,0,PCIE2,PCIE1,PCIE0}, where PCIE2 maps to PCMSK2, PCIE1 maps to PCMSK1, and PCIE0 maps to PCMSK0.

noInterrupts() globally disables interrupts (of all types), while interrupts() globally enables interrupts (of all types), but they will only occur if the requisite registers are set (e.g. PCMSK and PCICR).

What is an interrupt?

An interrupt is a signal that causes the microcontroller to halt execution of the program, and perform a subroutine known as an interrupt handler or Interrupt Service Routine (ISR). After the ISR, program execution continues. This allows the microcontroller to efficiently handle a time-sensitive function such as receiving a burst of data on one of its pins, by not forcing the microcontroller to wait for the data. It can perform other tasks until the interrupt is called.

Physically, interrupts take the form of signals on the pins of the microcontroller. To make the library generic, the dataPin responds to pin change interrupts which are available on all pins of the Arduino Uno, and a majority of pins on other variants. Pin change interrupts trigger an ISR on any change of state detected for a specified pin.

Obviously, interrupts are not always desirable, so they can be controlled by manipulating registers. Registers are small 1-byte (8-bit) stores of memory directly accessible by processor. There is one register PCICR, which can enable and disable pin change interrupts for a range of pins at a single time.

There are also three registers that store the state (enabled/disabled) of the pin change interrupts: PCMSK0, PCMSK1, and PCMSK2. Each bit stores a 1 (enabled) or 0 (disabled).

+ Add a custom sidebar

Clone this wiki locally

`https://github.com/EnviroDIY/A:`

For example, PCMSK0 holds an 8 bits which represent: PCMSK0 {PCINT7, PCINT6, PCINT5, PCINT4, PCINT3, PCINT2, PCINT1, PCINT0}

On an Arduino Uno, these map to: PCMSK0 {XTAL2, XTAL1, Pin 13, Pin 12, Pin 11, Pin 10, Pin 9, Pin 8}

Enabling an interrupt.

Initially, no interrupts are enabled, so PCMSK0 looks like: {00000000}.

If we were to use pin 9 as the data pin, we would set the bit in the pin 9 position to 1, like so: {00000010}.

To accomplish this, we can make use of some predefined macros.

One macro "digitalPinToPCMSK" is defined in "pins_arduino.h" which allows us to quickly get the proper register (PCMSK0, PCMSK1, or PCMSK2) given the number of the Arduino pin. So when we write: digitalPinToPCMSK(9), the address of PCMSK0 is returned. We can use the dereferencing operator '*' to get the value of the address.

That is, *digitalPinToPCMSK(9) returns: {00000000}.

Another macro , "digitalPinToPCMSKbit" is also defined in "pins_arduino.h" and returns the position of the bit of interest within the PCMSK of interest.

So when we write: digitalPinToPCMSKbit(9), the value returned is 1, because pin 9 is represented by the "1st bit" within PCMSK0, which has a zero-based index. That is: PCMSK { 7th bit, 6th bit, 5th bit, 4th bit, 3rd bit, 2nd bit, 1st bit, 0th bit }

The leftward bit shift operator "<<" can then used to create a "mask". Masks are data that are used during bitwise operations to manipulate (enable / disable) one or more individual bits simultaneously.

The syntax for the operator is (variable << number_of_bits).

Some examples:

```
byte a = 5;       // binary: a =
00000101
byte b = a << 4;  // binary: b =
01010000
```

So when we write: (1<<digitalPinToPCMSKbit(9)), we get: {00000010}. Or equivalently: (1<<1), we get: {00000010}.

To use the mask to set the bit of interest we use the bitwise or operator '|'. We will use the compact '|=' notation which does the operation and then stores the result back into the left hand side.

So the operation:

```
*digitalPinToPCMSK(_dataPin) |=
(1<<digitalPinToPCMSKbit(9));
```

Accomplishes:

```
    (1<<digitalPinToPCMSKbit(9))
{00000010}
    PCMSK0
|  {00000000}

-------------

{00000010}
```

We must also enable the global control for the interrupt. This is done in a similar fashion.

```
*digitalPinToPCICR(_dataPin) |=
(1<<digitalPinToPCICRbit(_dataPin));
```

Now let's assume that part of your Arduino sketch outside of SDI-12 had set a pin change interrupt on pin 13. Pin 9 and pin 13 are on the same PCMSK in the case of the Arduino Uno.

This time before we set the bit for pin nine,

```
*digitalPinToPCMSK(9) returns:
{00100000}.
```

So now:

```
    (1<<digitalPinToPCMSKbit(9))
{00000010}
    PCMSK0
|  {00100000}
```

```
-------------

{00100010}
```

By using a bitmask and bitwise operation, we have successfully enabled pin 9 without effecting the state of pin 13.

Disabling an interrupt.

When the we would like to put the SDI-12 object in the DISABLED state, (e.g. the destructor is called), we need to make sure the bit corresponding to the data pin is unset.

Let us consider again the case of where an interrupt has been enabled on pin 13: {00100010}. We want to be sure not to disturb this interrupt when disabling the interrupt on pin 9.

We will make use of similar macros, but this time we will use an inverted bit mask and the AND operation. The "&=" operator is equivalent to a bitwise AND operation between the PCMSK register of interest and the previous result, which is then stored back into the PCMSK of interest. The "*" is the dereferencing operator, which is equivalently translated to "value pointed by", and allows us to store the result back into the proper PCMSK.

Again

```
1<<digitalPinToPCMSKbit(9) returns
```

```
{00000010}
```

The inversion symbol ~ modifies the result to {11111101}

So to finish our example:

```
    ~(1<<digitalPinToPCMSKbit(9))
{11111101}
    PCMSK0
&  {00100010}

-------------

{00100000}
```

So only the interrupt on pin 13 remains set. As a matter of book keeping, if we unset the last bit in the PCMSK, we ought to also unset the respective bit in the PCICR.

```
        !(*digitalPinToPCMSK(9)
                will evaluate TRUE
if PCMSK {00000000}
                will evaluate FALSE
if PCMSK != {00000000}
```

In this case, pin 13 is set, so the expression would be FALSE. If we go back to the original case without pin 13, the expression after disabling pin 9 would evaluate to TRUE.

Therefore if we evaluate to TRUE, we should tidy up:

```


if(!*digitalPinToPCMSK(_dataPin)){
```

```
*digitalPinToPCICR(_dataPin) &= ~
(1<<digitalPinToPCICRbit(_dataPin));
        }
```

+ Add a custom footer