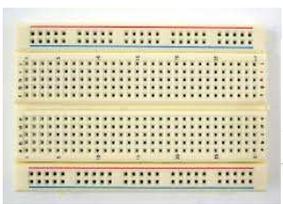
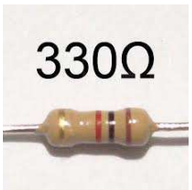
ARDUINO Y LAS INTERRUPCIONES

Gestionando excepciones (Versión 25-8-17)

OBJETIVOS

- Presentar el concepto de **Interrupcion** en Arduino.
- Conocer los tipos de interrupciones.
- Mostrar los **Flags** de disparo.
- Consideraciones generales.

MATERIAL REQUERIDO.

	**Arduino UNO o equivalente**.
 	Una Protoboard más cables.
	Una resistencia de 330Ω.
	Un pulsador.

LAS INTERRUPCIONES

Nunca he sabido muy bien que tiene esto de las **interrupciones**, que hacer temblar a programadores curtidos como si fueran unos novatos.

Recuerdo que en una época en que nos dedicábamos a los microprocesadores (cuando los dinosaurios dominaban la tierra) y comenzábamos a jugar con las interrupciones, había un

porcentaje de técnicos, ingenieros electrónicos o informáticos, que aun comprendiendo la idea de las interrupciones, parecía que su cerebro no podía abarcarlas y sencillamente las ignoraban elegantemente.

Mentarle las interrupciones a muchos Arduineros ya fogueados, en muchos casos supone que recordarán inmediatamente la necesidad de salir urgentemente a hacer algo. Nunca he sabido porque pasa esto, pero vamos a intentar ponerlo remedio inmediatamente.

¿Qué es una **interrupción hardware**?

A un nivel básico, una interrupción es una señal que interrumpe la actividad normal de nuestro microprocesador y salta a atenderla. Hay tres eventos que pueden disparar una interrupción:

- Un evento hardware, previamente definido.
- Un evento programado, o Timer
- Una llamada por software.

Cuando un evento dispara una interrupción, la ejecución normal del micro se suspende (ordenadamente para poder volver) y salta a ejecutar una función especial que llamamos **Interrupt Service Handler o ISH** (Servicio de gestión de interrupción).

Cuando el ISH finaliza, el procesador vuelve tranquilamente al punto donde lo dejó y sigue con lo que estaba como si no hubiese pasado nada.

- *El concepto de interrupción nace de la necesidad imperiosa de reaccionar de forma inmediata en respuesta a un acontecimiento electrónico fulgurante, que no admite demora. Bien sea por la urgencia del suceso o porque algo se podría perder de forma irrecuperable sino reaccionamos con suficiente presteza.*

Pero ¿Qué hay tan urgente que no pueda esperar? ¿Es que nuestros Duinos no son lo suficientemente rápidos para ver cada poco si hay una señal de alarma? ¿Por qué complicarnos la vida con una cosa tan extravagante?

La respuesta como siempre es… depende. Nuestro Arduino puede estar liado y solo leerá la señal de un pin de tanto en tanto. Y si la señal que aparece se desvanece antes de que hayamos ido a ver, ni siquiera lo sabremos, porque aunque los Duinos son rápidos una señal electrónica lo es varios millones de veces más. Este es otro motivo por el que usar delays tiene mucho peligro.

- *En la jerga técnica, a pasar de vez en cuando a ver como está el asunto, se le llama Polling.*

Por otro lado las interrupciones nos ofrecen una ventaja enorme a la hora de organizar nuestro programa. Se define la función que se ejecutará al recibir una interrupción dada y se ejecuta limpiamente cuando ocurre, no hay que comprobar si se da o no una situación.

Simplemente te olvidas y se ejecutará única y exclusivamente cuando se alce la interrupción. No me digáis que no es elegante (SI, es una obsesión).

En realidad, nosotros funcionamos por interrupciones habitualmente, en respuesta a sucesos no previstos que nos sacan de la rutina habitual.

Imagínate que estás viendo tu serie favorita en la tele y estas esperando a tu colega, amigo o novia.

Hay dos maneras de abrirle la puerta. Una es pasar a ver si ha llegada cada, digamos dos minutos, para ver si esta con cara de pánfilo/pánfila en la puerta esperando a que le abramos.

La otra es establecer una interrupción, para eso están los timbres. Cuando tu compi llega, pulsa el timbre. Tu paras tu capitulo tranquilamente, dejas el refresco en la mesa y vas a abrirle.

Cuando vuelves con él, reanudas tu peli y recoges el refresco. ¿Qué tienen de raro las interrupciones? ¿Qué me decís del teléfono o de los Whatsapp? Es la misma idea. Y lo mismo pasa con tu Arduino.

¿Por qué voy a renunciar a las interrupciones y dedicarme a pasar por la puerta cada poco? Es absurdo. Las interrupciones no tienen nada de extraño ni de incognoscible. Dedícale un poco de tiempo y te encontrarás una herramienta magnifica que te resolverá limpiamente más de un problema.

TIPOS DE INTERRUPCIONES

De los tres sucesos que pueden disparar una interrupción

- Un evento hardware,
- Un evento programado, o Timer
- Una llamada por software.

Nos encontramos con que Arduino no soporta las interrupciones por software.

¿Y entonces porque hemos hablado de ellas? Pues, porque otros entornos de programación las aceptan y no será raro que en el futuro Arduino también.

Los eventos programados o Timers, son muy interesantes y tendrán una sesión monográfica propia en un futuro próximo. Pero por ahora vamos a meternos con las interrupciones disparadas por hardware.

LAS INTERRUPCIONES POR HARDWARE

Estas interrupciones hardware, se diseñaron por la necesidad de reaccionar a suficiente velocidad en tiempos inimaginablemente cortos a los que la electrónica trabaja habitualmente y a los que ni siquiera el software era capaz de reaccionar.

La idea que debéis que tener en mente es que vamos a definir una función que se ejecutará de forma asíncrona, sin planificación, cuando se ocurra un cierto suceso electrónico.

Para definir una interrupción necesitamos tres cosas:

- Un pin de Arduino que recibirá la señal de disparo
- Una condición de disparo
- Una función que se ejecutará, cuando se dispare la interrupción (Llamada **call back function**).

Lo primero es un pin de Arduino donde conectaremos el “*timbre”* de llamada. Dependiendo del modelo Arduino que tengamos, tendremos varias posibilidades:

MODELO ARDUINO	INT 0	INT 1	INT 2	INT 3	INT 4	INT 5
UNO	Pin 2	Pin 3				
MEGA	2	3	21	20	19	18
DUE	Todos los pines del DUE pueden usarse para interrupciones.					
Leonardo	3	2	0	1	7	

Esto quiere decir que el Arduino UNO puede definir dos **interrupciones hardware** llamadas 0 y 1, conectadas a los pines 2 y 3 (Para que no sea fácil).

El Mega como es habitual en él, acepta nada menos que 6 interrupciones diferentes. Y el DUE, muy sobrado, exhibe su poderío.

En cuanto a la condición de disparo puede ser:

- LOW, La interrupción se dispara cuando el pin es LOW.
- CHANGE, Se dispara cuando pase de HIGH a LOW o viceversa.
- RISING, Dispara en el flanco de subida (Cuando pasa de LOW a HIGH).
- FALLING, Dispara en el flanco de bajada (Cuando pasa de HIGH a LOW).
- Y una solo para el DUE: HIGH se dispara cuando el pin esta HIGH.

Si nuestra call back function se llama Funcion1 (), para activar la interrupción usaremos:

```
attachInterrupt(interrupt, ISR, mode)
```

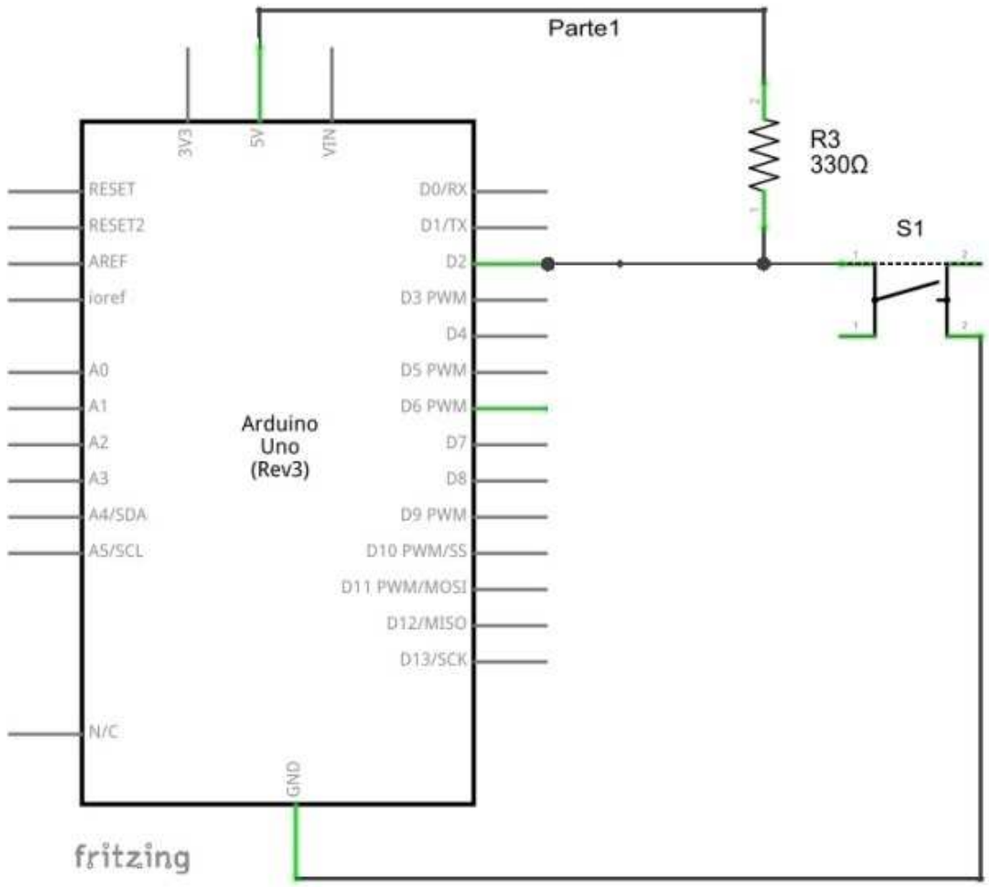
Donde Interrupt es el número de la interrupción, ISR será Funcion1 y mode es una de las condiciones que hemos visto arriba. Así en un Arduino UNO podría ser:

```
attachInterrupt(0, Funcion1, RISING) ;
```

Suponiendo que hemos enchufado la señal de interrupción al pin 2 de Arduino. Vamos a ver algún ejemplo de interrupciones.

ESQUEMA DE CONEXIONES

Es una especie de costumbre en Arduino, usar un pulsador para ilustrar el concepto de Interrupción, así que nos plegáramos a ello. Vamos a utilizar un típico circuito para leer un pulsador con un pullup.



Hasta ahora habríamos escrito el programa para leerlo así,

```
void setup()

   {   pinMode(2, INPUT);

       Serial.begin(9600);

   }



void loop()

   {   bool p = digitalRead(2);

       Serial.println(p);

   }
```

El resultado seria normalmente 1, por el pull up y la lectura bajaría a 0, al pulsar el botón. Nada nuevo en esto.

Pero vamos a reescribir el programa para establecer una interrupción en el pin 2 (Interrupción 0) .El programa quedara más o menos así:

```
int contador = 0;

int n = contador ;
```

```
void setup()

   {   Serial.begin(9600);

       attachInterrupt( 0, ServicioBoton, FALLING);

   }

void loop()

   {

       if (n != contador)

          {     Serial.println(contador);

                n = contador ;

          }

   }



void ServicioBoton()

   {    contador++ ;

   }
```

En primer lugar fijaros que hemos eliminado la definición de entrada del pin 2, porque no vamos a usarlo como input estrictamente. Con definir la **interrupción** es suficiente.

En segundo lugar usamos **attachInterrupt**() pasándole como parámetros la interrupción 0, que es el pin2 de Arduino UNO. SI fuese la Interrupción 1, la conectaríamos al pin 3 (Anda que…)

Le pasamos el nombre de la función **CallBack** ServicioBoton, que es de lo más sencilla. Un variable global contador, guarda el número de pulsaciones. Lo único que hace la función de servicio es aumentar contador en uno cada vez que se pulsa y volver.

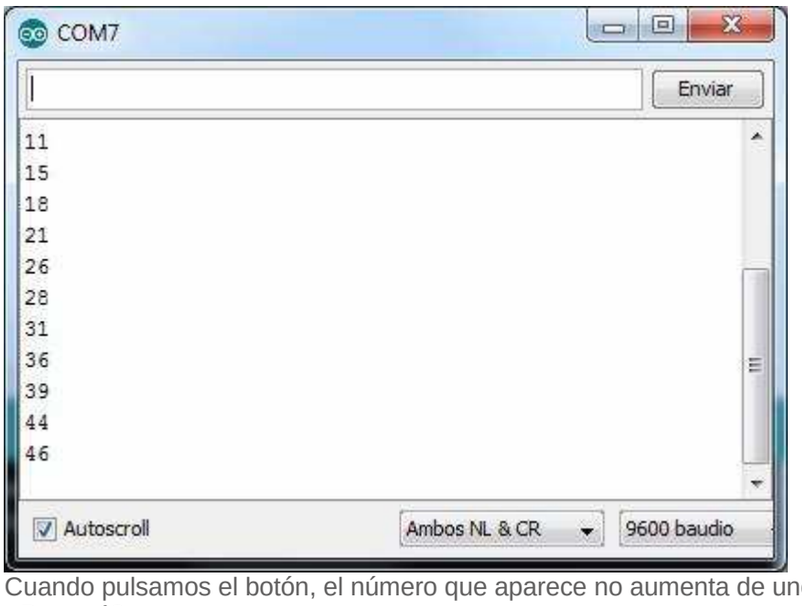
Y por último el trigger es FALLING porque el estado es normalmente HIGH y baja a LOW al pulsar el botón, utilizaremos el disparo con el flanco de bajada, o sea FALLING o LOW.

- Un disparador en inglés es trigger y es el nombre que encontrareis en la jerga técnica.

El loop comprueba si el número de pulsaciones ha cambiado y si es así lo imprime, pero puede dedicarse a hacer cualquier cosa, porque no vamos a perder ninguna pulsación.

Os puede parecer una manera extravagante de hacer las cosas pero no me digáis que no es elegante. De hecho todos los lenguajes modernos de alto nivel para Windows, Mac o Linux utilizan la programación por eventos que es algo básicamente igual a esto (Salvando las distancias claro).

Cuando veamos la salida en la consola tendremos una sorpresa esperada:



Cuando pulsamos el botón, el número que aparece no aumenta de uno en uno si no a golpes. ¿Por qué?

Pues como ya vimos en su día, se debe a los rebotes del pulsador. Decíamos en la sesión "Condicionales y botones", que para eliminar el rebote de los botones, tenemos que hacer el debouncing y allí lo hacíamos con un delay de 250 ms.

Pero vamos a tener un problema. No podemos usar un delay dentro de una interrupción. No funciona. ¿Cómo dice?

Hay varias consideraciones a tener en cuenta con las interrupciones:

- Haz lo que quieras pero no te demores. Acaba cuanto antes y lárgate.
- Hay cosas que no funcionan, como las funciones delay (), millis () y cualquier cosa que dependa de interrupciones o timers.
- Ni se te ocurra usar un Serial. Nada en una interrupción, son muy lentos (Y además tampoco funcionan porque también dependen de interrupciones).
- Debes entender que una interrupción es como un estado de excepción, que se puede usar sin reparos, pero entendiendo que hay que hacer el trabajo y salir cuanto antes.
- De hecho, una ISR o función CallBack, no puede devolver parámetros ni tampoco recibirlos

Además cuando definimos una variable global como contador, de la que depende una función ISR, se recomienda definirla como volatile y no como una global normal.

- *Estrictamente hablando, volatile no es un tipo de variable, sino una directiva para el compilador.*
- *Eso significa que la variable en cuestión, debe ser almacenado de un cierto modo que evite algunos problemas raros que pueden surgir cuando una variable puede ser cambiada por el ISR o por el programa (algo que no ocurre en nuestro ejemplo).*
- *Bajo ciertas circunstancias puede surgir un conflicto y volatile lo evita y por eso se recomineda hacerlo así siempre.*

Si necesitas un retraso para algo, o sea , un delay, siempre es mejor, comprobar el tiempo transcurrido y decidir si se toma la acción o no. Veamos otro ejemplo

```
volatile int contador = 0;   // Somos de lo mas obedientes

int n = contador ;

long T0 = 0 ;  // Variable global para tiempo
```

```
void setup()

   {    pinMode(2, INPUT);

        Serial.begin(9600);

        attachInterrupt( 0, ServicioBoton, LOW);

   }

void loop()

   {   if (n != contador)

          {   Serial.println(contador);

              n = contador ;

          }

   }

void ServicioBoton()

   {

       if ( millis() > T0  + 250)

          {   contador++ ;

              T0 = millis();

          }

    }
```

En primer lugar definimos contador como volátil, por prescripción médica, y definimos otra variable global T0 para almacenar el tiempo a partir del cual contaremos.

En la ISR la diferencia es que comprobamos si el valor actúan de millis es mayor en 250 ms a la última vez que paso por la interrupción. Si no es así, lo consideramos un rebote y lo ignoramos olímpicamente. Por el contrario si ha pasado un tiempo prudencial incrementamos contador.

La ventaja d este sistema es que no congelamos el procesador con un delay, si no que le dejamos seguir con su trabajo, atendiendo otras interrupciones, por ejemplo.

Pero….Un momento… No habíamos dicho que millis() no funciona en las interrupciones.

Así es. Mientras una **interrupción** *esta activa* millis *está congelada* y su valor no cambiará, pero sigue pudiéndose leer.

- *Mientras estas dentro de una interrupción, todas las demás interrupciones, son ignoradas, por eso, nada que dependa de otras interrupciones funciona.*
- *Por eso es importante salir pronto, para garantizar que no nos perdemos nada de interés.*
- *Mientras una interrupción está activa, millis() y micros()se congelan. Eso quiere decir que si tienes unos cuantos miles de interrupciones por segundo (Como si estas midiendo la frecuencia de una onda de audio) tu medida de tiempo con millis o micros se puede ver distorsionada.*

Por ultimo os conviene saber que existen algunas otras instrucciones relativas a las interrupciones:

- **noInterrupts**(), Desactiva la ejecución de interrupciones hasta nueva orden.
- **Interrupts**(), Reinicia las interrupciones definidas con attachInterrupt().
- **detachInterrupt**(num Interrupt), Anula la interrupción indicada.

RESUMEN DE LA SESIÓN

- Hemos conocido el motivo y el concepto de las interrupciones.
- Hemos empezado por estudiar las interrupciones hardware, dejando los Timers para futuras sesiones.
- Vimos que en Arduino UNO solo disponemos de 2 interrupciones llamadas 0 y 1 en los pines 2 y 3 respectivamente.
- Vimos los posibles disparos de una interrupción por flanco de subida, bajado o LOW.

MAS SOBRE INTERRUPCIONES (OTRO ARTICULO)

QUÉ SON Y CÓMO USAR INTERRUPCIONES EN ARDUINO

Las interrupciones son **un mecanismo muy potente y valioso en procesadores y autómatas**. Arduino, por supuesto, no es una excepción. En esta entrada veremos qué son las interrupciones, y como usarlas en nuestro código.

Para entender la utilidad y necesidad de las interrupciones, supongamos que tenemos Arduino conectado a un sensor, por ejemplo encoder óptico que cuenta las revoluciones de un motor, un detector que emite una alarma de nivel de agua en un depósito, o un simple pulsador de parada.

Si queremos detectar un cambio de estado en esta entrada, el método que hemos usado hasta ahora es emplear las entradas digitales para **consultar repetidamente el valor de la entrada**, con un intervalo de tiempo (delay) entre consultas.

Este mecanismo se denomina “poll”, y tiene 3 claras desventajas.

- Suponer un continuo consumo de procesador y de energía, al tener que preguntar continuamente por el estado de la entrada.
- Si la acción necesita ser atendida inmediatamente, por ejemplo en una alerta de colisión, esperar hasta el punto de programa donde se realiza la consulta puede ser inaceptable.
- Si el pulso es muy corto, o si el procesador está ocupado haciendo otra tarea mientras se produce, es posible que nos saltemos el disparo y nunca lleguemos a verlo.

Para resolver este tipo de problemas, los microprocesadores incorporan el concepto de interrupción, que es **un mecanismo que permite asociar una función a la ocurrencia de un determinado evento**. Esta función de callback asociada se denomina ISR (Interruption Service Rutine).

Cuando ocurre el evento **el procesador "sale" inmediatamente del flujo normal del programa y ejecuta la función ISR asociada** ignorando por completo cualquier otra tarea (por esto se llama interrupción). Al finalizar la función ISR asociada, el procesador vuelve al flujo principal, en el mismo punto donde había sido interrumpido.

Como vemos, las interrupciones son un mecanismo muy potente y cómodo que **mejora nuestros programas y nos permite realizar acciones que no serían posibles sin el uso de interrupciones**.

Para usar interrupciones en dispositivos físicos (como pulsadores, sensores ópticos, …) **debemos antes eliminar el efecto "rebote"**, como podéis ver en la entrada Aplicar debounce al usar interrupciones en Arduino

INTERRUPCIONES EN ARDUINO

Arduino dispone de dos tipos de eventos en los que definir interrupciones. Por un lado tenemos las interrupciones de timers (que veremos en su momento al hablar de temporizadores. Por otro lado, tenemos las **interrupciones de hardware, que responden a eventos ocurridos en ciertos pines físicos**.

Dentro de las interrupciones de hardware, que son las que nos ocupan en esta entrada, **Arduino es capaz de detectar los siguientes eventos**.

- RISING, ocurre en el flanco de subida de LOW a HIGH.
- FALLING, ocurre en el flanco de bajada de HIGH a LOW.
- CHANGING, ocurre cuando el pin cambia de estado (rising + falling).
- LOW, se ejecuta continuamente mientras está en estado LOW.

Los pines susceptibles de generar interrupciones varían en función del modelo de Arduino.

En Arduino y Nano se dispone de dos interrupciones, 0 y 1, asociados a los pines digitales 2 y 3.El Arduino Mega dispone de 6 interrupciones, en los pines 2, 3, 21, 20, 19 y 18 respectivamente. Arduino Due dispone de interrupciones en todos sus pines.

Modelo	INT0	INT1	INT2	INT3	INT4	INT5
UNO	2	3				
Nano	2	3				
Mini Pro	2	3				
Mega	2	3	21	20	19	18
Leonardo	3	2	0	1	7	
Due	En todos los pines					

LA FUNCIÓN ISR

La función asociada a una interrupción se denomina ISR (Interruption Service Routines) y, por definición, **tiene que ser una función que no recibe nada y no devuelva nada**.

Dos ISR no pueden ejecutarse de forma simultánea. En caso de dispararse otra interrupción mientras se ejecuta una ISR, la función ISR se ejecuta una a continuación de otra.

LA ISR, CUANTO MÁS CORTA MEJOR

Al diseñar una ISR debemos mantener como objetivo que tenga **el menor tiempo de ejecución posible**, dado que mientras se esté ejecutando el bucle principal y todo el resto de funciones se encuentran detenidas.

Imaginemos, por ejemplo, que el programa principal ha sido interrumpido mientras un motor acercaba un brazo para coger un objeto. Una interrupción larga podría hacer que el brazo no para a tiempo, tirando o dañando el objeto.

Frecuentemente la función de la ISR se limitará a activar un flag, incrementar un contador, o modificar una variable. Esta modificación será atendida posteriormente en el hilo principal, cuando sea oportuno.

No empleéis en una ISR un proceso que consuma tiempo. Esto incluye cálculos complejos, comunicación (serial, I2C y SPI) y, en la medida de lo posible, cambio de entradas o salidas tanto digitales como analógicas.

LAS VARIABLES DE LA ISR COMO “VOLATILES”

Para poder modificar una variable externa a la ISR dentro de la misma **debemos declararla como “volatile”**. El indicador “volatile” indica al compilador que la variable tiene que ser consultada siempre antes de ser usada, dado que puede haber sido modificada de forma ajena al flujo normal del programa (lo que, precisamente, hace una interrupción).

Al indicar una variable como Volatile el compilador desactiva ciertas optimizaciones, lo que supone una pérdida de eficiencia. Por tanto, sólo debemos marcar como volatile las variables que realmente lo requieran, es decir, **las que se usan tanto en el bucle principal como dentro de la ISR**.

EFECTOS DE LA INTERRUPCIÓN Y LA MEDICIÓN DEL TIEMPO

Las interrupciones **tienen efectos en la medición del tiempo de Arduino**, tanto fuera como dentro de la ISR, porque Arduino emplea interrupciones de tipo Timer para actualizar la medición del tiempo.

EFECTOS FUERA DE LA ISR

Durante la ejecución de una interrupción **Arduino no actualiza el valor de la función millis y micros**. Es decir, el tiempo de ejecución de la ISR no se contabiliza y Arduino tiene un desfase en la medición del tiempo.

Si un programa tiene muchas interrupciones y estas suponen un alto tiempo de ejecución, **la medida del tiempo de Arduino puede quedar muy distorsionada** respecto a la realidad (nuevamente, un motivo para hacer las ISR cortas).

EFECTOS DENTRO DE LA ISR

Dentro de la ISR el resto de interrupciones están desactivadas. Esto supone:

- La función millis no actualiza su valor, por lo que no podemos utilizarla para medir el tiempo dentro de la ISR. (sí podemos usarla para medir el tiempo entre dos ISR distintas)
- Como consecuencia la función delay() no funciona, ya que basa su funcionamiento en la función millis()
- La función micros() actualiza su valor dentro de una ISR, pero empieza a dar mediciones de tiempo inexactas pasado el rango de 500us.
- En consecuencia, la función delayMicroseconds funciona en ese rango de tiempo, aunque debemos evitar su uso porque no deberíamos introducir esperas dentro de una ISR.

CREAR INTERRUPCIONES EN ARDUINO

Para definir una interrupción en Arduino usamos la función:

```
1 attachInterrupt(interrupt, ISR, mode);
```

Donde interrupt es el número de la interrupción que estamos definiendo, ISR la función de callback asociada, y mode una de las opciones disponibles (Falling, Rising, Change y Low)

No obstante, **es más limpio emplear la función digitalPinToInterrupt()**, que convierte un Pin a la interrupción equivalente. De esta forma se favorece el cambio de modelo de placa sin tener que modificar el código.

```
1 attachInterrupt(digitalPinToInterrupt(pin), ISR, mode);
```

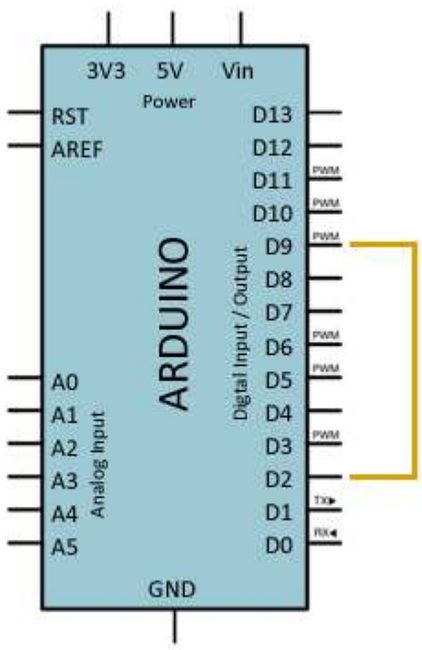
Otras funcionas interesantes para la gestión de interrupciones son:

- DetachInterrupt(interrupt), anula la interrupción.
- NoInterrupts(), desactiva la ejecución de interrupciones hasta nueva orden. Equivale a sli()
- Interrupts(), reactiva las interrupciones. Equivale a cli()

PROBANDO LAS INTERRUPCIONES EN ARDUINO

Para probar las interrupciones en Arduino, vamos a emplear una salida digital de Arduino para emular una señal digital. En el mundo real, sería otro dispositivo (un sensor, otro procesador…) el que generaría esta señal, y nosotros la captaríamos con Arduino.

Conectamos mediante un cable el pin digital 10 al pin digital 2, asociado a la interrupción 0.



HACIENDO PARPADEAR UN LED A TRAVÉS DE INTERRUPCIONES

En el siguiente código definimos el pin digital 10 como salida, para emular una onda cuadrada de periodo 300ms (150ms ON y 150ms OFF).

Para visualizar el funcionamiento de la interrupción, en cada flanco activo del pulso simulado, encendemos/apagamos el LED integrado en la placa, por lo que el LED parpadea a intervalos de 600ms (300ms ON y 300ms OFF)

Puede que a estas alturas ver parpadear un LED no parezca muy espectacular, pero no es tan simple como parece. No estamos encendiendo el LED con una salida digital, si no que **es la interrupción que salta la que enciende y apaga el LED** (el pin digital solo emula una señal externa).

```
1   const int emuPin = 10;
2
3   const int LEDPin = 13;
4   const int intPin = 2;
5   volatile int state = LOW;
6
7   void setup() {
8     pinMode(emuPin, OUTPUT);
9     pinMode(LEDPin, OUTPUT);
10    pinMode(intPin, INPUT_PULLUP);
11    attachInterrupt(digitalPinToInterrupt(intPin), blink, CHANGE);
12 }
13
14 void loop() {
15    //esta parte es para emular la salida
16    digitalWrite(emuPin, HIGH);
17    delay(150);
18    digitalWrite(emuPin, LOW);
19    delay(150);
20 }
21
22 void blink() {
23    state = !state;
24    digitalWrite(LEDPin, state);
25 }
```

CONTANDO DISPAROS DE UNA INTERRUPCIÓN

El siguiente código empleamos el mismo pin digital para emular una onda cuadrada, esta vez de intervalo 2s (1s ON y 1s OFF).

En cada interrupción actualizamos el valor de un contador. Posteriormente, en el bucle principal, comprobamos el valor del contador, y si ha sido modificado mostramos el nuevo valor.

Al ejecutar el código, veremos que en el monitor serie se imprimen números consecutivos a intervalos de dos segundos.

```
1   const int emuPin = 10;
2
3   const int intPin = 2;
4   volatile int ISRCounter = 0;
5   int counter = 0;
6
7
8   void setup()
9   {
10    pinMode(emuPin, OUTPUT);
11
12    pinMode(intPin, INPUT_PULLUP);
13    Serial.begin(9600);
14    attachInterrupt(digitalPinToInterrupt(intPin), interruptCount, LOW);
15 }
16
17 void loop()
18 {
19    //esta parte es para emular la salida
20    digitalWrite(emuPin, HIGH);
21    delay(1000);
22    digitalWrite(emuPin, LOW);
23    delay(1000);
24
25
26    if (counter != ISRCounter)
27    {
28      counter = ISRCounter;
29      Serial.println(counter);
30    }
31 }
32
33 void interruptCount()
34 {
```

```
35    ISRCounter++;
36    timeCounter = millis();
37 }
```

Lógicamente, nuestro objetivo es emplear interrupciones de hardware **no solo con otros dispositivos digitales, si no también con dispositivos físicos** como pulsadores, sensores ópticos…

Sin embargo, como hemos adelantado, **estos dispositivos generan mucho ruido en los cambios de estado**, lo que provoca que las interrupciones se disparen múltiples veces. Este fenómeno se denomina "rebote" y aprenderemos a eliminarlo en la siguiente entrada.

Cómo Y Por Qué Usar Las Interrupciones En Arduino (otro articulo)

Tienes tu módulo de detección de sonido, presencia… conectado a tu Arduino y quieres activar algún motor o modificar una variable cuando suceda algo (haya ruido, entre alguien…) pero claro ¿Si no sabes cuándo va a suceder ese evento cómo haces el código? ¿Te pasas todo el día monitorizando ese pin? ¿Y si justo sucede cuando la instrucción que se está ejecutando es un delay()? ¿Y si es un sonido impulsivo (como un golpe) y no lo detectas con el código? Todos estos problemas encuentran su solución en el uso de las interrupciones. **Las interrupciones en Arduino te permiten detectar eventos de forma asíncrona con independencia de las líneas de código que se estén ejecutando** en ese momento en tu microcontrolador.

Haz click para ir a lo que más te interese.

- 1 Tutorial Sobre El Manejo De Interrupciones
  - 1.1 ¿Qué Es Una Interrupción?
  - 1.2 ¿Para Qué Sirven Las Interrupciones De Arduino?

- 1.3 Ventajas E Inconvenientes De Usar Interrupciones
- 1.4 ¿En Qué Pines De Arduino Se Pueden Usar Las Interrupciones?
- 1.5 Modos De Activar Las Interrupciones
- 1.6 Código Asociado A Las Interrupciones
- 1.7 Ejemplo Del Uso De Interrupciones En Arduino
- 1.8 Eliminando El Efecto Rebote
  - 1.8.1 Solución Vía Hardware
  - 1.8.2 Solución Vía Software
- 1.9 Información Adicional

Tutorial Sobre El Manejo De Interrupciones

En este tutorial vas a aprender lo que se puede y no se puede hacer con las interrupciones en tu placa Arduino y de qué forma. Aprenderás cómo con unos simples pasos puedes aprovechar una de las herramientas más interesantes de Arduino (así como cualquier otro microcontrolador), con lo que seguro que muchos de tus proyectos mejorarán considerablemente (serán más eficientes y precisos).

Este (al igual que el anterior en el que te hablé de **cómo usar la memoria EEPROM de Arduino**) es uno de los tipos de post que más me gusta y eso se debe a que no necesitas comprar ningún módulo ni placa, es algo que ya tenías y que no sabías usar, es algo que tenías y, simplemente, estabas desaprovechando, es algo que me recuerda que te diga: ¡Conoce tu Arduino! Vale, tampoco me voy a poner en plan **Bruce Lee** pero en serio, gastamos mucho dinero para hacer cosas que podríamos hacer con lo que ya tenemos y usar las interrupciones en Arduino es un gran ejemplo de esto.

¿Qué Es Una Interrupción?

Una interrupción consiste básicamente en detectar detectar un evento y realizar algo en consecuencia. Tu Arduino tiene una serie de pines (podrás ver cuáles más adelante) a los que puedes asociar un módulo (un segmento de código) de modo que este se ejecute cuando en ese pin cambie de un estado a otro que tú previamente has establecido.

De hecho, seguramente ya habrás utilizado las interrupciones de Arduino sin saberlo. Cuando utilizas instrucciones del tipo millis() estás utilizando interrupciones. ¿Te suena eso de que la función **millis()** vuelve a 0 tras 50 días con tu Arduino conectado? ¡Exacto! Eso también se hace con interrupciones (aunque de otro tipo al que vas a ver en este post).

¿Para Qué Sirven Las Interrupciones De Arduino?

Como todos los códigos, sirve para lo que tú lo quieras usar en tu proyecto. No te ha ayudado mucho ¿Verdad? Bueno, para que te hagas un poco a la idea de para qué puedes usar las interrupciones te pongo unos cuantos ejemplos:

- Para detectar cambios como un pulsador que ha sido presionado.
- Para determinar cuando se ha terminado de gestionar la memoria EEPROM o Flash de tu Arduino.
- A modo de despertador del controlador. Esta es una interesantísima funcionalidad de tu placa que te permite mantener el consumo al mínimo dejando tu Arduino en standby hasta que suceda algún evento. Con ello podrás hacer que tus baterías duren mucho más. Como esta parte es especialmente interesante, escribiré un artículo al respecto. Entre tanto, te recomiendo que le eches un vistazo a **este post** si quieres aprender a optimizar las baterías de tu placa.
- Como un **Watchdog**. La idea es similar a la del punto anterior y espero también hacer un post sobre esto.
- Como el **complemento ideal a los módulos digitales** de sonido, temperatura… que disponen de un potenciómetro que regula cuándo se

activa la salida digital. Así puedes, por ejemplo, realizar un montaje simple en el que ocurra alguna acción cuando se supere un cierto umbral de sonido o una cierta distancia.

Ventajas E Inconvenientes De Usar Interrupciones

Soy de los que repiten una y otra vez los conceptos importantes así que aquí va: una interrupción es un evento asíncrono que se detecta con independencia del código que esté ejecutando en ese momento tu Arduino.

Ésto está muy bien si estás construyendo un coche que evita obstáculos ya que, si optas por el método tradicional, o vas parando cada poco tiempo los motores para medir la distancia, o es posible que, si todavía quedan muchas instrucciones por ejecutarse en tu código, cuando quieras comprobar la distancia para *cambiar de rumbo*, ya sea demasiado tarde (¡CRASH!). Sin embargo, **utilizando las interrupciones de Arduino puedes tener tu código ejecutando las instrucciones que sean y, sólo cuando esa interrupción se active, tu programa se va al código que has asociado a esa interrupción, lo ejecuta y luego vuelve a donde estaba** (a la parte del código que estaba ejecutando cuando se activó la interrupción).

Lo que te acabo de contar como una ventaja es al mismo tiempo un inconveniente. Si tu Arduino responde a la interrupción con independencia del código que se esté ejecutando ¿Qué pasa si se estaba ejecutando un delay() o una instrucción tipo millis()? Pues que tu Arduino atiende a la interrupción y vuelve por donde iba, esto es, ahora el tiempo real y el tiempo que *ha pasado* para tu Arduino no son lo mismo porque la interrupción se empezó a ejecutar cuando el tiempo que había transcurrido desde que conectaste tu Arduino (instrucción millis()) eran 200ms se activó tu interrupción y ha durado 20ms pero para tu microcontrolador ese tiempo no ha existido, es decir, el valor de la función millis() no se ha incrementado y ahora no puedes *fiarte* del tiempo transcurrido y lo peor es que **es posible que tú ni siquiera seas consciente de que la interrupción ha sucedido**. ¿Y si el motor de ese coche tenía que estar encendido 200ms y nada más que 200ms?

Como te puedes imaginar, todo esto que te acabo de contar es un poco más complejo. Sin embargo, espero que haya sido suficiente para que te hagas una idea de lo que puedes esperar cuando utilices interrupciones.

¿En Qué Pines De Arduino Se Pueden Usar Las Interrupciones?

Sólo hay unos pocos pines en los que se pueden realizar interrupciones y dependen del modelo de tu placa. Aquí te dejo una lista con los pines y los nombres de las interrupciones (al existir la posibilidad de tener varias interrupciones en la misma placa, tienes que asociarles un nombre):

Placa Arduino	Pin Digital De Cada Tipo Interrupción					
	Interrupción 0	Interrupción 1	Interrupción 2	Interrupción 3	Interrupción 4	Interrupción 5
UNO	2	3	-	-	-	-
DUE*	-	-	-	-	-	-
Leonardo	3	2	0	1	7	-
Mega	2	3	21	20	19	18
Micro	0	1	2	3	-	-
Mini	-	-	-	-	-	-
Nano	2	3	-	-	-	-
Ethernet	2	3	-	-	-	-
Esplora	-	-	-	-	-	-
Bluetooth	2	3	-	-	-	-
Fio	2	3	-	-	-	-
Pro Mini	22	3	-	-	-	-
Lilypad	-	-	-	-	-	-
*	La placa DUE permite establecer interrupciones en cualquier pin poniendo el nombre del mismo					

Modos De Activar Las Interrupciones

Ahora que sabes a qué pines se pueden asociar las interrupciones en Arduino, te falta conocer cómo activarlas, es decir, **qué tiene que suceder en ese pin para que la interrupción se ejecute**. Estos modos de activación los determinas tú cuando programas la interrupción y las posibilidades son las siguientes:

- LOW: La interrupción se activa cuando el valor de voltaje del pin elegido es bajo, esto es, 0V.
- CHANGE: La interrupción se activa cuando el pin cambia de valor, es decir, cuando pasa de LOW a HIGH o de HIGH a LOW.
- RISING: Se activa únicamente cuando el valor del pin pasa de LOW a HIGH.
- FALLING: Es el caso opuesto al modo RISING. Se activa la interrupción cuando el valor pasa de HIGH a LOW.
- HIGH: Este modo **solo está disponible en la placa DUE** y funciona de manera opuesta al modo LOW, es decir, se activa cuando el valor de voltaje del pin es alto.

Código Asociado A Las Interrupciones

Como te he comentado antes, la *gracia* de las interrupciones reside en que se ejecuten una serie de líneas de código cada vez que se activa la interrupción. Para conseguir esto deberás implementar un módulo asociado a dicha interrupción.

Seguramente ya conozcas el concepto de módulo pero por si tienes alguna duda de a lo que me estoy refiriendo te diré que, básicamente, se trata de un segmento de código que se ejecuta cuando es *invocado.* El void setup(){} y el void loop(){} a los que seguro que estás acostumbrado son un tipo particular de módulo. Puesto que no es el objetivo de este post, no voy a explicarte en profundidad las características de los módulos pero puedes dejar tu comentario si tienes alguna duda.

Para que puedas asociar un módulo a una interrupción correctamente, éste debe cumplir unas características concretas:

- **No puede tener parámetros de entrada**, es decir, no puede ser una función del tipo void f_interrupt (int entrada).
- No puede devolver ningún valor, por tanto **debe ser un módulo de tipo void**.
- Para que este segmento de código pueda interactuar con el resto de tu programa puedes crear una **variable de tipo volátil** (por ejemplo volatile int variable), lo que te permitirá leer el valor de dicha variable fuera del módulo asociado a la interrupción y actuar en consecuencia. Algunas de las prácticas más habituales consisten en crear una variable que cambia de estado (entre LOW y HIGH) cada vez que se activa la interrupción o un acumulador, una variable que incrementa su valor con cada interrupción y que puede ser consultada en cualquier momento por el resto de módulos.

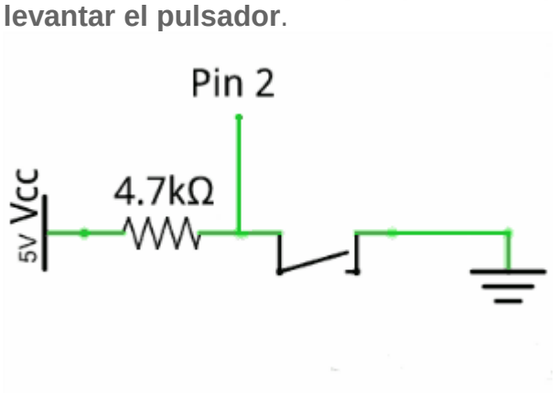
Ejemplo Del Uso De Interrupciones En Arduino

Ya conoces todas las características que debes tener en cuenta a la hora de implementar una interrupción. Estoy seguro de que podrías hacerlo sin problemas. Aun así, aquí tienes un ejemplo de cómo se crea una interrupción en Arduino.

Supón que quieres encender un LED o cualquier otro dispositivo mediante el uso de interrupciones (recuerda que en el caso del LED debes añadirle una resistencia, puedes leer más sobre la configuración de LEDs en **este post**), de tal manera que cada vez que se presione un pulsador, éste se encienda.

Una vez configurado un circuito para tu pulsador como el que puedes ver, conectado al pin correspondiente de interrupción que prefieras (en este caso el pin

2, es decir, la interrupción 0), debes elegir el modo en el que se activará dicha interrupción. Aunque quizás a primera vista pienses que el modo que debes elegir es LOW, recuerda que en ese estado la función se estaría ejecutando mientras tuvieses presionado el pulsador. Para evitar esto, **debes seleccionar el modo RISING o el FALLING según prefieras que tu LED se encienda al presionar o levantar el pulsador**.



Aquí te dejo el código para que veas cómo se haría.

```
/* EducaChip – Cómo Usar Las Interrupciones En Arduino*/
// www.educachip.com

//Se declara una variable asociada al pin en el que se va a conectar
//el LED. Se puede utilizar el valor 13 para observar el efecto de
//este código a través del LED de la placa de Arduino.
int LED = 8;

//Se declara una variable volátil que cambiará de estado en el
//módulo de la interrupción y se utilizará fuera.
//Si la variable no se utilizase fuera del módulo de la interrupción
//no sería necesario crearla de tipo volátil.
volatile int estado_actual = LOW;

//Se crea la función que se ejecutará cada vez que se active la
//interrupción. Esta función es de tipo void (por lo que no devuelve
//ningún valor) y no tiene parámetros de entrada.
void fun_cambio_estado()
{
  //Se cambia al estado contrario al actual, es decir, de LOW se pasa
  //a HIGH y de HIGH a LOW.
  estado_actual = !estado_actual;
}

void setup()
{
  //Se declara el pin digital correspondiente al LED como salida.
  pinMode(LED, OUTPUT);

  //Se determina la interrupción 0 (por lo que el pin asociado al
pulsador
  //será el 2 en caso de tratarse de un Arduino UNO o similar) a la que
se
  //le asocia la función fun_cambio_estado que se activará al presionar
el
  //pulsador (por tratarse del modo FALLING).
  attachInterrupt(0, fun_cambio_estado, FALLING);
}

void loop()
{
  //Se escribe el valor actual del LED.
  digitalWrite(LED, estado_actual);
}
```

Eliminando El Efecto Rebote

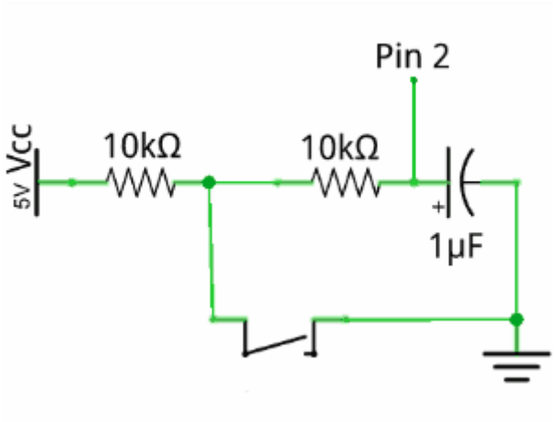
Cuando presionas un pulsador el contacto producido entre el botón y la base no siempre es fijo. Normalmente se producen una serie de rebotes al cambiar de estado el pulsador que tu Arduino percibe como si hubieses presionado varias

veces. Esto es un verdadero problema si estás utilizando las interrupciones de tu placa para incrementar el valor de una variable o si, como en el caso del LED, este problema provoca que, como no sabes cúantos rebotes va a haber, el estado final de tu variable estado_actual sea desconocido.

Para evitar este problema te voy a enseñar dos posibles soluciones:

Solución Vía Hardware

Esta solución es ideal si tu sistema necesita unos tiempos de reacción cortos o no quieres cargar tu microcontrolador con más código. La idea es sustituir el circuito del interruptor por el que puedes ver en la figura.



Solución Vía Software

Si no quieres gastar dinero en más componentes o el espacio es un factor crítico en tu proyecto, puedes modificar el código anterior añadiendo una variable de tiempo, de tal forma que justo después de que tu Arduino perciba un cambio de estado, tengan que pasar unos pocos milisegundos (deberás adaptarlo en función de tu pulsador pero un valor entorno a los 200ms debería funcionar en la mayoría de los casos) antes de que *se le haga caso* a cualquier otra pulsación.

Si tienes algún problema realizando cualquiera de los dos procedimientos recuerda dejar un comentario.

Información Adicional

Ahora que ya sabes cómo usar las interrupciones en Arduino, aquí tienes una serie de conceptos y tips que considero importantes:

- Debes implementar los módulos de tus interrupciones lo más cortos y eficientes que te sea posible.
- Sólo se puede ejecutar una instrucción de cada vez, esto es, si se está ejecutando una interrupción y se activa una segunda, no tendrá efecto hasta que la primera termine.
- No es posible utilizar la función delay() conjuntamente con las interrupciones.
- La función millis() no incrementa su valor durante el transcurso de una interrupción.
- A pesar de que las funciones delay() y millis() no funcionan correctamente cuando hay activadas interrupciones, **sí que puedes utilizar la función delayMicroseconds()**.
- Si estás transmitiendo o recibiendo datos (Serial.begin()) cuando se activa una interrupción, es posible que los pierdas.
- Si quieres utilizar la variable que es modificada por tu interrupción (estado_actual) fuera de ese módulo, debes crearla de tipo volatile.

- Puedes deshabilitar las interrupciones en cualquier momento utilizando la **instrucción detachInterrupt()**.

Esto ha sido todo. Espero que te haya gustado el post y que hayas aprendido a usar las interrupciones de Arduino. Si tienes cualquier duda, quieres contarnos tu experiencia con las interrupciones, etc. deja tu comentario.

Interrupciones (otro artículo)

Si queremos detectar un cambio de estado en esta entrada, el método que hemos usado hasta ahora es emplear las entradas digitales para consultar repetidamente el valor de la entrada, con un intervalo de tiempo (delay) entre consultas.

Este mecanismo se denomina “poll”, y tiene 3 claras desventajas.

- Suponer un continuo consumo de procesador y de energía, al tener que preguntar continuamente por el estado de la entrada.
- Si la acción necesita ser atendida inmediatamente, por ejemplo en una alerta de colisión, esperar hasta el punto de programa donde se realiza la consulta puede ser inaceptable.
- Si el pulso es muy corto, o si el procesador está ocupado haciendo otra tarea mientras se produce, es posible que nos saltemos el disparo y nunca lleguemos a verlo.

Para resolver este tipo de problemas, los microprocesadores incorporan el concepto de **interrupción**, que es un mecanismo que permite asociar una función a la ocurrencia de un determinado evento. Esta función de callback asociada se denomina ISR (Interruption Service Rutine).

En programación, una interrupción es una señal recibida por el procesador o MCU, para indicarle que debe «interrumpir» el curso de ejecución actual y pasar a ejecutar código específico para tratar esta situación.

Una interrupción es una suspensión temporal de la ejecución de un proceso, para pasar a ejecutar una subrutina de servicio de interrupción, la cual, por lo general, no forma parte del programa. Una vez finalizada dicha subrutina, se reanuda la ejecución del programa. Las interrupciones HW son generadas por los dispositivos periféricos habilitando una señal del CPU (llamada IRQ del inglés “interrupt request”) para solicitar atención del mismo.

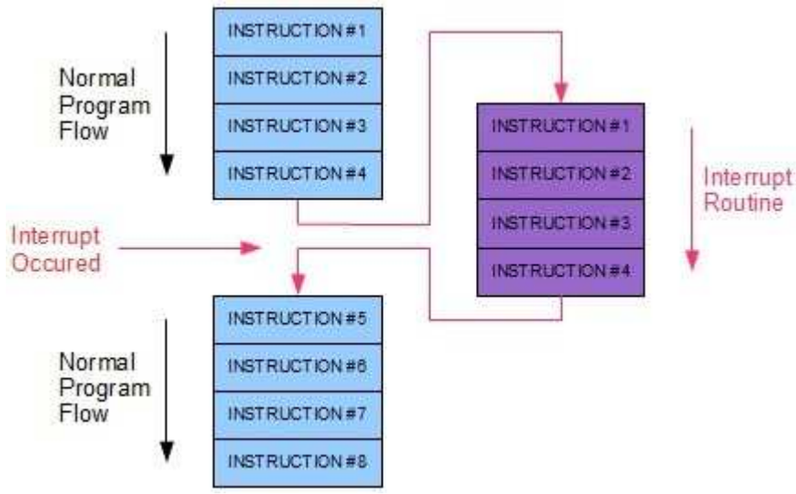
Todos los dispositivos que deseen comunicarse con el procesador por medio de interrupciones deben tener asignada una línea única capaz de avisar al CPU cuando le requiere para realizar una operación. Esta línea se denomina IRQ.

Las IRQ son líneas que llegan al **controlador de interrupciones**, un componente de hardware dedicado a la gestión de las interrupciones, y que está integrado en la MCU.

El **controlador de interrupciones** debe ser capaz de habilitar o inhibir las líneas de interrupción y establecer prioridades entre las mismas. Cuando varias líneas de petición de interrupción se activan a la vez, el controlador de interrupciones utilizará estas prioridades para escoger la interrupción sobre la que informará al procesador principal. También puede darse el caso de que una rutina de tratamiento de interrupción sea interrumpida para realizar otra rutina de tratamiento de una interrupción de mayor prioridad a la que se estaba ejecutando.

Procesamiento de una Interrupción:

1. Terminar la ejecución de la instrucción máquina en curso.
2. Salvar el estado del procesador (valores de registros y flags) y el valor del contador de programa en la pila, de manera que en la CPU, al terminar el proceso de interrupción, pueda seguir ejecutando el programa a partir de la última instrucción.
3. La CPU salta a la dirección donde está almacenada la rutina de servicio de interrupción (Interrupt Service Routine, o abreviado ISR) y ejecuta esa rutina que tiene como objetivo atender al dispositivo que generó la interrupción.
4. Una vez que la rutina de la interrupción termina, el procesador restaura el estado que había guardado en la pila en el paso 2 y retorna al programa que se estaba usando anteriormente.



Tipos de Interrupciones:

- Interrupciones HW o externas: Estas son asíncronas a la ejecución del procesador, es decir, se pueden producir en cualquier momento independientemente de lo que esté haciendo el CPU en ese momento. Las causas que las producen son externas al procesador y a menudo suelen estar ligadas con los distintos dispositivos de entrada o salida.
- Interrupciones SW: Las interrupciones por software son aquellas generadas por un programa en ejecución. Para generarlas, existen distintas instrucciones en el código máquina que permiten al programador producir una interrupción. Arduino no soporta las interrupciones por software
- Un evento programado o Timer. Son las interrupciones asociadas a los timers y gracias a ellas funciona millis().
- Excepciones: Son aquellas que se producen de forma síncrona a la ejecución del procesador típicamente causada por una condición de error en un programa. Normalmente son causadas al realizarse operaciones no permitidas tales como la división entre 0, el desbordamiento, el acceso a una posición de memoria no permitida, etc.

Definiciones genéricas de Interrupciones:

- https://es.wikipedia.org/wiki/Interrupci%C3%B3n
- http://programarfacil.com/blog/arduino-blog/interrupciones-con-arduino-ejemplo-practico/
- https://en.wikipedia.org/wiki/Interrupt

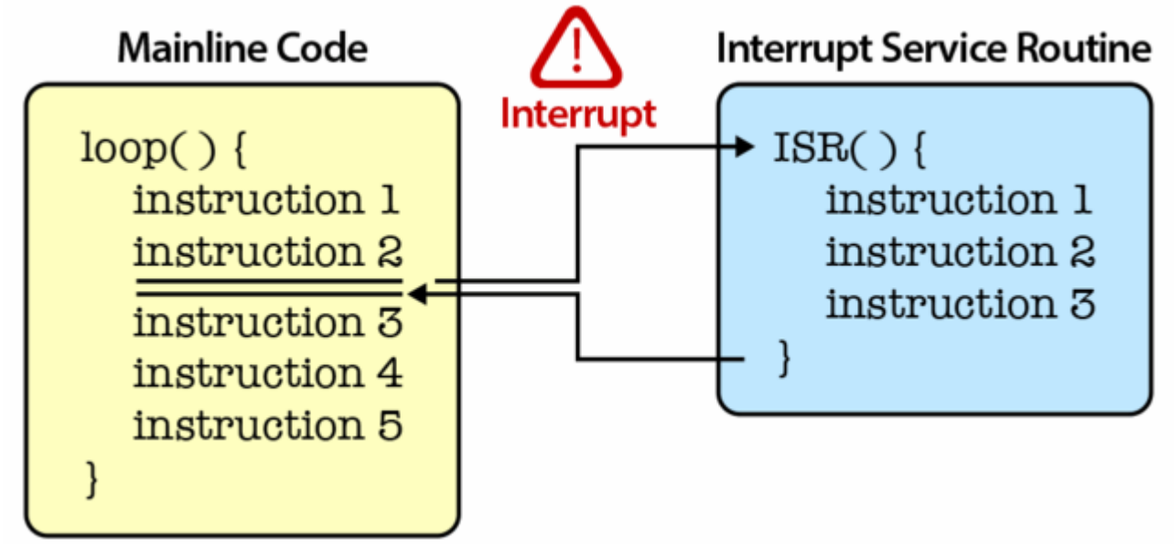
Interrupciones en Arduino

Internamente, Arduino (mejor dicho el microcontrolador AtMega) tiene ciertas interrupciones configuradas que lanza según la situación. Para la transmisión de datos a través del puerto serie, para resetear la placa antes de cargar un programa, comunicación I2C, etc…

El uso de interrupciones es casi obligatorio en un programa avanzado de un microcontrolador. Básicamente cuando un evento ocurre se levanta una bandera y la ejecución se traslada a una rama de código diferente. De esta forma no es necesario esperar un loop a comprobar que un evento ha ocurrido para ejecutar una acción.

Las interrupciones pueden ocurrir por un cambio en un puerto (solo en aquellos que soporten interrupciones HW), overflow en un timer, comunicación serie (USART), etc…

Normalmente no se ve, pero las interrupciones ocurren constantemente durante la operación normal de Arduino. Por ejemplo las interrupciones ayudan a las funciones delay() y millis() así como a la función Serial.read().



El procesador dentro de cualquier Arduino tiene dos tipos de interrupciones: “externas” y “cambio de pin”.

Más información:

- http://www.prometec.net/interrupciones/
- http://programarfacil.com/blog/arduino-blog/interrupciones-con-arduino-ejemplo-practico/
- http://www.sites.upiicsa.ipn.mx/polilibros/portal/Polilibros/P_terminados/PolilibroFC/Unidad_V/Unidad%20V_2.htm
- http://playground.arduino.cc/Code/Interrupts

Las interrupciones son muy útiles para hacer que las cosas ocurran automáticamente en los programas del microcontrolador y pueden resolver problemas de temporización.

Las tareas más usuales en las que usar interrupciones son en la monitorización de entradas de usuario o entradas externas críticas en el tiempo, así como en lectura de periféricos con requisitos de temporización muy específicos donde queramos capturar un evento que tiene una duración muy corta inferior al tiempo de loop de nuestro programa.

Para definir una interrupción necesitamos tres cosas:

- Un pin de Arduino que recibirá la señal de disparo
- Una condición de disparo
- Una función que se ejecutará, cuando se dispara la interrupción (Llamada call back function).

En cuanto a la condición de disparo puede ser:

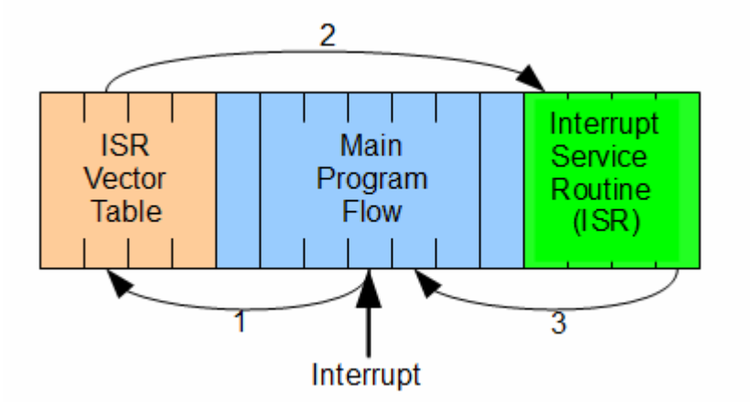
- LOW, La interrupción se dispara cuando el pin es LOW.
- CHANGE, Se dispara cuando pase de HIGH a LOW o viceversa.
- RISING, Dispara en el flanco de subida (Cuando pasa de LOW a HIGH).
- FALLING, Dispara en el flanco de bajada (Cuando pasa de HIGH a LOW).
- Y una solo para el DUE: HIGH se dispara cuando el pin esta HIGH.

Para saber todo sobre las interrupciones en el ATmega328p, debemos consultar su información en la página 32 y 82 de http://www.atmel.com/Images/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_datasheet.pdf

Los vectores de interrupción es una tabla en memoria que contiene la dirección de memoria de la primera instrucción de interrupción. ATmega328p Interrupt vectors, además esta tabla establece la prioridad de las interrupciones:

VectorNo.	Program Address[2]	Source	Interrupt Definition
1	0x0000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0x0012	TIMER2 OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1 COMPB	Timer/Coutner1 Compare Match B
14	0x001A	TIMER1 OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0 COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0 COMPB	Timer/Counter0 Compare Match B
17	0x0020	TIMER0 OVF	Timer/Counter0 Overflow
18	0x0022	SPI, STC	SPI Serial Transfer Complete
19	0x0024	USART, RX	USART Rx Complete
20	0x0026	USART, UDRE	USART, Data Register Empty
21	0x0028	USART, TX	USART, Tx Complete
22	0x002A	ADC	ADC Conversion Complete
23	0x002C	EE READY	EEPROM Ready
24	0x002E	ANALOG COMP	Analog Comparator
25	0x0030	TWI	2-wire Serial Interface
26	0x0032	SPM READY	Store Program Memory Ready

Y así funciona el ISR vector table:



Las librería de avr-libc que se usa para manejar las interrupciones es <avr/interrupt.h>:

- http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html
- http://www.atmel.com/webdoc/AVRLibcReferenceManual/group__avr__interrupts.html

Más información:

- http://courses.cs.washington.edu/courses/csep567/10wi/lectures/Lecture7.pdf
- http://ee-classes.usc.edu/ee459/library/documents/avr_intr_vectors/
- http://www.luisllamas.es/2016/04/que-son-y-como-usar-interrupciones-en-arduino/
- http://avrmicrotutor.blogspot.com.es/2011/08/basic-understanding-of-microcontroller.html
- https://felixmaocho.wordpress.com/2013/08/05/arduino-manejo-de-interrupciones/

Interrupciones Externas en Arduino

Estas interrupciones hardware, se diseñaron por la necesidad de reaccionar a suficiente velocidad en tiempos inimaginablemente cortos a los que la electrónica trabaja habitualmente y a los que ni siquiera el software era capaz de reaccionar.

Para las **interrupciones externas o hardware**, solo hay dos pines que las soportan en los ATmega 328 (p.e. Arduino UNO), son las INT0 y INT1 que están mapeadas a los pines 2 y 3. Estas interrupciones se pueden configurar con disparadores en RISING o FALLING para flancos o en nivel LOW. Los disparadores son interpretados por hardware y la interrupción es muy rápida.

El Arduino mega tiene más pines disponibles para interrupciones externas. Pines de External Interrupts para Mega: 2 (interrupt 0), 3 (interrupt 1), 18 (interrupt 5), 19 (interrupt 4), 20 (interrupt 3), and 21 (interrupt 2). Estos pines pueden ser configurados para disparar una interrupción al detectar un nivel bajo, un flanco ascendente, un flanco descendente o un cambio de nivel.

En el pin de reset también hay otra interrupción que sólo se dispara cuando detecta voltaje LOW y provoca el reset del microcontrolador.

Para configurar una interrupción en Arduino se usa la función attachInterrupt(). El primer parámetro a configurar es el número de interrupción, normalmente se usa el nº de pin para traducir al nº de interrupción.

Tabla de pines que soportan interrupción HW en Arduino:

Board	**Digital Pins Usable For Interrupts**
Uno, Nano, Mini, other 328-based	2, 3
Mega, Mega2560, MegaADK	2, 3, 18, 19, 20, 21
Micro, Leonardo, other 32u4-based	0, 1, 2, 3, 7
Zero	all digital pins, except 4
MKR1000 Rev.1	0, 1, 4, 5, 6, 7, 8, 9, A1, A2
Due	all digital pins

Esto quiere decir que el Arduino UNO puede definir dos interrupciones hardware llamadas 0 y 1, conectadas a los pines 2 y 3

Para saber qué número de interrupción estás asociada a un pin, debemos usar la función **digitalPinToInterrupt(pin)**. El número de interrupción su mapeo en los pines dependerá del MCI. El uso de número de interrupción puede provocar problemas de compatibilidad cuando el sketch funciona en diferentes placas.

Tabla de interrupciones y número de pin asociado:

Board	int.0	int.1	int.2	int.3	int.4	int.5
Uno, Ethernet	2	3				
Mega2560	2	3	21	20	19	18
32u4 based (e.g Leonardo, Micro)	3	2	0	1	7	

Arduino Due tiene grandes capacidades a nivel de interrupciones que permiten asociar una interrupción a cada uno de los pines disponibles. Arduino Zero permite asociar una interrupción a todos los pines excepto para el pin 4.

Aspectos importantes a tener en cuenta con el uso de interrupciones:

- Dentro de la función llamada desde la interrupción, la función delay() no funciona y el valor devuelto por millis() no aumenta. La razón es que estas funciones hacen uso de interrupciones que no se disparan mientras está disparada una interrupción externa.
- Los datos recibidos por el puerto serie se pueden perder mientras se está en la función llamada por la interrupción.

- Se deben declarar como “volatile” cualquier variable que sea modificada dentro de la función llamada por una interrupción: https://www.arduino.cc/en/Reference/Volatile

Tutorial de Interrupciones externas en MCUs AVR de 8 bits: http://www.avr-tutorials.com/interrupts/The-AVR-8-Bits-Microcontrollers-External-Interrupts

Programación de interrupciones externas en C: http://www.avr-tutorials.com/interrupts/avr-external-interrupt-c-programming

Las interrupciones de hardware, también conocidas como INT0 e INT1, llaman a una rutina de servicio de interrupción cuando algo sucede con uno de los pines asociados. La ventaja es que Arduino tiene una simple rutina de configuración para conectar la rutina de servicio de interrupción al evento: attachInterrupt (). La desventaja es que sólo funciona con dos pines específicos: pin digital 2 y 3 en la placa Arduino.

El Atmega328p solo tiene dos interrupciones de hardware INT0 e INT1, sin embargo los microcontroladores AVR pueden tener una interrupción ante un cambio en cualquier pin, es lo que denomina **pin change interrupt**. http://playground.arduino.cc/Code/Interrupts. Estas interrupciones no son soportadas directamente por Arduino y necesitan ser accedidas a través de una librería adicional.

Las **interrupciones de cambio de pin** pueden habilitarse en más pines. Para los ATmega 328, pueden habilitarse en cualquiera de los pines de señal disponibles. Estas son disparadas igual en flancos RISING o FALLING, pero depende del código de la interrupción configurar el pin que recibe la interrupción y determinar qué ha pasado. Las interrupciones de cambio de pin están agrupadas en 3 puertos de la MCU, por lo tanto hay 3 vectores de interrupciones para todo el conjunto de pines. Esto hace el trabajo de resolver la acción en una interrupción más complicada.

Port Pins	Source	Description
PORTD: PD2	INT0	External Interrupt Request 0
PORTD: PD3	INT1	External Interrupt Request 0
PORTB: PB0 to PB7	PCINT0	Pin Change External Interrupt Request 0
PORTC: PC0 to PC6	PCINT1	Pin Change External Interrupt Request 1
PORTD: PD0 to PD7	PCINT2	Pin Change External Interrupt Request 2

The AVR ATmega328P Microcontroller External Interrupt Table

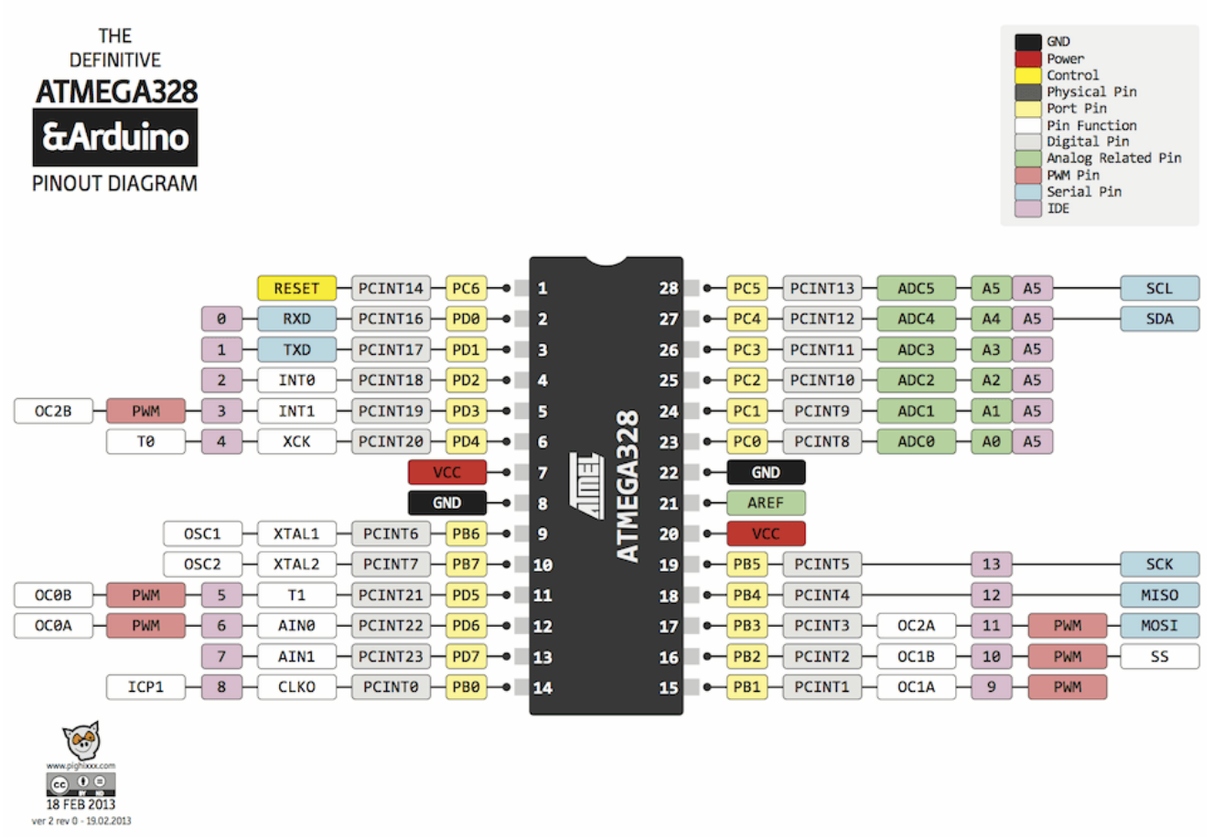
Si necesita más pines para interrupciones, hay un mecanismo para generar una interrupción cuando se cambia cualquier pin en uno de los puertos de 8 bits. No sabes pin, sino sólo en qué puerto. Si se genera una interrupción cuando se cambia uno de los pines ADC0 a ADC5 (utilizado como entrada digital). En ese caso, se llama a la rutina de servicio de interrupción ISR (PCINT1_vect). En la rutina se puede averiguar cuál de los pines específicos dentro de ese puerto ha sido el que ha cambiado.

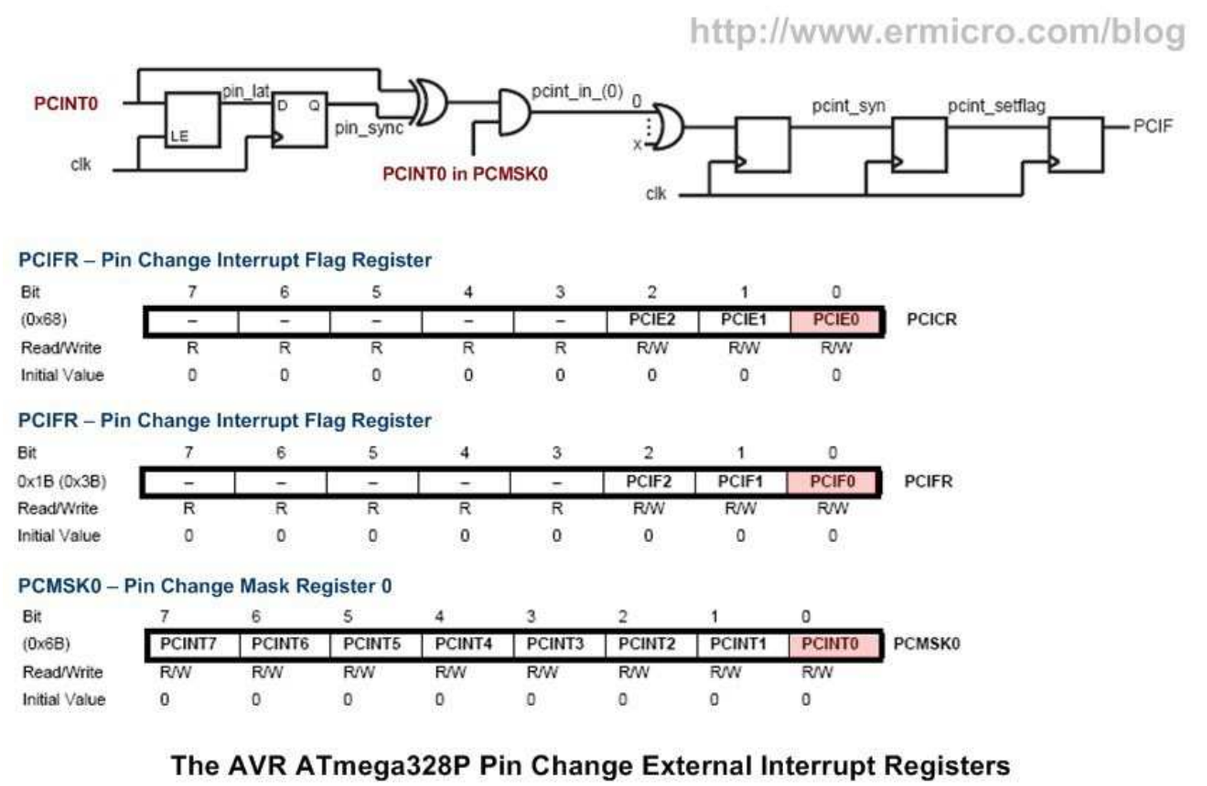
En primer lugar, los indicadores de habilitación de interrupción de cambio de pin deben ajustarse en el registro PCICR. Estos son los bits PCIE0, PCIE1 y PCIE2 para los grupos de pines PCINT7..0, PCINT14..8 y PCINT23..16 respectivamente. Los pines individuales se pueden activar o deshabilitar en los registros PCMSK0, PCMSK1 y PCMSK2. En el circuito Arduino, en combinación con la hoja de datos Atmel Atmega328, se puede ver que el pin PCINT0 corresponde al pin 0 en el puerto B (llamado PB0). Este es el pin 8 en el Arduino Uno. Otro ejemplo es el pin A0 de la tarjeta Arduino, que puede utilizarse como una entrada digital:

- Pin Change Interrupt Request 0 (pins D8 to D13) (PCINT0_vect)
- Pin Change Interrupt Request 1 (pins A0 to A5) (PCINT1_vect)
- Pin Change Interrupt Request 2 (pins D0 to D7) (PCINT2_vect)

“The Pin Change Interrupt Request 2 (PCI2) will trigger if any enabled PCINT[23:16] pin toggles. The Pin Change Interrupt Request 1 (PCI1) will trigger if any enabled PCINT[14:8] pin toggles. The Pin Change Interrupt Request 0 (PCI0) will trigger if any enabled PCINT[7:0] pin toggles.”

PCINT son unos pines determinados que se puede ver en la página 19 a que corresponden de http://www.atmel.com/Images/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_datasheet.pdf





The AVR ATmega328P Pin Change External Interrupt Registers

Los pines de interrupción deben estar configurados como INPUT, las resistencias pullup pueden habilitarse para poder detectar interruptores simples.

Es cierto que solo se puede configurar una rutina de interrupción para cada grupo de pines en un puerto, pero hay muchos casos donde es suficiente.

Más información:

- https://thewanderingengineer.com/2014/08/11/arduino-pin-change-interrupts/
- http://www.geertlangereis.nl/Electronics/Pin_Change_Interrupts/PinChange_en.html
- http://playground.arduino.cc/Main/PinChangeInterrupt
- http://www.ermicro.com/blog/?p=2292
- http://gammon.com.au/interrupts

- http://playground.arduino.cc//Main/PinChangeIntExample
- http://www.avr-tutorials.com/interrupts/about-avr-8-bit-microcontrollers-interrupts

También disponemos de la PinChangeInt Library. PinChangeInt implementa interrupciones de cambio de pin para el entorno Arduino. Esta biblioteca fue diseñada para el Arduino Uno / Duemilanove, y funciona bien en el Nano. El soporte MEGA está incluido pero no de una forma completa.

- http://playground.arduino.cc/Main/PinChangeInt
- https://github.com/GreyGnome/PinChangeInty
- nueva versión EnableInterrupt https://github.com/GreyGnome/EnableInterrupt
- wiki https://github.com/GreyGnome/EnableInterrupt/wiki

Registros para las interrupciones externas y pin change son:

- External Interrupt Control Register A: The External Interrupt Control Register A contains control bits for interrupt sense control.
  Name: **EICRA**
- External Interrupt Mask Register: When addressing I/O Registers as data space using LD and ST instructions, the provided offset must be used. When using the I/O specific commands IN and OUT, the offset is reduced by 0x20, resulting in an I/O address offset within 0x00 – 0x3F.
  Name: **EIMSK**
- External Interrupt Flag Register: When addressing I/O Registers as data space using LD and ST instructions, the provided offset must be used. When using the I/O specific commands IN and OUT, the offset is reduced by 0x20, resulting in an I/O address offset within 0x00 – 0x3F.
  Name: **EIFR**
- Pin Change Interrupt Control Register
  Name: **PCICR**
- Pin Change Interrupt Flag Register: When addressing I/O Registers as data space using LD and ST instructions, the provided offset must be used. When using the I/O specific commands IN and OUT, the offset is reduced by 0x20, resulting in an I/O address offset within 0x00 – 0x3F.
  Name: **PCIFR**
- Pin Change Mask Register 2
  Name: **PCMSK2**
- Pin Change Mask Register 1
  Name: **PCMSK1**
- Pin Change Mask Register 0
  Name: **PCMSK0**

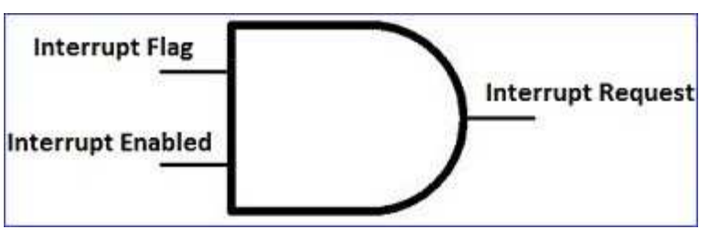
Cada interrupción está asociada con dos bits, un bit de indicador de interrupción (flag) y un bit habilitado de interrupción (enabled). Estos bits se encuentran en los registros de E/S asociados con la interrupción específica:

- El bit de indicador de interrupción se establece siempre que se produce el evento de interrupción, independientemente de que la interrupción esté o no habilitada. Registro **EIFR**
- El bit activado por interrupción se utiliza para activar o desactivar una interrupción específica. Básicamente se le dice al microcontrolador si debe o no responder a la interrupción si se dispara. Registro **EIMSK**

En resumen, básicamente, tanto la Interrupt Flag como la Interrupt Enabled son necesarias para que se genere una solicitud de interrupción como se muestra en la figura siguiente.

Aparte de los bits habilitados para las interrupciones específicas, el bit de enable de interrupción global DEBE ser habilitado para que las interrupciones se activen en el microcontrolador.

Para el microcontrolador AVR de 8 bits, este bit se encuentra en el registro de estado de E/S (SREG). La interrupción global habilitada es bit 7, en el SREG.



Interrupciones Internas en Arduino

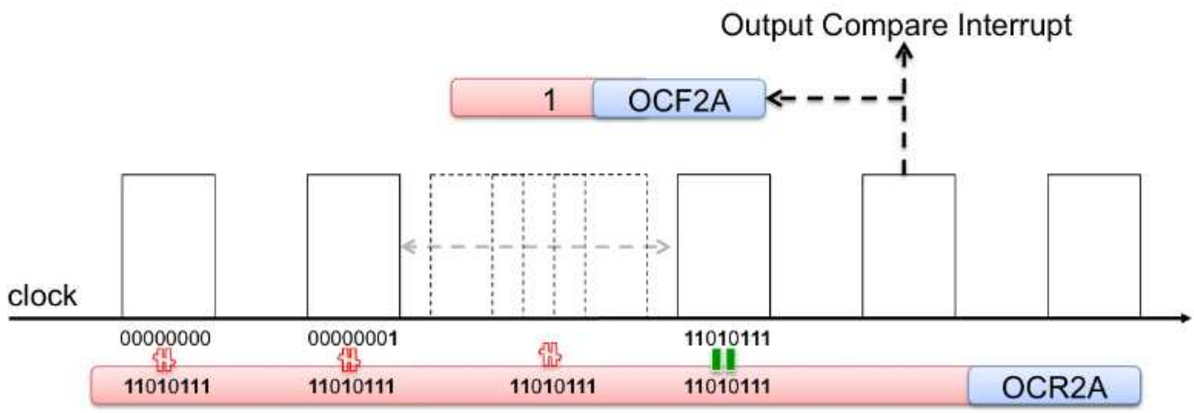
Las interrupciones internas en Arduino son aquellas interrupciones relacionadas con los timers y que también son denominadas interrupciones de eventos programados.

Arduino tiene tres timers. Son el timer cero, timer uno, timer dos.Timer cero y dos son de 8 bits y el temporizador uno es de 16 bits.

Crearemos una interrupción interna utilizando la interrupción de un temporizador. Para ello necesitaremos una librería adecuada para el temporizador que usemos. Podemos crear tres tipos de interrupción interna, son interrupción de desbordamiento de temporizador (timer overflow), interrupción de comparación de salida (Output Compare Match Interrupt), interrupción de captura de entrada (Timer Input Capture Interrupt).

Ya vimos que las interrupciones de los timers se usan para PWM y también con la librería MSTimer2: http://www.pjrc.com/teensy/td_libs_MsTimer2.html

En el apartado de timers trataremos más a fondo los temporizadores de Arduino y sus interrupciones asociadas.



Problemas y soluciones con las interrupciones en Arduino: http://rcarduino.blogspot.com.es/2013/04/the-problem-and-solutions-with-arduino.html

ISR (Interrupt Service Routines)

La función de callback asociada a una interrupción se denomina ISR (Interruption Service Rutine). ISR es una función especial que tiene algunas limitaciones, una ISR no puede tener ningún parámetro en la llamada y no pueden devolver ninguna función.

Dos ISR no pueden ejecutarse de forma simultánea. En caso de dispararse otra interrupción mientras se ejecuta una ISR, la función ISR se ejecuta una a continuación de otra.

Una ISR debe ser tan corta y rápida como sea posible, puesto que durante su ejecución se paraliza el curso normal del programa y las interrupciones se deshabilitan.

Se se usan varias ISR en el sketch, solo una se puede ejecutar y otras interrupciones serán ejecutadas después de que la ISR actual finalice, en un orden que depende las prioridad de las interrupciones y que depender de interrupt handler.

La función millis() no funciona dentro del ISR puesto que usa interrupciones para su uso. La función micros() funciona dentro de ISR pero después de 1-2 ms se empieza a comportar de forma extraña. delayMicroseconds() no usa ningún contador y funcionará correctamente dentro del ISR. Por lo tanto si la ISR dura mucho tiempo provocará retrasos en el reloj interno, puesto que millis() no avanza mientras se ejecuta el ISR.

Para pasar datos entre el programa principal y el ISR se usan las variables globales, pero para que estas variables se actualicen correctamente deben declararse con el modificador "volatile": https://www.arduino.cc/en/Reference/Volatile

Volatile es una palabra reservada que se pone delante de la definición de una variable para modificar la forma en que el compilador y el programa trata esa variable.

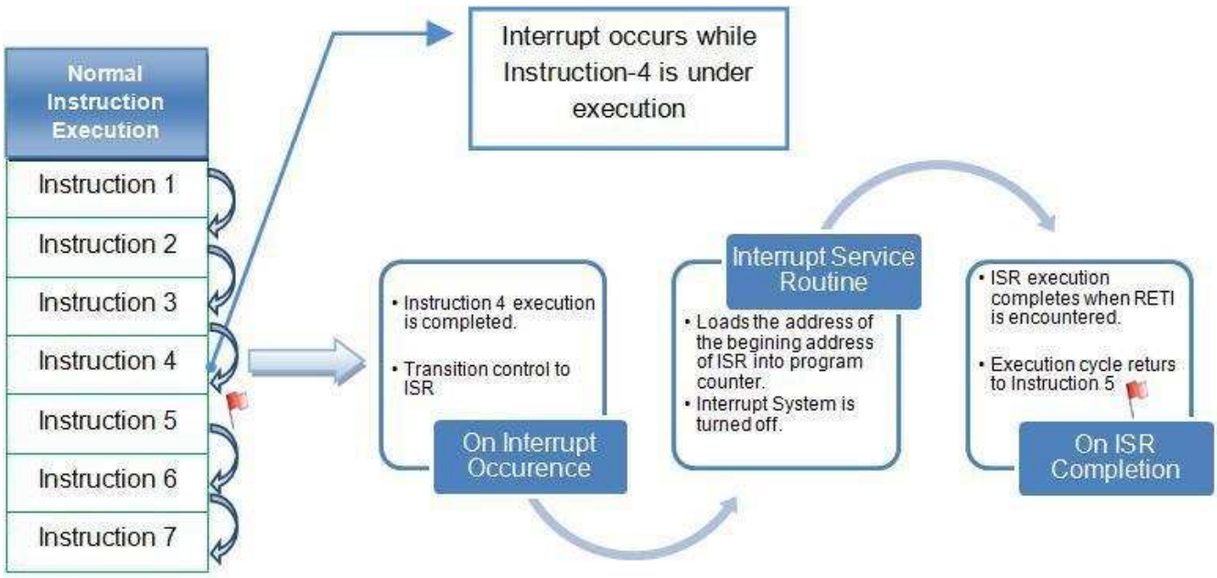
Declarar una variable volátil es una directiva para el compilador. Específicamente, dice al compilador que cargue la variable desde la RAM y no desde un registro de almacenamiento, que es una ubicación de memoria temporal donde se almacenan y manipulan las variables del programa. Bajo ciertas condiciones, el valor de una variable almacenada en registros puede ser inexacto.

Una variable debe ser declarada volátil siempre que su valor pueda ser cambiado por algo más allá del control de la sección de código en la que aparece, como un subproceso que se ejecuta simultáneamente. En Arduino, el único lugar en el que es probable que ocurra es en secciones de código asociadas con interrupciones, las llamadas rutinas de servicio de interrupción (ISR).

Para poder modificar una variable externa a la ISR dentro de la misma debemos declararla como "volatile". El indicador "volatile" indica al compilador que la variable tiene que ser consultada siempre antes de ser usada, dado que puede haber sido modificada de forma ajena al flujo normal del programa (lo que, precisamente, hace una interrupción). Al indicar una variable como Volatile el compilador desactiva ciertas optimizaciones, lo que supone una pérdida de eficiencia. Por tanto, sólo debemos marcar como volatile las variables que realmente lo requieran, es decir, las que se usan tanto en el bucle principal como dentro de la ISR.

Esta es la secuencia cuando se dispara una interrupción:

1. El microcontrolador completa la instrucción que está siendo ejecutada.
2. El programa de Arduino que se está ejecutando, transfiere el control a la Interrupt Service Routine (ISR). Cada interrupción tiene asociada una ISR que es una función que le dice al microcontrolador que hacer cuando ocurre una interrupción.
3. Se ejecuta la ISR mediante la carga de la dirección de comienzo de la ISR en el contador del programa.
4. La ejecución del ISR continua hasta que se encuentra el RETI (return from the interrupt instruction). Más información: http://www.atmel.com/webdoc/avrassembler/avrassembler.wb_RETI.html
5. Cuando ha finalizado ISR, el microcontrolador continua la ejecución del programa donde lo dejó antes de que ocurriera la interrupción.



Más información en: http://gammon.com.au/interrupts

Hay muchos I/O registros en el microcontrolador asociados a las interrupciones externas. Estos registros almacenan los flag de las interrupciones, los bits de enable, así como información de control de cada interrupción externa.

Más información de interrupciones: http://cs4hs.cs.pub.ro/wiki/roboticsisfun/chapter2/ch2_10_interrupts

Funciones de Interrupciones en Arduino

El core de Arduino ofrece una serie de instrucciones para programar las interrupciones externas, pero no las de pin change ni las de temporizadores:

- interrupts() – https://www.arduino.cc/en/Reference/Interrupts
  Habilita las interrupciones (antes han debido ser inhabilitadas con noInterrupts()). Las interrupciones permiten a ciertas tareas importantes que se ejecuten en segundo plano y defecto las interrupciones están habilitadas. Algunas funciones no funcionarán si se deshabilitan las interrupciones y las comunicaciones entrantes serán ignoradas, también podrán modificar ligeramente la temporización del código. Las interrupciones se pueden deshabilitar para casos particulares de secciones críticas del código.

- noInterrupts() – https://www.arduino.cc/en/Reference/NoInterrupts
  Deshabilita las interrupciones. Las interrupciones pueden ser habilitadas de nuevo con interrupts().
- attachInterrupt() – https://www.arduino.cc/en/Reference/AttachInterrupt
  Me permite configurar una interrupción externa, pero no otro tipo de interrupciones. El primer parámetro es el número de interrupción que va asociado a un pin, luego la función ISR y finalmente el modo.
- detachInterrupt() – https://www.arduino.cc/en/Reference/DetachInterrupt
  Deshabilita la interrupción. El parámetro que se le pasa es el número de la interrupción.
- digitalPinToInterrupt(pin) traduce el pin al número de interrupción específica.
- usingInterrupt() – https://www.arduino.cc/en/Reference/SPIusingInterrupt
  Deshabilita la interrupción externa pasada como parámetro en la llamada a SPI.beginTransaction() y se habilita de nuevo en endTransaction() para prevenir conflictos en las transacciones del bus SPI

En attachInterrupt() los modos disponibles que definen cuando una interrupción externa es disparada, se hace mediante 4 constantes:

- LOW, La interrupción se dispara cuando el pin es LOW.
- CHANGE, Se dispara cuando pase de HIGH a LOW o viceversa, es decir un cambio en el estado del pin.
- RISING, Dispara en el flanco de subida (Cuando pasa de LOW a HIGH).
- FALLING, Dispara en el flanco de bajada (Cuando pasa de HIGH a LOW).
- Y solo para el DUE: HIGH se dispara cuando el pin esta HIGH.

Normalmente, las variables globales se utilizan para pasar datos entre un ISR y el programa principal. Para asegurarse de que las variables compartidas entre un ISR y el programa principal se actualizan correctamente, declararlas como volátiles.

La función delay() no deshabilita las interrupciones, por lo tanto los datos recibidos por el serial Rx son guardados, los valores PWM funcionan y las interrupciones externas funcionan. Sin embargo la función delayMicroseconds() deshabilita las interrupciones mientras está ejecutándose.

Más información:

- https://www.arduino.cc/en/Reference/AttachInterrupt
- http://www.engblaze.com/we-interrupt-this-program-to-bring-you-a-tutorial-on-arduino-interrupts/

Para manejar las interrupciones por cambio de pin disponemos de varias librerías:
Ver http://playground.arduino.cc/Main/LibraryList#Interrupts

Multitarea

Utilizar interrupciones nos permitirá olvidarnos de controlar ciertos pines. Esto muy importante ya que dentro de una aplicación o programa, no vamos a hacer una única cosa. Por ejemplo, queremos que un LED se encienda o se apague cuando pulsamos un botón. Esto es relativamente sencillo pero cuando además queremos que otro LED parpadee, la cosa se complica.

La probabilidad de capturar el evento cuando se pulsa el botón disminuye con el aumento de tiempo de parpadeo. En estos casos, y en muchos otros, nos interesa liberar el procesador de Arduino para que solo cuando se pulse el botón, haga una acción determinada. Así no tendremos que estar constantemente comprobando el pin X si ha pulsado el botón.

Precisamente este es el sentido de las interrupciones, poder hacer otras cosas mientras no suceda el evento, pero cuando ese evento externo esté presente, que ejecute rápidamente el código asociado.

Los controladores de interrupciones: administran la ejecución de tareas por interrupciones, lo cual permite la multitarea.

Todas las tareas que hemos realizado hasta ahora han sido síncronas. Es decir, solicitamos unos datos (puerto serie, ethernet, etc…), esperamos la respuesta y los mostramos en pantalla. La contraposición es un proceso asíncrono, nosotros lanzamos la petición y en cuanto se pueda se realizará y se mostrará el resultado sin esperar.

Si nos encontramos con respuestas lentas, no es buena técnica esperar de forma “síncrona” porque seguramente obtengamos un error de que se ha excedido el tiempo de espera “timeout”. Si estamos descargando datos de gran volumen, tampoco es buena técnica que dejemos la aplicación “congelada” mientras se descargan los datos. Lo ideal es lanzar la descarga de fondo, es decir, de forma “asíncrona”.

Un proceso síncrono es aquel que se ejecuta y hasta que no finaliza solo se ejecuta ese proceso (todo en el mismo loop), mientras que uno asíncrono comienza la ejecución y continúan ejecutándose otras tareas por lo que el proceso total se completa en varios loops. Son dos formas de atacar un problema. Este nuevo concepto es muy interesante y abre muchas posibilidades a nuestros programas que requieren conexiones remotas.

Multitask en Arduino, muy recomendable:

- https://learn.adafruit.com/multi-tasking-the-arduino-part-1
- https://learn.adafruit.com/multi-tasking-the-arduino-part-1/using-millis-for-timing
- http://harteware.blogspot.com.es/2010/11/protothread-powerfull-library.html

Ejercicios Interrupciones Arduino

Hacer un ejercicio simple donde asociemos al pin 2 una interrupción que encienda y apague el pin 13.

Código:

```
1
2   const byte ledPin = 13;
3   const byte interruptPin = 2;
4   volatile byte state = LOW;
5
6   void setup() {
7   pinMode(ledPin, OUTPUT);
8   pinMode(interruptPin, INPUT_PULLUP);
9   attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);
10  }
11
12  void loop() {
13  digitalWrite(ledPin, state);
14  }
15
16  void blink() {
17  state = !state;
18  }
19
```

En el siguiente código definimos el pin digital 10 como salida, para emular una onda cuadrada de periodo 300ms (150ms ON y 150ms OFF). Para visualizar el funcionamiento de la interrupción, en cada flanco activo del pulso simulado, encendemos/apagamos el LED integrado en la placa, por lo que el LED parpadea a intervalos de 600ms (300ms ON y 300ms OFF). No estamos encendiendo el LED con una salida digital, si no que es la interrupción que salta la que enciende y apaga el LED (el pin digital solo emula una señal externa).

```
1   const int emuPin = 10;
2
3   const int LEDPin = 13;
4   const int intPin = 2;
5   volatile int state = LOW;
6
7   void setup() {
8       pinMode(emuPin, OUTPUT);
9       pinMode(LEDPin, OUTPUT);
10      pinMode(intPin, INPUT_PULLUP);
11      attachInterrupt(digitalPinToInterrupt(intPin), blink, RISING);
12  }
13
14  void loop() {
15      //esta parte es para emular la salida
16      digitalWrite(emuPin, HIGH);
17      delay(150);
18      digitalWrite(emuPin, LOW);
19      delay(150);
20  }
21
22  void blink() {
23      state = !state;
24      digitalWrite(LEDPin, state);
25  }
```

El siguiente código empleamos el mismo pin digital para emular una onda cuadrada, esta vez de intervalo 2s (1s ON y 1s OFF). En cada interrupción actualizamos el valor de un contador. Posteriormente, en el bucle principal, comprobamos el valor del contador, y si ha sido modificado mostramos el nuevo valor. Al ejecutar el código, veremos que en el monitor serie se imprimen números consecutivos a intervalos de dos segundos.

```
const int emuPin = 10;

const int intPin = 2;
volatile int ISRCounter = 0;
int counter = 0;


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

    pinMode(intPin, INPUT_PULLUP);
    Serial.begin(9600);
    attachInterrupt(digitalPinToInterrupt(intPin), interruptCount, LOW);
}

void loop()
{
    //esta parte es para emular la salida
    digitalWrite(emuPin, HIGH);
    delay(1000);
    digitalWrite(emuPin, LOW);
    delay(1000);


    if (counter != ISRCounter)
    {
        counter = ISRCounter;
        Serial.println(counter);
    }
}

void interruptCount()
{
    ISRCounter++;
    timeCounter = millis();
}
```

Esta entrada se publicó en Arduino, Interrupciones y está etiquetada con Arduino, Interrupciones, ISR, Multitarea, timer en 13 noviembre, 2016.

.