Interrupt

attachInterrupt()

Description

Specifies a named Interrupt Service Routine (ISR) to call when an interrupt occurs. Replaces any previous function that was attached to the interrupt. Most Arduino boards have two external interrupts: numbers 0 (on digital pin 2) and 1 (on digital pin 3). The table below shows the available interrupt pins on various boards.

Board	int.0	int.1	int.2	int.3	int.4	int.5
Uno, Ethernet	2	3				
Mega2560	2	3	21	20	19	18
Leonardo	3	2	0	1	7	
Due			(see below)			

The Arduino Due board has powerful interrupt capabilities that allows you to attach an interrupt function on all available pins. You can directly specify the pin number in attachInterrupt().

Note

Inside the attached function, delay() won't work and the value returned by millis() will not increment. Serial data received while in the function may be lost. You should declare as volatile any variables that you modify within the attached function. See the section on ISRs below for more information.

Using Interrupts

Interrupts are useful for making things happen automatically in microcontroller programs, and can help solve timing problems. Good tasks for using an interrupt may include reading a rotary encoder, or monitoring user input.

If you wanted to insure that a program always caught the pulses from a rotary encoder, so that it never misses a pulse, it would make it very tricky to write a program to do anything else, because the program would need to constantly poll the sensor lines for the encoder, in order to catch pulses when they occurred. Other sensors have a similar interface dynamic too, such as trying to read a sound sensor that is trying to catch a click, or an infrared slot sensor (photo-interrupter) trying to catch a coin drop. In all of these situations, using an interrupt can free the microcontroller to get some other work done while not missing the input.

About Interrupt Service Routines

ISRs are special kinds of functions that have some unique limitations most other functions do not have. An ISR cannot have any parameters, and they shouldn't return anything.

Generally, an ISR should be as short and fast as possible. If your sketch uses multiple ISRs, only one can run at a time, other interrupts will be ignored (turned off) until the current one is finished. as delay() and millis() both rely on interrupts, they will not work while an ISR is running. delayMicroseconds(), which does not rely on interrupts, will work as expected.

Typically global variables are used to pass data between an ISR and the main program. To make sure variables used in an ISR are updated correctly, declare them as `volatile`.

For more information on interrupts, see Nick Gammon's notes.

Syntax

attachInterrupt(interrupt, ISR, mode)

attachInterrupt(pin, ISR, mode) (*Arduino Due only*)

Parameters

interrupt: the number of the interrupt (*int*)

pin: the pin number (*Arduino Due only*)

ISR: the ISR to call when the interrupt occurs; this function must take no parameters and return nothing. This function is sometimes referred to as an *interrupt service routine*.

mode: defines when the interrupt should be triggered. Four contstants are predefined as valid values:

- **LOW** to trigger the interrupt whenever the pin is low,
- **CHANGE** to trigger the interrupt whenever the pin changes value
- **RISING** to trigger when the pin goes from low to high,
- **FALLING** for when the pin

The Due board allows also:

- HIGH to trigger the interrupt whenever the pin is high. (*Arduino Due only*)

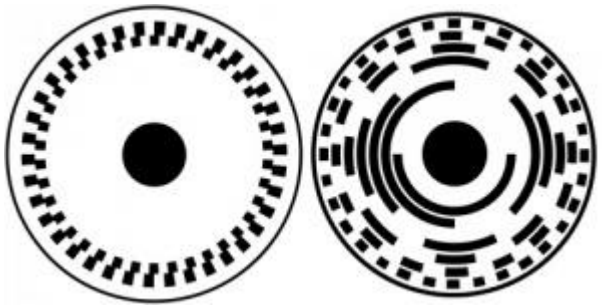
Returns

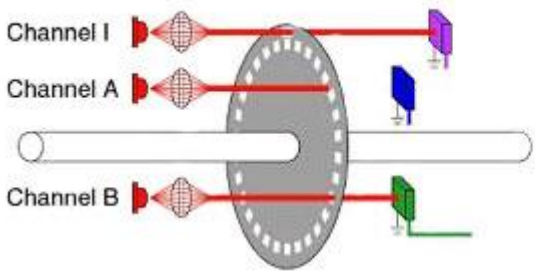
none

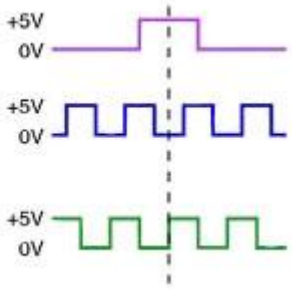
Example

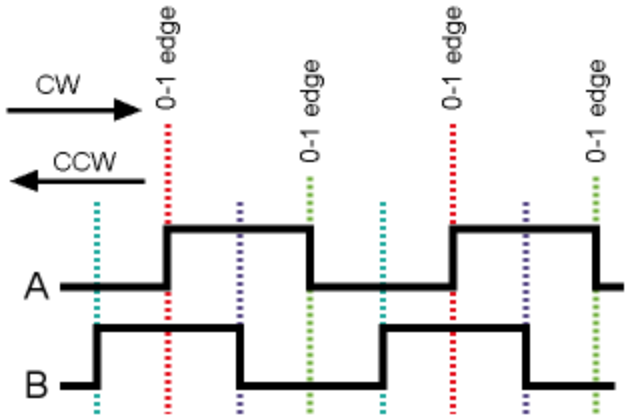
```
int pin = 13;
volatile int state = LOW;

void setup()
{
  pinMode(pin, OUTPUT);
  attachInterrupt(0, blink, CHANGE);
}

void loop()
{
  digitalWrite(pin, state);
}

void blink()
{
  state = !state;
}
```

Encoder

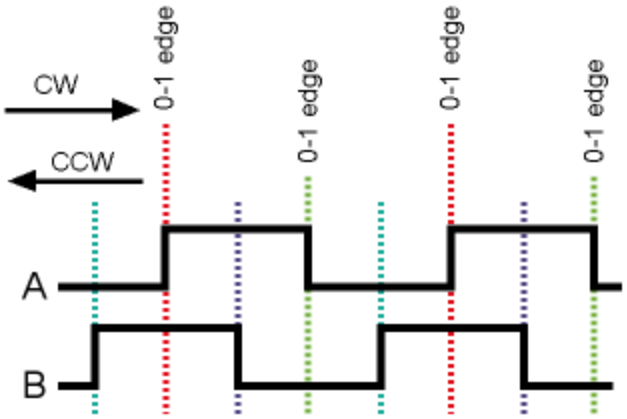



Channel I
Channel A
Channel B


+5V
0V
+5V
0V
+5V
0V


CW
CCW
0-1 edge
0-1 edge
0-1 edge
0-1 edge
A
B

Below is an image showing the waveforms of the A & B channels of an encoder.



This might make it a little more clear how the code above works. When the code finds a low-to-high transition on the A channel, it checks to see if the B channel is high or low and then increments/decrements the variable to account for the direction that the encoder must be turning in order to generate the waveform found.

One disadvantage of the code above is that it is really only counting one fourth of the possible transitions. In the case of the illustration, either the red or the lime green transitions, depending on which way the encoder is moving.