**Periféricos AVR de 8 bits**

**Descripción general del oscilador megaAVR**

Los microcontroladores de 8 bits megaAVR® de Microchip tienen varias opciones de fuente de reloj, seleccionables a través de la programación de los bits de fusible **CKSEL**[**Flash**](https://microchipdeveloper.com/8avr:avrfuses) . Esta discusión es específica de la MCU [ATmega328PB .](http://www.microchip.com/wwwproducts/en/ATmega328PB)Los bits de fusible pueden seleccionar uno de:

* Oscilador de cristal de baja potencia
* Oscilador de cristal de baja frecuencia
* Oscilador RC interno de 128 kHz
* Oscilador RC interno calibrado, y
* Reloj externo.

La **fuente del reloj del sistema** **no se** puede cambiar durante el tiempo de ejecución, ya que se configura a través de la programación de fusibles.

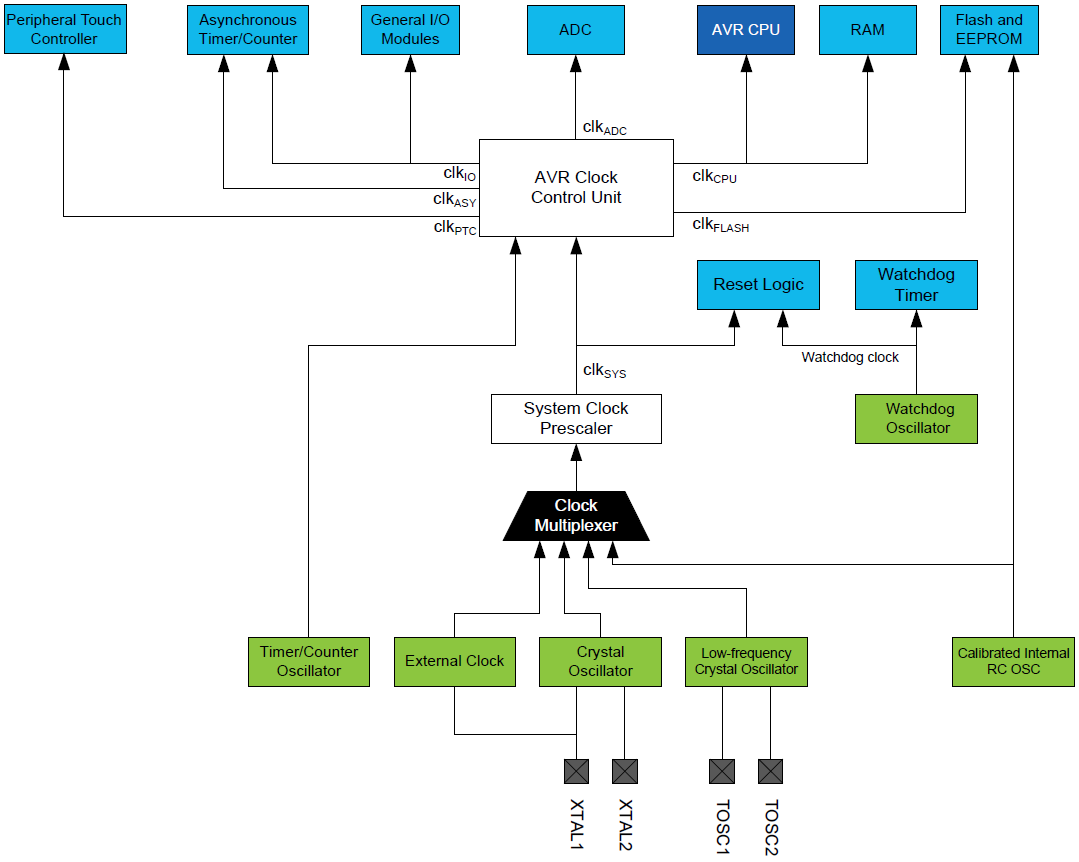
La **frecuencia del reloj del sistema** **se** puede cambiar durante el tiempo de ejecución escribiendo en el registro del preescalador del [**reloj del sistema**](https://microchipdeveloper.com/8avr:osc-mega-overview#system-clock-prescaler) (CLKPR).

Cada fuente de reloj proporciona una opción de retraso después del reinicio o encendido del dispositivo para mantener el dispositivo reiniciado hasta que se suministre con Vcc mínimo. El reloj de la fuente seleccionada se ingresa al generador de reloj AVR® y se enruta a los módulos apropiados.

**La frecuencia operativa máxima del megaAVR® depende de V CC** . El software de la aplicación debe garantizar que la frecuencia de la fuente de reloj seleccionada se encuentre dentro del área de operación segura (consulte la sección 33.4 en la [**hoja de datos del dispositivo**](http://ww1.microchip.com/downloads/en/DeviceDoc/40001906A.pdf) ).

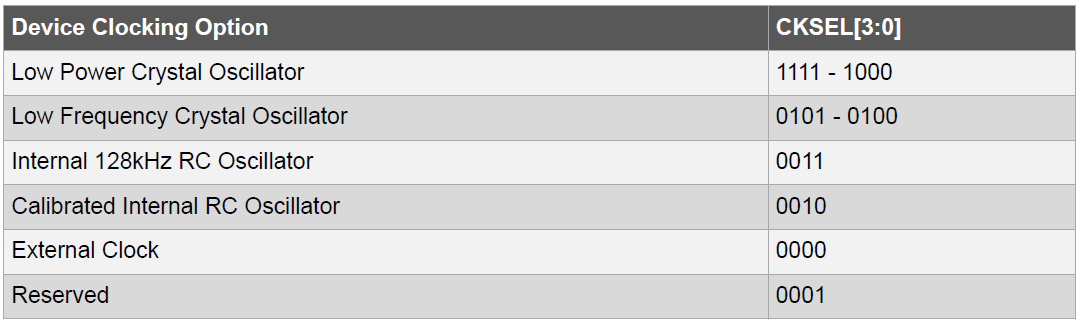
**Visión general**

La siguiente figura ilustra los principales sistemas de reloj del dispositivo y su distribución. No es necesario que todos los relojes estén activos en un momento dado. Para reducir el consumo de energía, los relojes de los módulos que no se utilizan se pueden detener utilizando diferentes [modos de suspensión](https://microchipdeveloper.com/8avr:avrsleep) . Los sistemas de reloj se describen en las siguientes secciones. La frecuencia del reloj del sistema se refiere a la frecuencia generada por el preescalador del reloj del sistema. Todas las salidas de reloj de la unidad de control de reloj AVR funcionan a la misma frecuencia.



**Fuentes de reloj**

El dispositivo tiene las siguientes opciones de fuente de reloj, seleccionables a través de los bits **CKSEL** Flash Fuse como se muestra a continuación. El reloj de la fuente seleccionada se ingresa al generador de reloj AVR® y se enruta a los módulos apropiados.



## **Fuente de reloj predeterminada**

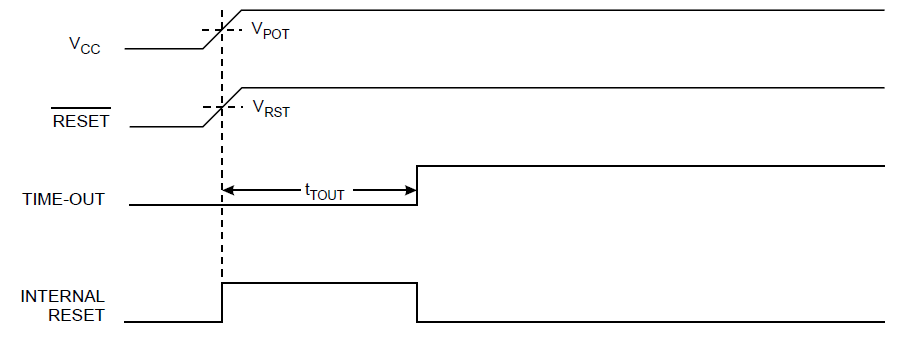
El dispositivo se envía con el oscilador RC interno seleccionado a 8,0 MHz y con el fusible CKDIV8 programado, lo que da como resultado un reloj del sistema de 1,0 MHz. El tiempo de inicio se establece al máximo y el período de tiempo de espera está habilitado: CKSEL=0010, SUT=10, CKDIV8=0. Esta configuración predeterminada garantiza que todos los usuarios puedan configurar la fuente de reloj deseada utilizando cualquier interfaz de programación disponible.

## Secuencia de inicio del reloj

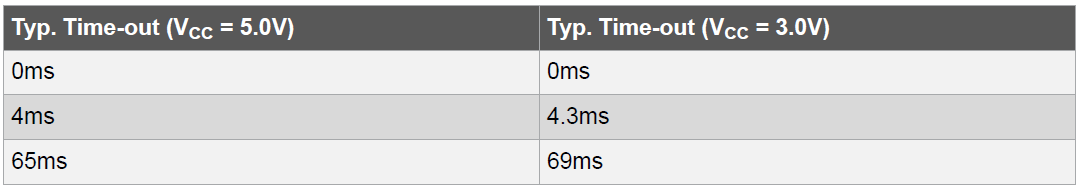
Cualquier fuente de reloj necesita (i) **un V CC suficiente para comenzar a oscilar** y (ii) **un número mínimo de ciclos de oscilación antes de que pueda considerarse estable** .

### **Estabilidad Vcc**

Para garantizar suficiente VCC , el dispositivo emite un restablecimiento interno con un retraso de tiempo de espera ( **t TOUT** ) después de que todas las demás fuentes de restablecimiento liberan el restablecimiento del dispositivo:



El retardo ( **t TOUT** ) se cronometra desde el oscilador de vigilancia y el tiempo de retardo se establece mediante los bits de fusible SUTx y CKSELx. Los retardos seleccionables para t TOUT se muestran en la siguiente tabla. Tenga en cuenta que la frecuencia del Watchdog Oscillator depende del voltaje:



V CC no se controla durante el retraso, por lo que se requiere seleccionar un retraso más largo que el tiempo de subida de V CC .Si esto no es posible, se debe utilizar un circuito de detección de Brown-Out (BOD) interno o externo. Un circuito BOD garantizará suficiente V CC antes de liberar el reinicio, y el retardo de tiempo de espera se puede desactivar. No se recomienda deshabilitar el retardo de tiempo de espera sin utilizar un circuito de detección de Brown-Out.

El Fusible CKSEL0 junto con los Fusibles SUT[1:0] seleccionan los tiempos de arranque (ver sección 11.3 en la [**hoja de datos del dispositivo**](http://ww1.microchip.com/downloads/en/DeviceDoc/40001906A.pdf) ).

## **Oscilador de cristal de baja frecuencia**

El oscilador de cristal de baja frecuencia está optimizado para su uso con un cristal de reloj de 32,768 kHz. El oscilador de cristal de baja frecuencia debe seleccionarse configurando los fusibles CKSEL en '0110' o '0111', y los tiempos de inicio están determinados por el SUT fusibles

## **Oscilador RC interno calibrado**

De forma predeterminada, el oscilador RC interno proporciona un reloj de 8,0 MHz. Aunque depende del voltaje y la temperatura, el usuario puede calibrar este reloj con mucha precisión. El dispositivo se envía con el fusible CKDIV8 programado, lo que proporciona una frecuencia de reloj del sistema de 1 MHz. Este reloj se puede seleccionar como el reloj del sistema programando los fusibles CKSEL en '0010':. Si se selecciona, funcionará sin componentes externos. Durante el reinicio, el hardware carga el valor de calibración preprogramado en el registro OSCCAL y, por lo tanto, calibra automáticamente el oscilador RC.

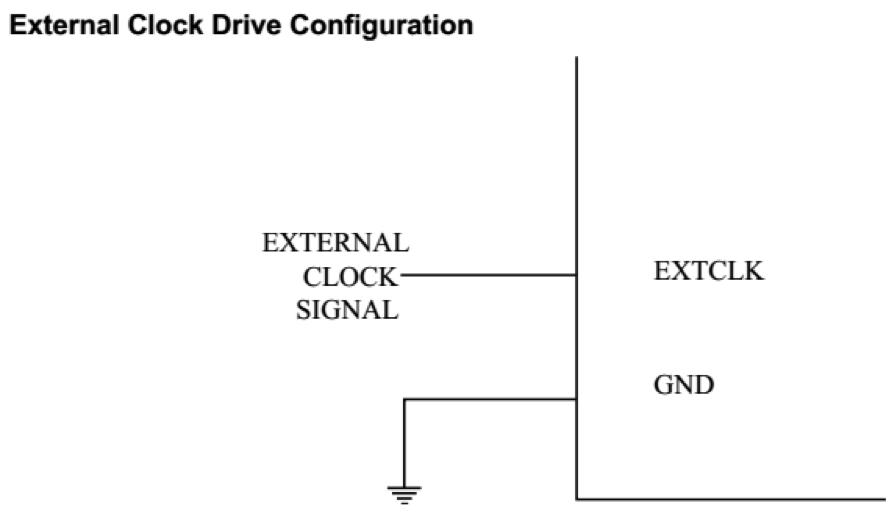
Consulte la [**nota de aplicación AVR053**](http://ww1.microchip.com/downloads/en/AppNotes/Atmel-2555-Internal-RC-Oscillator-Calibration-for-tinyAVR-and-megaAVR-Devices_ApplicationNote_AVR053.pdf) que describe el procedimiento para volver a calibrar el oscilador RC interno.

## **Oscilador interno de 128 kHz**

El oscilador interno de 128 kHz es un oscilador de baja potencia que proporciona un reloj de 128 kHz. Este reloj se puede seleccionar como reloj del sistema programando los fusibles CKSEL en '0011'.

## **Reloj externo**

Para controlar el dispositivo desde una fuente de reloj externa, EXTCLK debe controlarse como se muestra en la figura a continuación. Para ejecutar el dispositivo en un reloj externo, los fusibles CKSEL deben programarse en '0000'.



## **Búfer de salida de reloj**

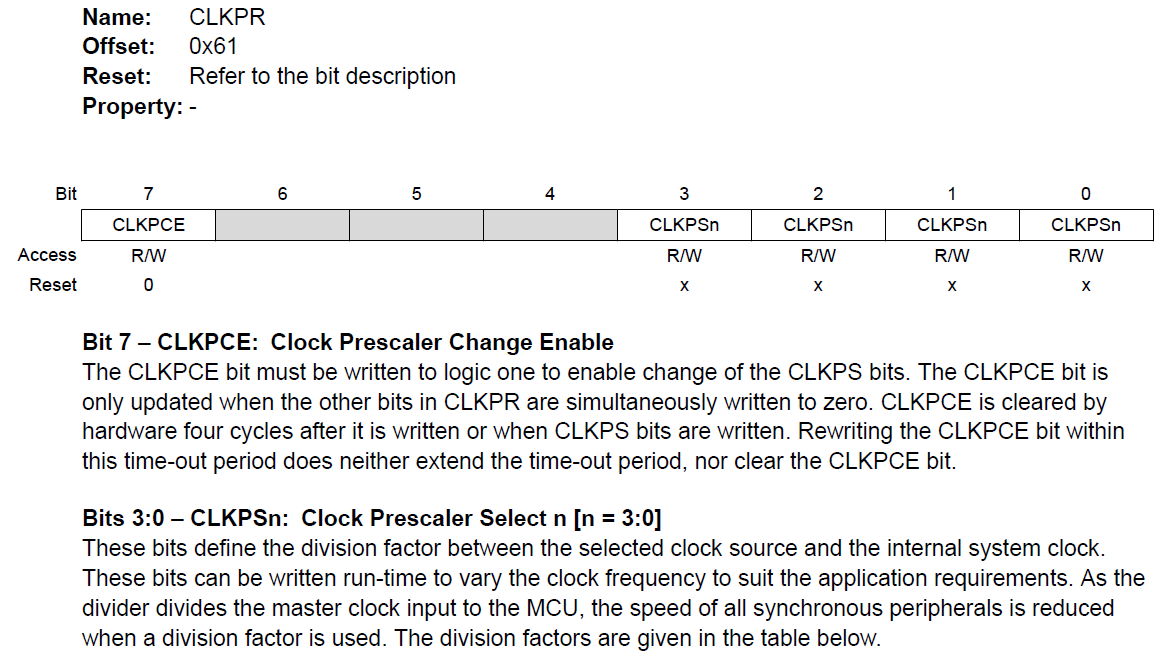
El dispositivo puede generar el reloj del sistema en el pin CLKO. Para habilitar la salida, se debe programar el Fusible CKOUT. Este modo es adecuado cuando el reloj del chip se usa para controlar otros circuitos en el sistema. El reloj también se emitirá durante el reinicio, y el funcionamiento normal del pin de E/S se anulará cuando se programe el fusible. Cualquier fuente de reloj, incluido el oscilador RC interno, se puede seleccionar cuando el reloj se emite en CLKO. Si se utiliza el preescalador de reloj del sistema, lo que se emite es el reloj del sistema dividido.

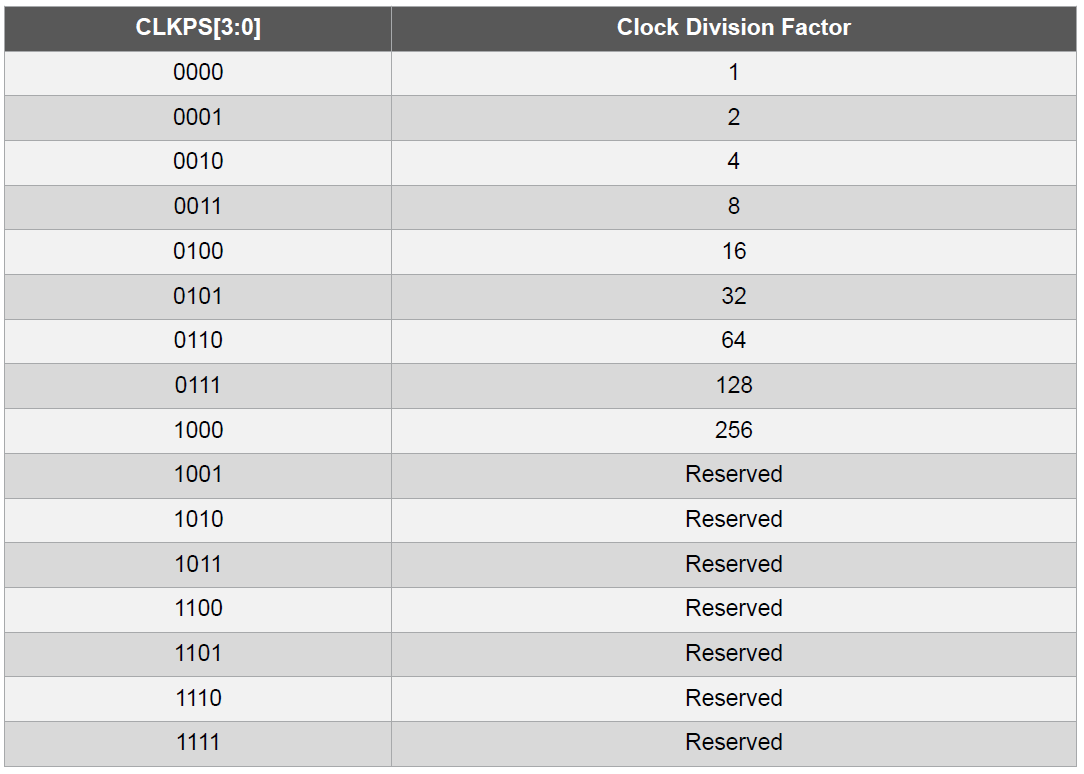
## **Temporizador/Contador Oscilador**

El dispositivo utiliza el mismo oscilador de cristal para el oscilador de baja frecuencia y el temporizador/contador de osciladores. Consulte Oscilador de cristal de baja frecuencia para obtener detalles sobre los requisitos del oscilador y del cristal.  
  
En este dispositivo, los pines del oscilador del temporizador/contador (TOSC1 y TOSC2) se comparten con EXTCLK. Cuando se utiliza el temporizador/contador de osciladores, el reloj del sistema debe ser cuatro veces la frecuencia del oscilador. Debido a esto y al uso compartido de pines, el temporizador/contador de osciladores solo se puede usar cuando el oscilador RC interno calibrado se selecciona como fuente de reloj del sistema. Se puede aplicar una fuente de reloj externa a TOSC1 si el bit Habilitar entrada de reloj externo en el registro de estado asíncrono (ASSR.EXCLK) se escribe en '1'. Consulte la descripción de la operación asíncrona del temporizador/contador2 para obtener una descripción más detallada sobre la selección de un reloj externo como entrada en lugar de un cristal de reloj de 32,768 kHz.

# Prescaler del reloj del sistema

El dispositivo tiene un preescalador de reloj del sistema, y ​​el reloj del sistema se puede dividir configurando el Registro de preescala de reloj (CLKPR). Esta función se puede utilizar para disminuir la frecuencia del reloj del sistema y el consumo de energía cuando el requisito de potencia de procesamiento es bajo. Esto se puede usar con todas las opciones de fuente de reloj y afectará la frecuencia de reloj de la CPU y todos los periféricos síncronos. clk I/O , clk ADC , clk CPU y clk FLASH se dividen por un factor como se muestra en la descripción de CLKPR:





## **Escribiendo a CLKPR**

Al cambiar entre las configuraciones del preescalador, el Preescalador del reloj del sistema asegura que no ocurran fallas en el sistema del reloj. También asegura que ninguna frecuencia intermedia sea superior a la frecuencia de reloj correspondiente a la configuración anterior, ni a la frecuencia de reloj correspondiente a la nueva configuración. El contador de ondas que implementa el preescalador se ejecuta a la frecuencia del reloj indiviso, que puede ser más rápido que la frecuencia del reloj de la CPU. Por lo tanto, no es posible determinar el estado del preescalador; incluso si fuera legible, el tiempo exacto que se tarda en cambiar de una división de reloj a otra no se puede predecir con exactitud. Desde el momento en que se escriben los valores de los bits de selección del preescalador de reloj (CLKPS[3:0]), transcurren entre T1 + T2 y T1 + 2 \* T2 antes de que se active la nueva frecuencia de reloj. En este intervalo, se producen dos flancos de reloj activos. Aquí, T1 es el período de reloj anterior y T2 es el período correspondiente a la nueva configuración del preescalador. Para evitar cambios involuntarios de la frecuencia del reloj, se debe seguir un procedimiento de escritura especial para cambiar los bits CLKPS:

1. Escriba el bit de habilitación de cambio de preescalador de reloj (CLKPCE) en '1' y todos los demás bits en CLKPR en cero: CLKPR=0x80.
2. Dentro de cuatro ciclos, escriba el valor deseado en CLKPS[3:0] mientras escribe un cero en CLKPCE: CLKPR=0x0N

Las interrupciones deben desactivarse al cambiar la configuración del preescalador para asegurarse de que el procedimiento de escritura no se interrumpa.

### **Ejemplo de código**

La siguiente función se puede utilizar para actualizar dinámicamente CLKPR como se requiere anteriormente. Tenga en cuenta el uso de las funciones cli() y sei() para garantizar que el procedimiento de escritura CLKPR no se interrumpa.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #include <stdint.h>  // Std integral type definitions  #include <avr/io.h>  // SFR definitions  #include <avr/interrupt.h> // ISR macros    **void** clkPrescaleSet(uint8\_t divisionFactor){      cli();                        // disable interrupts      CLKPR = (1 << CLKPCE);        // enable change of the CLKPSx bits      CLKPR = divisionFactor;       // update the CLKPSx bits      sei();                        // re-enable interrupts  } |

Para ver esta función en uso, visite el proyecto de ejemplo del [**oscilador megaAVR®**](https://microchipdeveloper.com/8avr:osc-mega-example)

## **CLKDIV8 Fusible y CLKPR**

El fusible CKDIV8 determina el valor inicial de los bits CLKPS. Si CKDIV8 no está programado, los bits CLKPS se restablecerán a "0000". Si se programa CKDIV8, los bits CLKPS se restablecen a "0011", dando un factor de división de 8 al inicio. Esta función debe utilizarse si la fuente de reloj seleccionada tiene una frecuencia superior a la frecuencia máxima del dispositivo en las condiciones de funcionamiento actuales. Tenga en cuenta que se puede escribir cualquier valor en los bits CLKPS independientemente de la configuración del fusible CKDIV8. El software de la aplicación debe garantizar que se elija un factor de división suficiente si la fuente de reloj seleccionada tiene una frecuencia superior a la frecuencia máxima del dispositivo en las condiciones de funcionamiento actuales. El dispositivo se envía con el fusible CKDIV8 programado.

**Descripción general de AVR® USART**

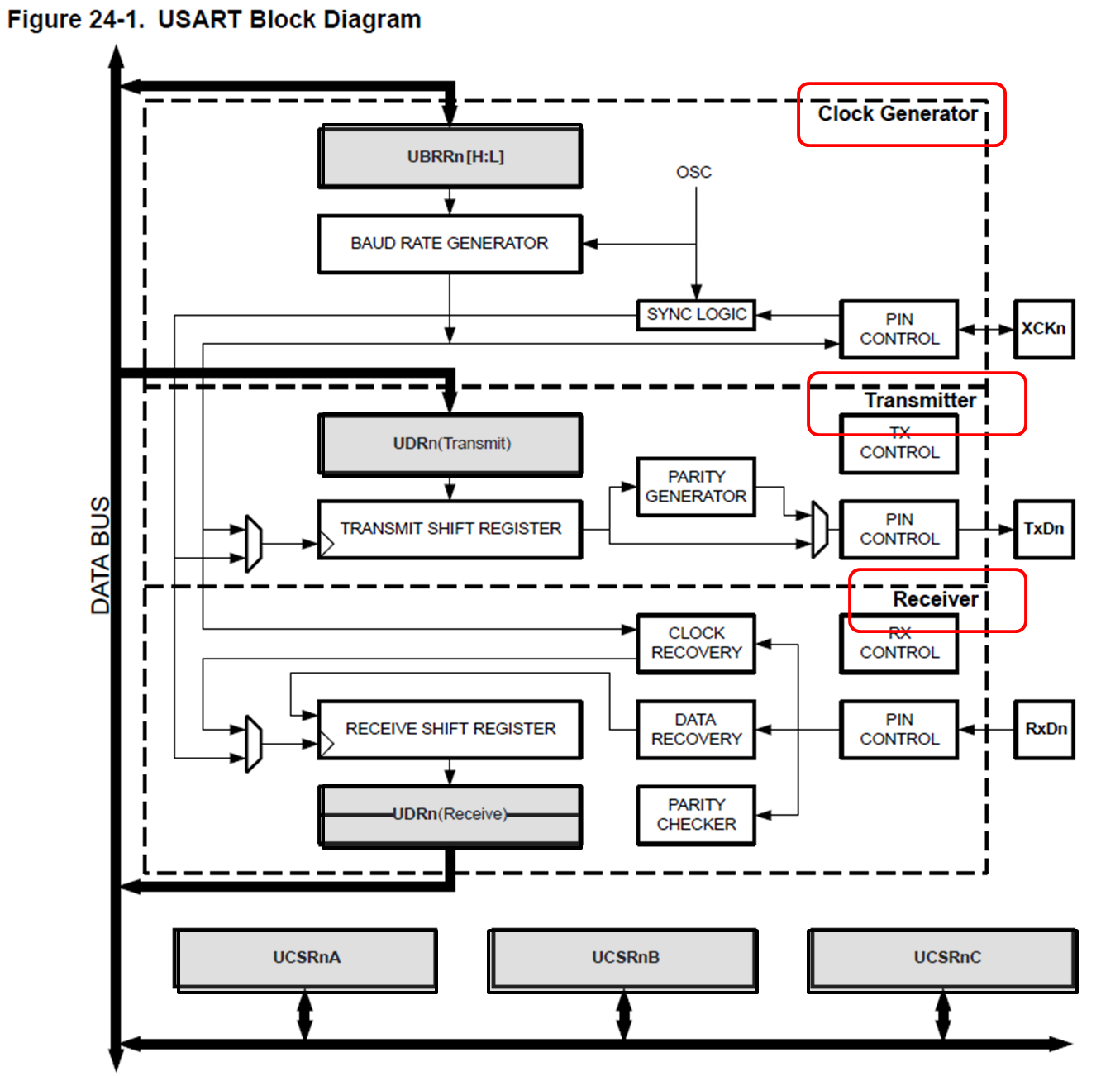
Los microcontroladores Microchip AVR® de 8 bits contienen un periférico de comunicación altamente flexible conocido como **USART** (receptor y transmisor serie universal síncrono y asíncrono). Este periférico se puede utilizar para comunicarse con una amplia variedad de otros componentes, incluidos otros microcontroladores, módulos inalámbricos, pantallas LCD, módulos GPS, etc. El periférico USART puede funcionar en uno de dos modos principales: sincrónico o asincrónico. Este módulo se centra en el [**modo de funcionamiento asíncrono**](https://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter) .

**Configuración megaAVR® USART**

En esta sección, cubriremos los pasos básicos de codificación necesarios para configurar/usar el módulo USART en un MCU megaAVR®, con un enfoque en el dispositivo [**ATmega328PB .**](http://www.microchip.com/wwwproducts/en/ATmega328PB)

# Visión general

El módulo USART consta de tres secciones principales, como se muestra en el siguiente diagrama: **generador de reloj** , **transmisor** y **receptor** .



Los registros clave (resaltados en gris) incluyen:

* Registros de control y estado ( **UCSRnA** , **UCSRnB** , **UCSRnC** ) compartidos por las tres secciones.
* Registro de datos **UDRn** compartido por las secciones Transmisor y Receptor.
* Registros de control de velocidad en baudios **UBRRn[H:L]** utilizados por el generador de reloj.

" **n** " en el nombre del registro/bit identifica la instancia de hardware USART específica (0, 1, 2) a la que está asociado el registro/bit. Por ejemplo **, UCSR0A** se refiere a **USART0** Control & Status Register **A**

## **Usando el USART (Resumen)**

Para la operación sondeada básica, se deben realizar los siguientes pasos mínimos:

1. Elija una tasa de baudios y programe los **registros UBRRn[H:L]** en consecuencia.
2. Habilite las secciones de transmisión y recepción en serie de USART.
3. Si está transmitiendo, espere hasta que el registro de desplazamiento de transmisión esté vacío (sondee en **UCSRnA.UDREn** ), luego cargue su byte de datos en **UDRn** .
4. Si recibe, espere hasta que se establezca el bit de recepción de datos del receptor (sondee en **UCSRnA.RXCn** ), luego lea los datos de **UDRn** . La lectura de UDRn borra automáticamente el bit y prepara el hardware para recibir el siguiente byte.

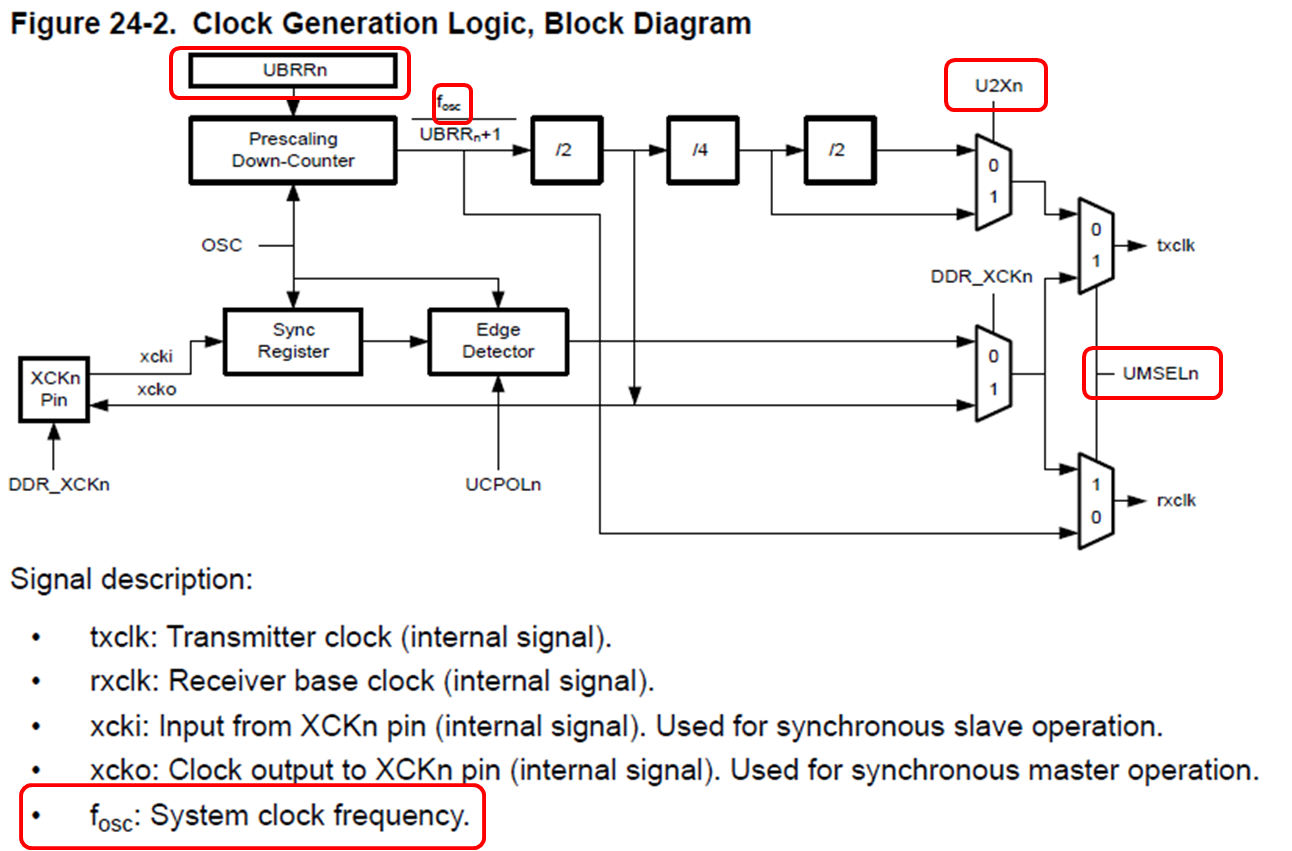
# Inicialización

El USART debe inicializarse antes de que pueda tener lugar cualquier comunicación. El proceso de inicialización normalmente consiste en:

* Configuración de la velocidad en baudios,
* Configuración del formato de cuadro y
* Habilitación del Transmisor o del Receptor según el uso.

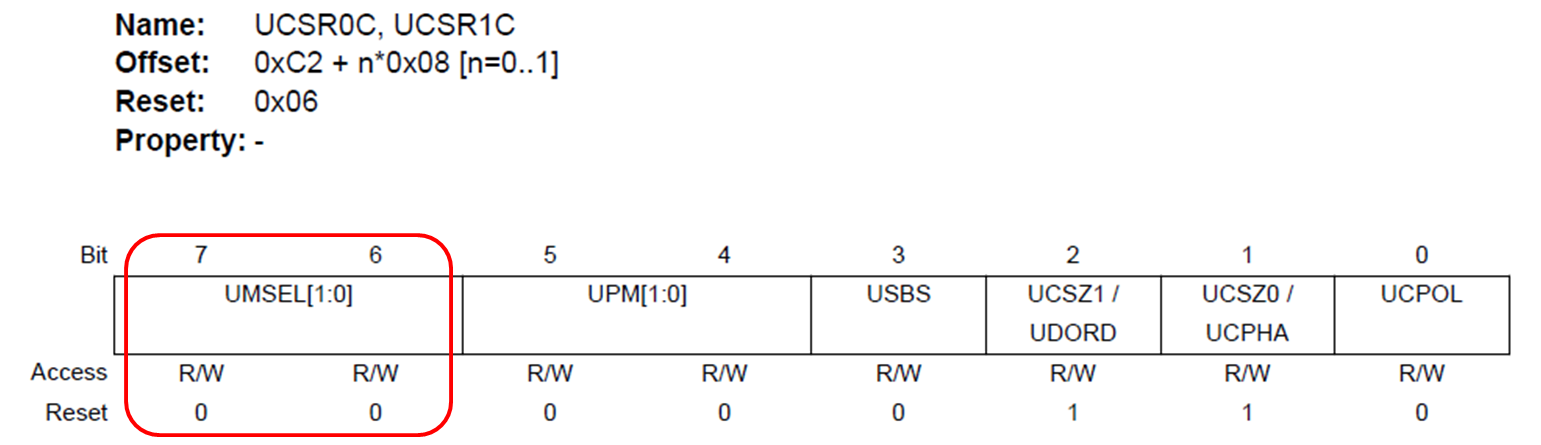
## **Configuración de la tasa de baudios**

La generación de reloj interno se utiliza para el modo de funcionamiento asíncrono. La lógica de generación de reloj genera el reloj base para el transmisor y el receptor (los registros clave y los bits de control están resaltados):



### **Selección de modo USART (UMSELn)**

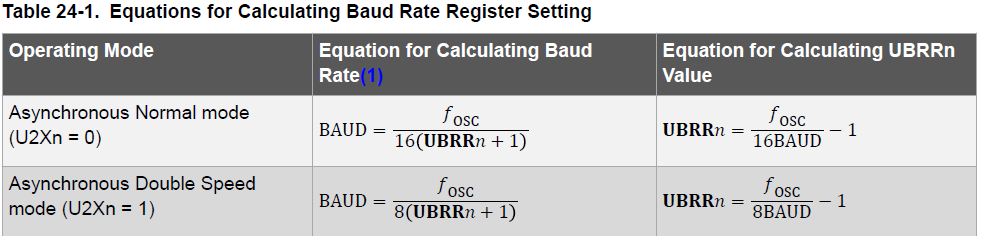
# La ecuación de velocidad en baudios utilizada por el módulo se establece en función del modo de funcionamiento. Para la operación en modo asíncrono, los bits de selección de modo USART en el registro C de control y estado de USART ( ****UCSRnC.UMSELn[1:0]**** ) se utilizan para seleccionar ****la operación asíncrona**** ( ****UMSEL[1:0] = 00**** ) como se muestra:



Con el modo de velocidad doble configurado, el receptor solo usará la mitad del número de muestras (reducidas de 16 a 8) para el muestreo de datos y la recuperación del reloj y, por lo tanto, se requiere una configuración de velocidad en baudios y un reloj del sistema más precisos cuando se usa este modo.

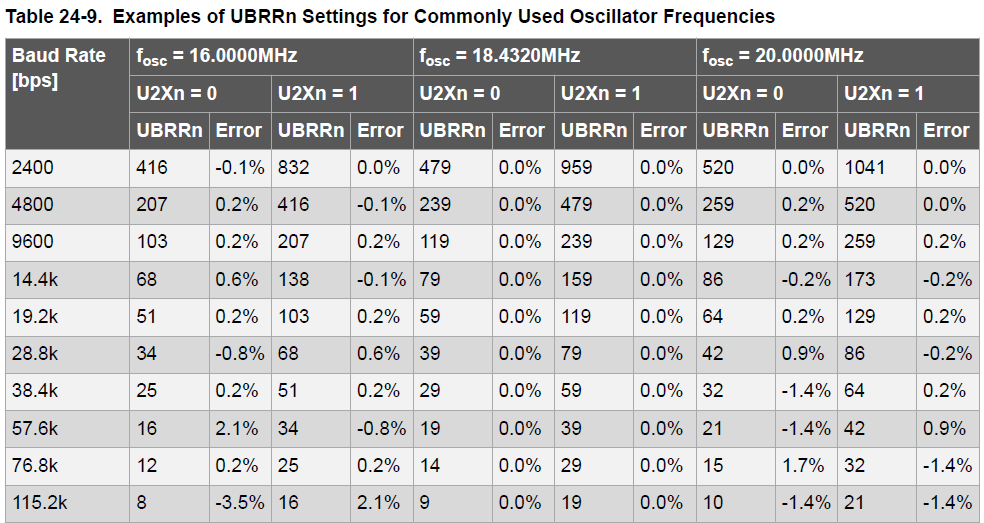
### **Registro de velocidad en baudios (UBRRn)**

El registro de tasa de baudios USART ( **UBRRn** ) y el contador descendente conectado a él funcionan como un preescalador programable o generador de tasa de baudios. El contador regresivo, que se ejecuta en el reloj del sistema (f osc ), se carga con el valor UBRRn cada vez que el contador llega a cero o cuando se escribe el registro UBRRnL. Se genera un reloj cada vez que el contador llega a cero. Este reloj es la salida del reloj del generador de velocidad en baudios (= f osc/(UBRRn+1)). El transmisor divide la salida del reloj del generador de velocidad en baudios por 2, 8 o 16 según el modo. La salida del generador de velocidad en baudios es utilizada directamente por el reloj del receptor y las unidades de recuperación de datos. Sin embargo, las unidades de recuperación usan una máquina de estado que usa 2, 8 o 16 estados según el modo establecido por el estado de los bits UMSEL, U2Xn y DDR\_XCK. La siguiente tabla contiene ecuaciones para calcular la tasa de baudios (en bits por segundo) y para calcular el valor UBRRn para cada modo de operación utilizando una fuente de reloj generada internamente.



La biblioteca [**AVR-LIBC Setbaud**](http://www.nongnu.org/avr-libc/user-manual/group__util__setbaud.html) contiene macros útiles para calcular los valores correctos para escribir en los registros UBRRnH y UBRRnL. Consulte [el ejemplo de código de inicialización a](https://microchipdeveloper.com/8avr:usart-mega-configuration#code-example) continuación.

También se proporcionan tablas en la hoja de datos del dispositivo que contienen valores UBRRn para tasas de baudios comunes, dadas varias frecuencias de oscilador:

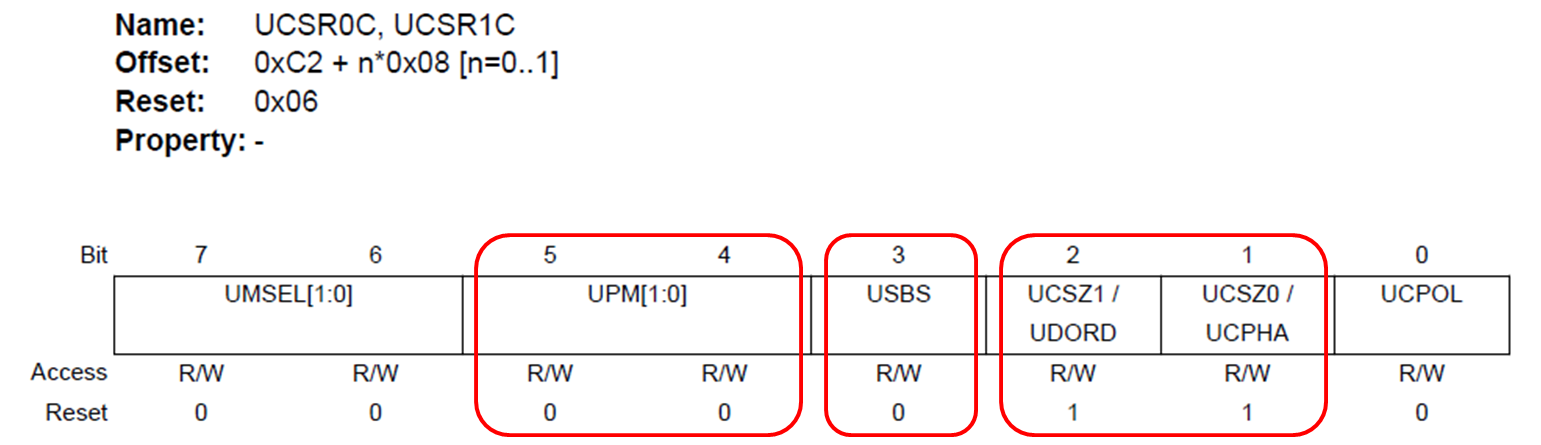


Para los cálculos de frecuencia en baudios, generalmente se acepta que los porcentajes de error de menos de ± 2% son aceptables.

## **Configuración del formato de marco**

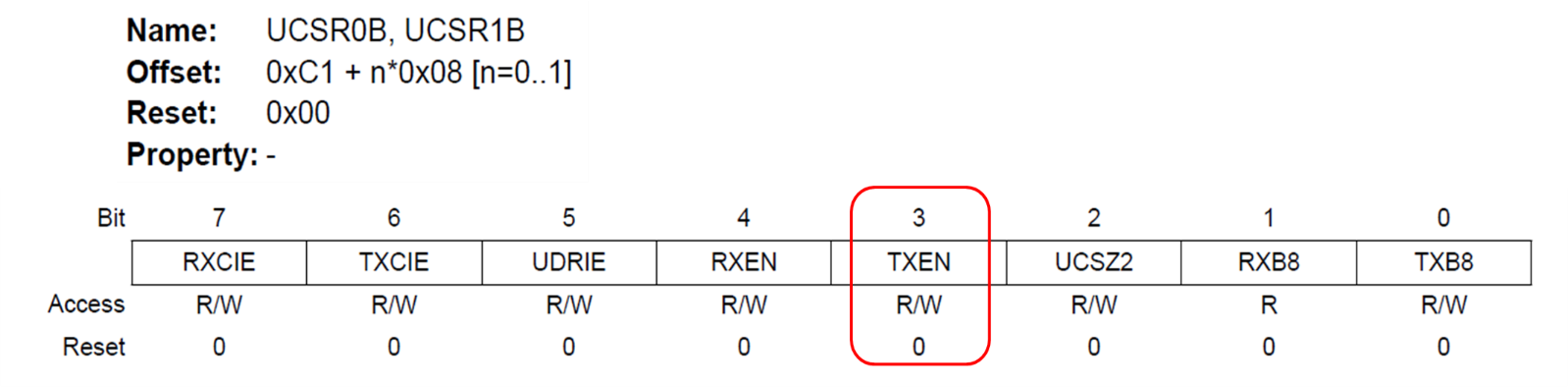
El registro C de control y estado de USART ( **UCSRnC** ) se utiliza para configurar el formato de la trama de comunicación UART: paridad, número de bits de parada y número de bits de datos. Los ajustes para el formato de cuadro típico “8N1” son los siguientes:

* **UPM[1:0] = 00** para Sin paridad
* **USBS = 0** para 1 bit de parada
* **UCSZ1[1:0]** = 11 para 8 Bits

[](https://microchipdeveloper.com/local--files/8avr:usart-mega-configuration/usart-frame-format-settings.png)

## **Habilitación del transmisor**

El transmisor USART se habilita configurando el bit de **habilitación de transmisión (TXEN)** en el registro **UCSRnB** :

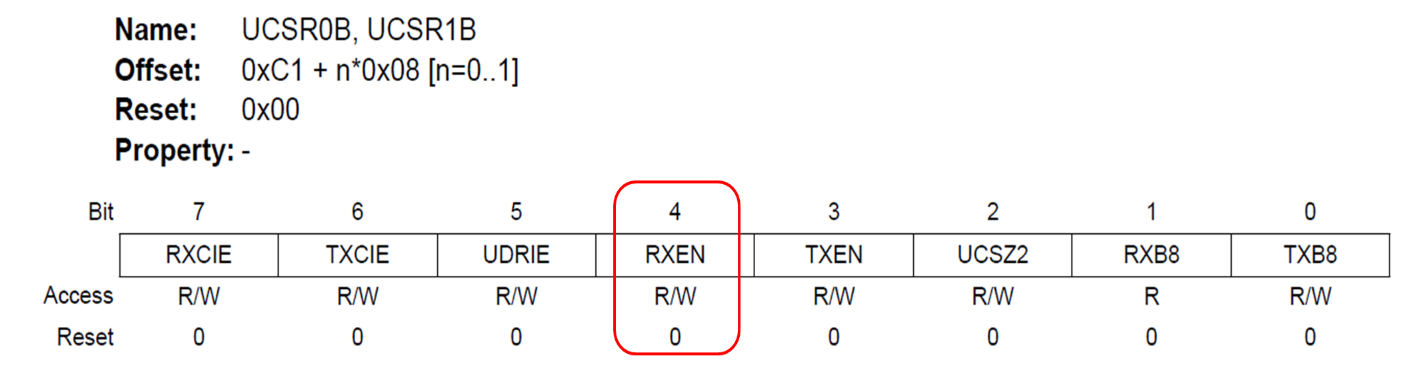
[](https://microchipdeveloper.com/local--files/8avr:usart-mega-configuration/usart-ucsrnb-txen.png)

Cuando el transmisor está habilitado, la operación normal del puerto del pin TxDn es anulada por el USART y se le asigna la función de salida en serie del transmisor.

La velocidad en baudios, el modo de operación y el formato de trama deben configurarse una vez antes de realizar cualquier transmisión.

## **Habilitación del receptor**

El receptor USART se habilita escribiendo el bit de **habilitación de recepción (RXEN)** en el registro **UCSRnB** a '1':

[](https://microchipdeveloper.com/local--files/8avr:usart-mega-configuration/usart-ucsrnb-rxen.png)

Cuando el Receptor está habilitado, el USART anula la operación normal del puerto del pin RxDn y se le asigna la función como entrada en serie del Receptor.

La velocidad en baudios, el modo de operación y el formato de trama deben configurarse una vez antes de realizar cualquier transmisión.

## **Ejemplo de código**

El siguiente ejemplo de código de inicialización de USART utiliza la biblioteca de [**utilidades setbaud**](http://www.nongnu.org/avr-libc/user-manual/group__util__setbaud.html) en AVR-LIBC. Esta biblioteca proporciona macros que usan el preprocesador c para calcular los valores apropiados para **UBBRn** . **Entradas** Este archivo de encabezado requiere que los valores de entrada ya estén definidos para **F\_CPU** y **BAUD** . Además, la macro BAUD\_TOL definirá la tolerancia de velocidad en baudios (en porcentaje) que es aceptable durante los cálculos. El valor de BAUD\_TOL por defecto será +/- 2%. **Salidas** Suponiendo que los BAUD solicitados son válidos para la F\_CPU dada, entonces la macro **UBRR\_VALUE** se establece en el valor del preescalador requerido. Se proporcionan dos macros adicionales para los bytes alto y bajo del preescalador, respectivamente: **UBRRL\_VALUE** se establece en el byte inferior de UBRR\_VALUE y **UBRRH\_VALUE** se establece en el byte superior. Se definirá una macro adicional **USE\_2X** . Su valor se establece en 1 si la tasa de BAUDIOS deseada dentro de la tolerancia dada solo se puede lograr al establecer el bit **U2Xn** en la configuración de UART. Se definirá a 0 si no se necesita **U2Xn** .

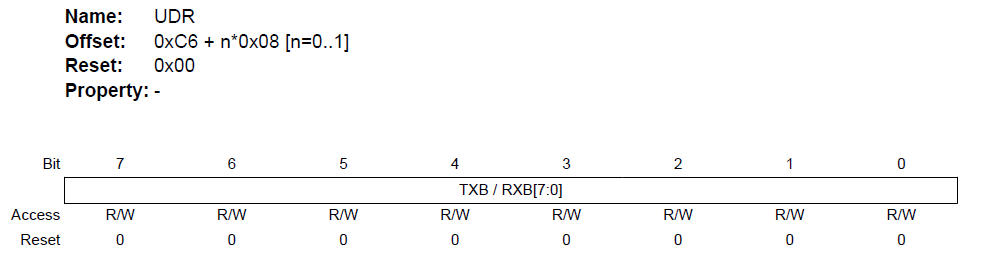
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  dieciséis  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | <font></font>  #define F\_CPU 16000000UL            // required for setbaud & other libraries<font></font>  #define BAUD 38400UL                // desired baud<font></font>  #define BAUD\_TOL 2                  // desired baud rate tolerance (+/- %)<font></font>  <font></font>  #include <avr/io.h><font></font>  #include <util/setbaud.h><font></font>  <font></font>  **void** USART0\_Init(**void**){<font></font>  <font></font>      // Set the BAUD rate<font></font>  <font></font>      UBRR0H = UBRRH\_VALUE;<font></font>      UBRR0L = UBRRL\_VALUE;<font></font>      #if USE\_2X                       // USE\_2X defined by setbaud.h based on inputs<font></font>      UCSR0A |= (1 << U2X0);<font></font>      #else<font></font>      UCSR0A &= ~(1 << U2X0);<font></font>      #endif<font></font>  <font></font>      // Set the Mode & Frame Parameters<font></font>  <font></font>      UCSR0C = 0x06;                   // Asynchronous, 8-data, No parity, 1-stop<font></font>  <font></font>      // Enable USART0 Transmitter and Receiver<font></font>  <font></font>      UCSR0B = (1 << TXEN0) | (1 << RXEN0);<font></font>  <font></font>  }<font></font>  <font></font> |

La biblioteca setbaud genera mensajes de advertencia durante la compilación si los parámetros de entrada generan una configuración de tasa BAUD que producirá una tasa de baudios fuera del BAUD\_TOL deseado

# Transmisión de datos

## Transmitir

Una transmisión de datos se inicia cargando el búfer de transmisión con los datos a transmitir. La CPU puede cargar el búfer de transmisión escribiendo en el registro **UDRn** . Para la operación de sondeo, el firmware debe monitorear el indicador de registro de datos vacío ( **UCSRnA.UDREn** ) antes de cargar **UDRn** . Los datos almacenados en el búfer de transmisión se moverán al registro de desplazamiento cuando el registro de desplazamiento esté listo para enviar una nueva trama. El registro de desplazamiento se carga con nuevos datos si está en estado inactivo (sin transmisión en curso) o inmediatamente después de que se transmita el último bit de parada de la trama anterior. Cuando el registro de desplazamiento se carga con nuevos datos, transferirá un cuadro completo a la velocidad dada por el registro de baudios.



El indicador de interrupción de transmisión completa ( **USCRnA.TXCn** ) se establece y se puede generar una interrupción de TX opcional (si está habilitada) cuando se ha desplazado todo el marco en el registro de desplazamiento. El bit indicador **UCSRnA.TXCn** se borra automáticamente cuando se ejecuta una interrupción de transmisión completa, o se puede borrar escribiendo un uno en su ubicación de bit.

## Recibir

El receptor inicia la recepción de datos cuando detecta un bit de inicio válido. Cada bit que sigue al bit de inicio se muestreará a la velocidad en baudios o al reloj XCKn, y se desplazará al registro de desplazamiento de recepción hasta que se reciba el primer bit de parada de una trama. El búfer de recepción se puede leer leyendo el registro **UDRn** . La recepción completa de un byte se puede verificar sondeando el bit RXCn en **el registro UCSRnA** .

El indicador de interrupción de recepción completa ( **RXCn** ) se establece y se puede generar una interrupción de RX opcional (si está habilitada) cuando el cuadro completo en el registro de desplazamiento se ha copiado en el registro **UDRn** . Esta es una interrupción persistente , es decir, el firmware debe leer los datos recibidos de **UDRn** para borrar el indicador **RXCn**

## Ejemplo de código

Las siguientes API de bloqueo simple envían y reciben un byte de datos a través de USART0.

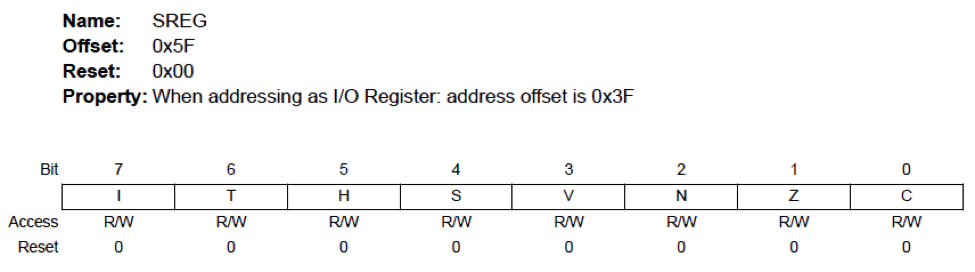
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  dieciséis  17  18  19  20  21 | <font></font>  **void** USART0\_Transmit(**unsigned** **char** data){<font></font>  <font></font>      // Wait for empty transmit buffer<font></font>  **while**(!(UCSR0A & (1 << UDRE0)));<font></font>  <font></font>      // Put data into buffer, sends the data<font></font>      UDR0 = data;<font></font>  <font></font>  }<font></font>  <font></font>  **unsigned** **char** USART0\_Receive(**void**){<font></font>  <font></font>      // Wait for data to be received<font></font>  **while**(!(UCSR0A & (1 << RXC0)));<font></font>  <font></font>      // Get and return received data from buffer<font></font>  **return** UDR0;<font></font>  <font></font>  }<font></font>  <font></font> |

**Resumen de interrupciones de megaAVR®**

La familia megaAVR® proporciona varias fuentes de interrupción diferentes, todas las cuales son enmascarables y se dividen en tres categorías:

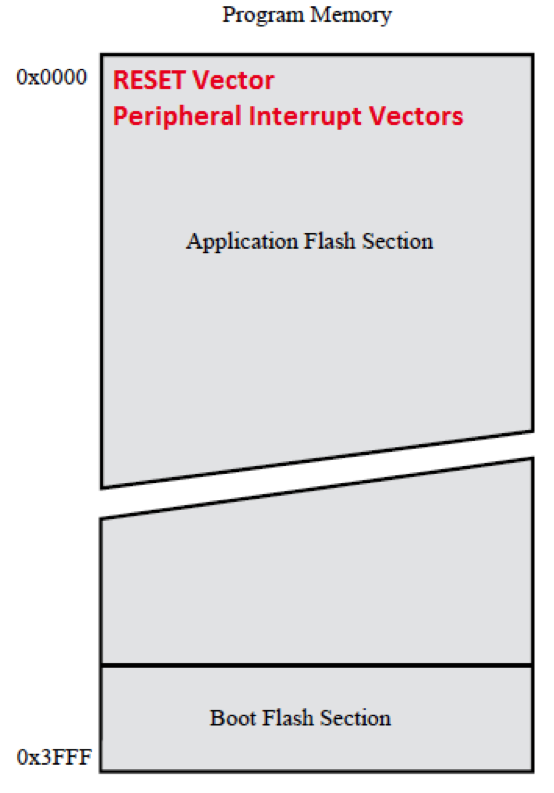
* **Interrupciones periféricas internas**
  + Asociado con temporizadores, USART, SPI, periféricos ADC
* **Interrupciones de clavijas externas**
  + Asociado con los pines de interrupción externa INT0-INT7
* **Interrupciones de cambio de pin**
  + Asociado con interrupciones externas PCINT0-PCINT2 que ocurren en un cambio de pin de puerto

A los periféricos se les asignan **bits de habilitación de interrupción** individuales en su respectivo **registro de máscara de interrupción** que debe escribirse como uno lógico junto con el **bit I de habilitación de interrupción global** en el **registro** de estado para habilitar la interrupción.



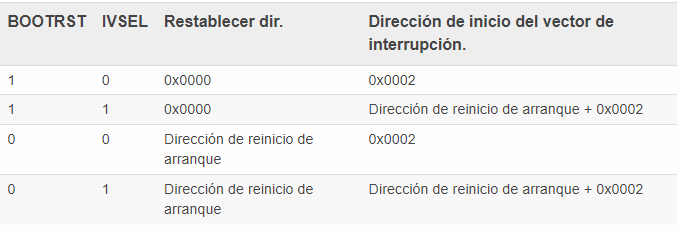
# Restablecer e interrumpir ubicaciones de vectores

Cada una de las fuentes de reinicio e interrupción tiene un vector de programa separado en el **espacio de memoria** del programa . Las direcciones más bajas en el espacio de la memoria del programa se definen de manera predeterminada como vectores de reinicio e interrupción, como se muestra:

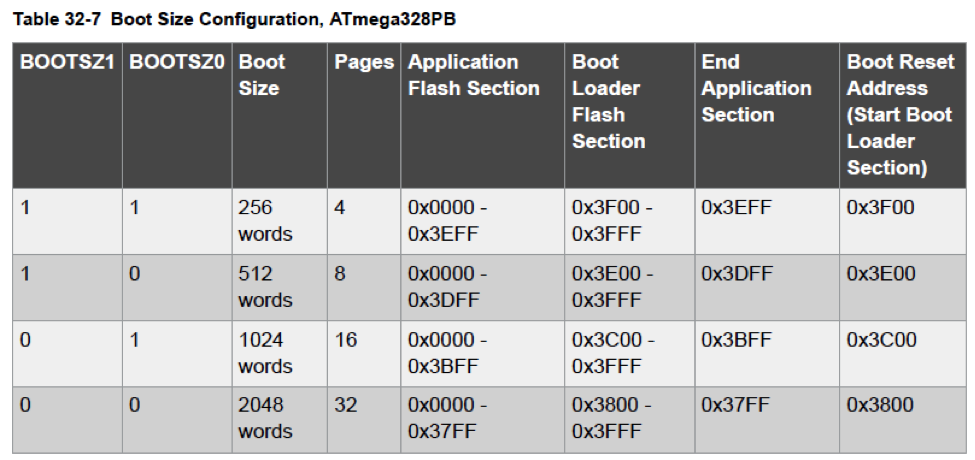


## Reubicación de vectores

El usuario puede reubicar el vector RESET así como la ubicación de inicio de los vectores de Interrupción en la **Sección Flash de Arranque** del espacio de la memoria del programa programando el bit de fusible **BOOTRST** en "0" y configurando el bit **IVSEL** del Registro de Configuración del Microcontrolador ( **MCUCR** ) en "1". Aquí se muestra la posible ubicación del vector de interrupción y RESET:



La **dirección de reinicio de arranque** se establece mediante bits de fusible BOOTSZ0/BOOTSZ1 como se muestra aquí para ATmega328PB:



Los fusibles se programan utilizando un [**procedimiento de programación especial**](https://microchipdeveloper.com/8avr:avrfuses) dentro de Atmel Studio 7 u otro programador.

Para evitar cambios no intencionales de las tablas de vectores de interrupción, se debe seguir un procedimiento de escritura especial para cambiar el bit IVSEL:

* Escriba el bit de habilitación de cambio de vector de interrupción (IVCE) a uno.
* Dentro de cuatro ciclos, escriba el valor deseado en IVSEL mientras escribe un cero en IVCE.

Aquí hay un ejemplo de código que muestra cómo modificar el bit IVSEL y reubicar los vectores de interrupción:

**void** move\_interrupts(**void**)

{

  uchar temp;

  /\* GET MCUCR \*/

  temp = MCUCR;

  /\* Enable change of Interrupt Vectors \*/

  MCUCR = temp | (1 << IVCE);

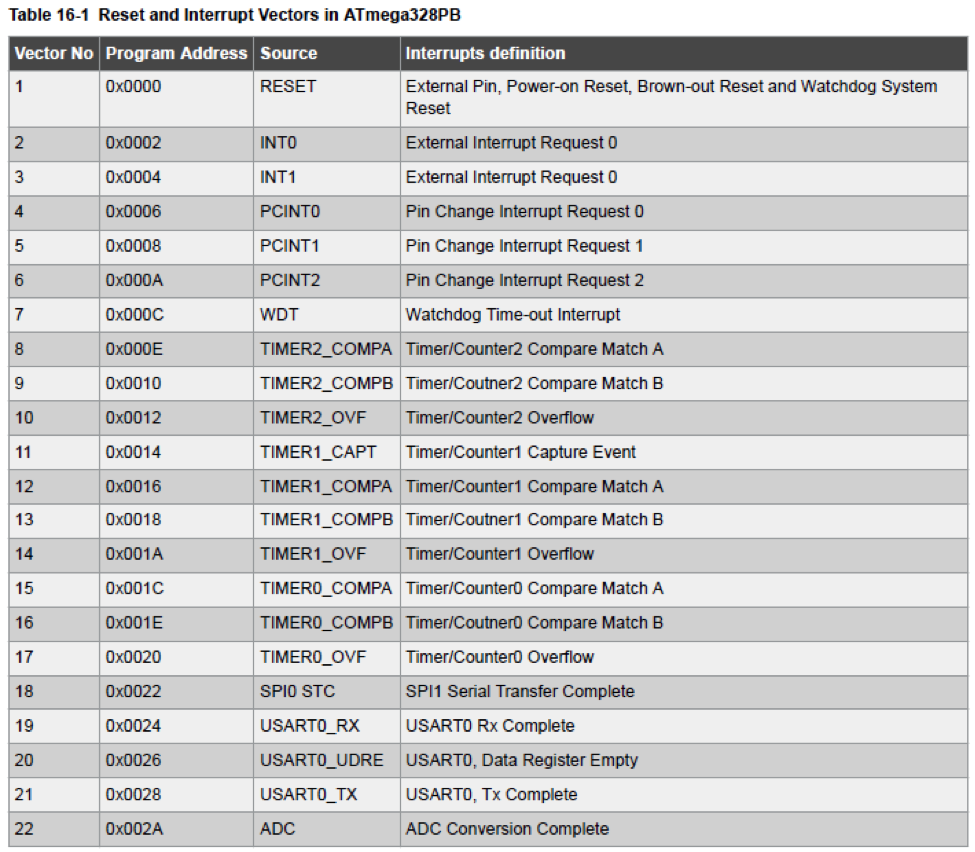
  /\* Move interrupts to Boot Flash section \*/

  MCUCR = temp | (1 << IVSEL);

}

# Nivel de prioridad

Cada vector tiene un nivel de prioridad predeterminado: cuanto **menor** sea la dirección, **mayor** será el nivel de prioridad. RESET tiene la prioridad más alta, y el siguiente es INT0: la solicitud de interrupción externa 0. El siguiente gráfico muestra la lista de vectores parciales para la MCU ATmega328PB:



# Procesamiento de interrupciones

Cuando ocurre una interrupción, el bit I de habilitación de interrupción global se borra y todas las interrupciones se desactivan. El bit I se establece automáticamente cuando se ejecuta una instrucción Return from Interrupt (RETI).

El software del usuario puede escribir uno lógico en el bit I para habilitar **interrupciones anidadas** . Todas las interrupciones habilitadas pueden interrumpir la rutina de interrupción actual.

## Interrupciones persistentes

Este tipo de interrupción se activará siempre que la condición de interrupción esté presente. Estas interrupciones no necesariamente tienen banderas de interrupción. **Ejemplo: Interrupción de recepción completa de** USART El USART contiene un indicador de recepción completa (RXC) que se establece si hay datos no leídos en el búfer de recepción. Cuando se establece la habilitación de interrupción de recepción completa (RXCIE) en UCSRnB, la interrupción de recepción completa de USART se ejecutará siempre que el indicador RXC esté establecido (siempre que las interrupciones globales estén habilitadas). Cuando se utiliza la recepción de datos impulsada por interrupciones, la rutina de recepción completa debe leer los datos recibidos de UDR para borrar el indicador RXC; de lo contrario, se producirá una nueva interrupción una vez que finalice la rutina de interrupción.

## Interrupciones no persistentes

Este tipo de interrupción se desencadena por un evento que establece un **indicador de interrupción** . Para estas interrupciones, el contador de programa se vectoriza al vector de interrupción real para ejecutar la rutina de manejo de interrupciones, y **el hardware borra el indicador de interrupción correspondiente .**. Las banderas de interrupción también se pueden borrar escribiendo un uno lógico en la(s) posición(es) del bit de bandera que se va a borrar. Si se produce una condición de interrupción mientras se borra el bit de activación de interrupción correspondiente, el indicador de interrupción se establecerá y se recordará hasta que se habilite la interrupción o el software borre el indicador. De manera similar, si ocurren una o más condiciones de interrupción mientras se borra el bit de habilitación de interrupción global, los indicadores de interrupción correspondientes se establecerán y recordarán hasta que se establezca el bit de habilitación de interrupción global, y luego se ejecutarán por orden de prioridad. **Ejemplo: Timer/Counter0 Overflow Interrupt** Bit-0 del Timer0 Interrupt Flag Register (TIFR0) contiene el indicador de interrupción TOV0. Este indicador se establece cuando se produce un desbordamiento en Timer/Counter0. TOV0 es **borrado por el hardware al ejecutar el vector de manejo de interrupción correspondiente** . Alternativamente, TOV0 se borra escribiendo un uno lógico en la bandera. Cuando se establecen el bit I de SREG, TOIE0 (habilitación de interrupción de desbordamiento del temporizador/contador0) y TOV0, se ejecuta la interrupción de desbordamiento del temporizador/contador0.

**Configuración de interrupciones megaAVR**

El desarrollador de la aplicación debe inicializar cuidadosamente la operación de interrupción de AVR®. Esta página resume los pasos clave de inicialización y uso necesarios para usar interrupciones en una aplicación. Se proporciona más información sobre el uso de interrupciones en la sección [**Módulo de interrupción de la biblioteca**](http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html)[**AVR-LIBC**](http://www.nongnu.org/avr-libc/) .

# Paso 1. ****#incluye**** encabezados estándar

La aplicación debe incluir archivos de encabezado avr/io.h y avr/interrupt.h como se muestra a continuación:

|  |  |
| --- | --- |
| 1  2 | #include <avr/io.h>  #include <avr/interrupt.h> |

El archivo de encabezado avr/interrupt.h proporciona varias macros destinadas a simplificar la aplicación de interrupciones en una aplicación, como macros para habilitar/deshabilitar interrupciones globalmente (bit I en el registro de estado), así como una macro para asignar una interrupción. función a un vector de interrupción específico:

* si( )
* CLI( )
* ISR(vector\_id, atributos)

Las macros **vector\_id** se definen en el archivo de encabezado específico del procesador (incluido a través de avr/io.h ), así como en la hoja de datos del dispositivo. Su construcción [se define a](https://microchipdeveloper.com/8avr:interrupts-mega-configuration#vector-id) continuación.

**Paso 2. Proporcionar rutina de servicio de interrupción**

Una función de manejo de interrupciones es diferente a una función ordinaria en que maneja el contexto guardar y restaurar para asegurar que al regresar de la interrupción, se mantenga el contexto del programa. También se usa una secuencia de código diferente para regresar de estas funciones.

Hay varias acciones que el compilador debe realizar para generar una rutina de servicio de interrupción:

* Se debe indicar al compilador que use una forma alternativa de instrucción de retorno ( RETI vs. RET )
* Se debe informar al compilador sobre cualquier opción adicional específica
  + Habilitar el anidamiento de interrupciones
  + Opciones para la generación de código de prólogo/epílogo
* La función debe vincularse a un vector de interrupción específico.

Se proporcionan varios atributos de función de controlador al desarrollador de la aplicación, lo que habilita estas opciones.

* La macro ISR( ) se proporciona para facilitar la definición de funciones de manejo de interrupciones con atributos

Para todos los vectores de interrupción sin controladores específicos, se instalará un controlador de interrupción predeterminado: **el controlador de interrupción predeterminado restablecerá el dispositivo** . Una aplicación puede anular el controlador predeterminado y proporcionar un controlador de interrupción predeterminado específico de la aplicación utilizando **BADISR\_vect** vector\_id dentro de la macro ISR( ) .

## Macro ISR( )

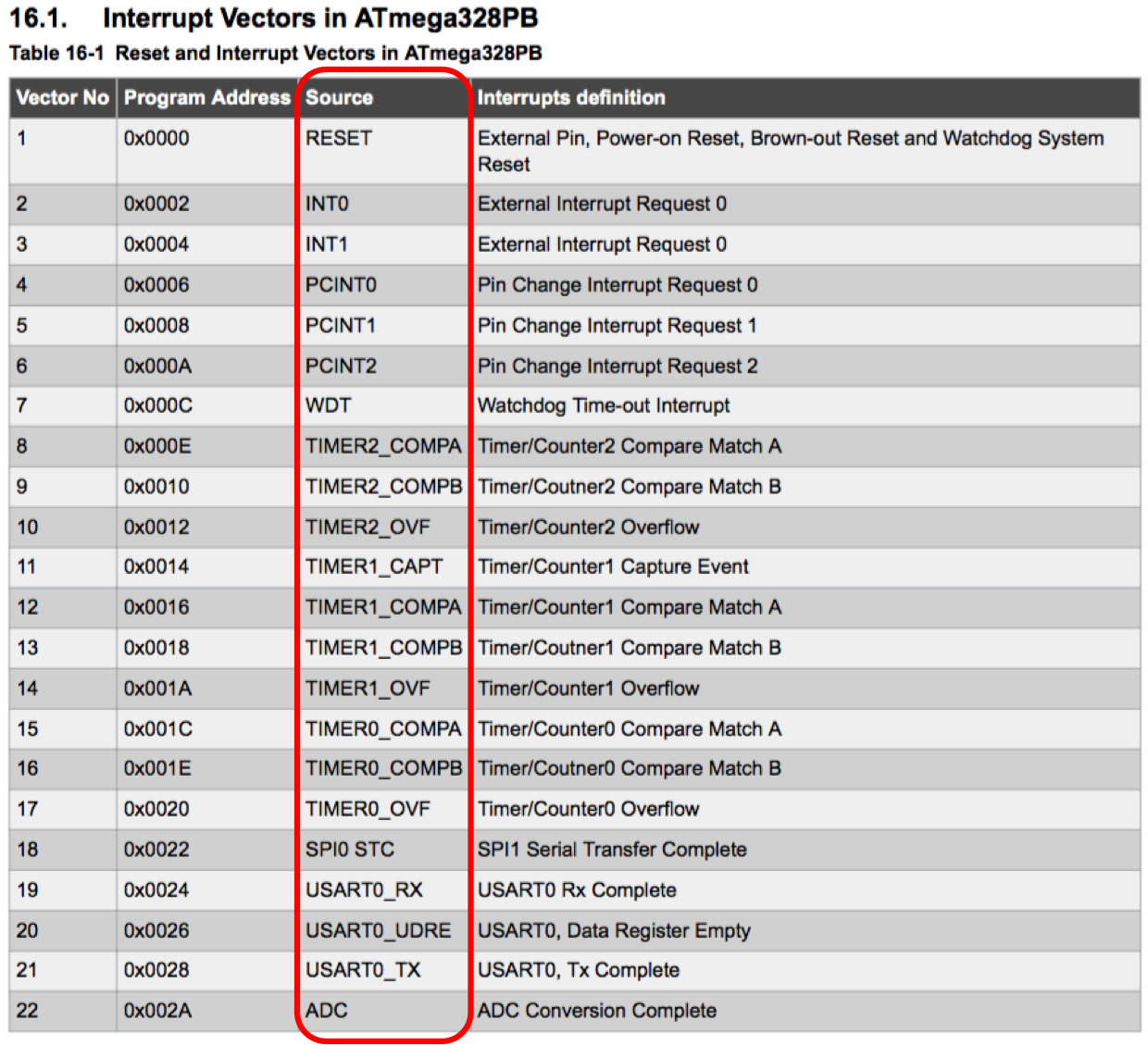
El siguiente ejemplo de código muestra cómo usar la macro ISR() para definir una función de interrupción:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <font></font>  ISR(vector\_id, ISR\_[BLOCK|NOBLOCK|NAKED|ALIASOF])<font></font>  {<font></font>      /\* Hardware auto-clears the interrupt flag (most interrupt sources) \*/<font></font>      /\* Clear the cause of the interrupt (required by some interrupt sources) \*/<font></font>      /\* ISR-specific processing \*/<font></font>  }        <font></font>  <font></font> |

Los diversos parámetros se describirán ahora con más detalle.

### **id\_vector**

Este identificador es una concatenación de un **Vector Source ID** y **\_vect** . Los ID de fuente de vector se encuentran en la hoja de datos del dispositivo, como se muestra (parcialmente) en el siguiente ejemplo para ATmega328PB:



Los vector\_ids mal escritos **aún generarán una función** , sin embargo, no se conectará a la tabla de vectores de interrupción. El compilador generará una advertencia si detecta un nombre sospechoso.

### Atributos

Los atributos ISR( ) proporcionan más instrucciones al compilador sobre cómo configurar la función de interrupción.

#### **ISR\_BLOCK**

Las interrupciones globales son inicialmente deshabilitadas por el hardware AVR al ingresar al ISR. Esta configuración **no modifica** este estado.

Este atributo es **idéntico a una macro ISR( ) sin atributo especificado**

#### **ISR\_NOBLOCK**

ISR se ejecuta con interrupciones globales habilitadas inicialmente. El compilador activa el indicador de habilitación de interrupción lo antes posible dentro de la ISR para garantizar un retraso de procesamiento mínimo para las interrupciones anidadas.

Esto se puede usar para crear ISR anidados, sin embargo, se debe tener cuidado para evitar desbordamientos de pila o para evitar ingresar infinitamente al ISR en aquellos casos en los que el hardware AVR no borre el indicador de interrupción respectivo antes de ingresar al ISR.

#### **ISR\_NAKED**

ISR se crea sin código de prólogo o epílogo. El código de usuario es responsable de la preservación del estado de la máquina, incluido el registro SREG, así como de colocar un reti() al final de la rutina de interrupción.

#### **ISR\_ALIASOF(id\_vector)**

Esto se puede usar para definir vectores adicionales que comparten el mismo controlador. El siguiente ejemplo crea un alias del vector PCINT1 para el controlador PCINT0:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <font></font>  ISR(PCINT0\_vect)<font></font>  {<font></font>    ...<font></font>    // Code to handle the event.<font></font>  }<font></font>  ISR(PCINT1\_vect, ISR\_ALIASOF(PCINT0\_vect));<font></font>  <font></font> |

### Ejemplo de ISR( )

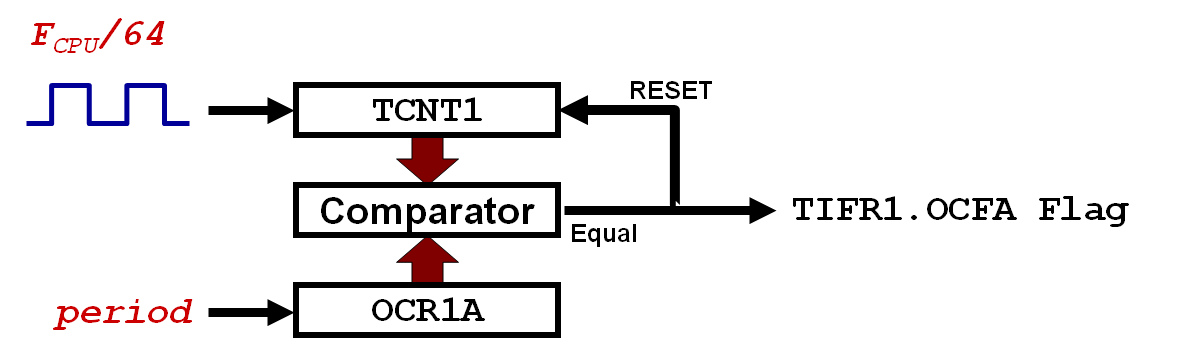
En este ejemplo de código, destacamos los archivos de encabezado requeridos y la definición ISR correcta de una función de controlador para la fuente de interrupción del modo Timer/Counter1 Clear-Timer-On-Compare (CTC). El controlador alterna **LED0** en el [**ATmega328PB Xplained Mini**](https://microchipdeveloper.com/boards:atavr328) cada 100 mS:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  dieciséis  17  18  19  20  21  22  23  24  25  26  27  28 | #include <avr/io.h>  #include <avr/interrupt.h>    ISR(TIMER1\_COMPA\_vect, ISR\_BLOCK)  {      PORTB ^= (1 << PORTB5);   // Toggle LED0  }    **int** main(**void**)  {      // Initialization        // Set LED as output      DDRB |= (1 << PORTB5);     // Configure PB5 as digital output      PORTB &= ~(1 << PORTB5);   // Set initial level for PB5        // Set up Timer/Counter1      TCCR1B |= (1 << WGM12 );   // Configure timer 1 for CTC mode      OCR1A = 25000;             // Set CTC compare value to 10Hz (100mS)                                 // at 16MHz AVR clock, with a prescaler of 64      TIMSK1 |= (1 << OCIE1A );  // Enable CTC interrupt      TCCR1B |= ((1 << CS10 ) | (1 << CS11 )); // Start Timer/Counter1 at F\_CPU/64        // Enable all interrupts      sei();    **while**(1);  } |

**Paso 3. Configurar el periférico**

A continuación, debe configurar el periférico para generar eventos de solicitud de interrupción.

Por ejemplo, el ATmega328PB contiene varios módulos periféricos de temporizador/contador. Cada módulo tiene un modo llamado **Clear Timer on Compare** (CTC) que, cuando se inicializa correctamente, activará periódicamente una señal de **indicador de coincidencia de comparación de salida del** temporizador 1 en el registro de indicador de interrupción TC1 (TIFR1), como se muestra:

[](https://microchipdeveloper.com/local--files/8avr:interrupts-mega-configuration/timer1-ctc-int-flag.png)

En este ejemplo, inicializaremos Timer/Counter1 en modo CTC para generar solicitudes de interrupción cada 100 mS, dada una entrada preescala de 250 kHz (16 MHz/64):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  dieciséis  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | <font></font>  #include <avr/io.h><font></font>  #include <avr/interrupt.h><font></font>  <font></font>  ISR(TIMER1\_COMPA\_vect, ISR\_BLOCK)<font></font>  {<font></font>      PORTB ^= (1 << PORTB5);   // Toggle LED0<font></font>  }<font></font>  <font></font>  **int** main(**void**)<font></font>  {<font></font>      // Initialization<font></font>  <font></font>      // Set LED as output<font></font>      DDRB |= (1 << PORTB5);     // Configure PB5 as digital output        <font></font>      PORTB &= ~(1 << PORTB5);   // Set initial level for PB5<font></font>  <font></font>      // Set up Timer/Counter1<font></font>      TCCR1B |= (1 << WGM12 );   // Configure timer 1 for CTC mode<font></font>      OCR1A = 25000;             // Set CTC compare value to 10Hz (100mS)<font></font>                                 // at 16MHz AVR clock, with a prescaler of 64<font></font>      TIMSK1 |= (1 << OCIE1A );  // Enable CTC interrupt<font></font>      TCCR1B |= ((1 << CS10 ) | (1 << CS11 )); // Start Timer/Counter1 at F\_CPU/64<font></font>  <font></font>      // Enable all interrupts<font></font>      sei(); <font></font>  <font></font>  **while**(1);<font></font>  }<font></font>  <font></font> |

Este es un ejemplo de una [interrupción no persistente](https://microchipdeveloper.com/8avr:interrupts-mega-overview#non-persistent-interrupts) . El indicador TIFR1.OCFA se borra automáticamente por el hardware al ingresar al controlador.

El indicador TIFR1.OCFA también se puede borrar manualmente escribiendo un "1" lógico en la ubicación del bit.

# Paso 4. Habilitar todas las interrupciones

Finalmente, debemos habilitar globalmente todas las interrupciones periféricas habilitadas configurando el **bit I de habilitación de interrupción global** en el **registro de estado (SREG)** . La biblioteca de interrupciones AVR-LIBC proporciona dos funciones de macro útiles para esto:

* sei() para habilitar interrupciones globalmente
* cli( ) para deshabilitar las interrupciones globalmente

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  dieciséis  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | <font></font>  #include <avr/io.h><font></font>  #include <avr/interrupt.h><font></font>  <font></font>  ISR(TIMER1\_COMPA\_vect, ISR\_BLOCK)<font></font>  {<font></font>      PORTB ^= (1 << PORTB5);   // Toggle LED0<font></font>  }<font></font>  <font></font>  **int** main(**void**)<font></font>  {<font></font>      // Initialization<font></font>  <font></font>      // Set LED as output<font></font>      DDRB |= (1 << PORTB5);     // Configure PB5 as digital output        <font></font>      PORTB &= ~(1 << PORTB5);   // Set initial level for PB5<font></font>  <font></font>      // Set up Timer/Counter1<font></font>      TCCR1B |= (1 << WGM12 );   // Configure timer 1 for CTC mode<font></font>      OCR1A = 25000;             // Set CTC compare value to 10Hz (100mS)<font></font>                                 // at 16MHz AVR clock, with a prescaler of 64<font></font>      TIMSK1 |= (1 << OCIE1A );  // Enable CTC interrupt<font></font>      TCCR1B |= ((1 << CS10 ) | (1 << CS11 )); // Start Timer/Counter1 at F\_CPU/64<font></font>  <font></font>      // Enable all interrupts<font></font>      sei(); <font></font>  <font></font>  **while**(1);<font></font>  }<font></font>  <font></font> |

# Compartir datos con el ISR

Las variables compartidas entre el ISR y el programa principal deben declararse como **volátiles** y tener un alcance **global .**Al compilar usando el optimizador, en un bucle como el siguiente:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <font></font>  uint8\_t flag;<font></font>  ...<font></font>  ISR(SOME\_vect) {<font></font>      flag = 1;<font></font>  }<font></font>  ...<font></font>  **while**(flag == 0) {<font></font>  ...<font></font>  }<font></font>  <font></font> |

el compilador generalmente accederá a "bandera" solo una vez y optimizará los accesos adicionales por completo, ya que su análisis de ruta de código muestra que nada dentro del ciclo podría cambiar el valor de "bandera" de todos modos. Para decirle al compilador que esta variable podría cambiarse fuera del alcance de su análisis de ruta de código (por ejemplo, dentro de una rutina de servicio de interrupción), la variable debe declararse así:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <font></font>  **volatile** uint8\_t flag;<font></font>  ...<font></font>  ISR(SOME\_vect) {<font></font>      flag = 1;<font></font>  }<font></font>  ...<font></font>  **while**(flag == 0) {<font></font>  ...<font></font>  }<font></font>  <font></font> |

Cuando la variable se declara **volátil** como se indicó anteriormente, el compilador se asegura de que dondequiera que se actualice o lea la variable, siempre escribirá los cambios en la memoria SRAM y leerá la variable desde SRAM.

# Operaciones de datos atómicos

Para que una operación sea considerada **atómica** , debe garantizar el acceso **ininterrumpido** de una determinada variable. Muchos lenguajes ensambladores brindan esto en ciertos niveles, es decir, prueba y configuración de bits, sin embargo, no existe **ninguna disposición** para proporcionar automáticamente la atomicidad de todos los tipos de variables en el lenguaje ANSI C.

Las expresiones y declaraciones ANSI-C **no son atómicas** .

Este problema puede ser problemático (en ciertas situaciones) cuando **las variables de varios bytes se comparten** con un ISR. Si bien declarar una variable de este tipo como **volátil** garantiza que el compilador no optimizará los accesos a ella, no garantiza el acceso **atómico** a ella. Considere el siguiente ejemplo de código:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  dieciséis  17  18  19  20  21  22  23  24 | <font></font>  #include <stdint.h><font></font>  #include <avr/io.h><font></font>  #include <avr/interrupt.h><font></font>  <font></font>  **volatile** uint16\_t ctr;<font></font>  <font></font>  ISR(TIMER1\_OVF\_vect)<font></font>  {<font></font>    ctr--;<font></font>  }<font></font>  ...<font></font>  **int**<font></font>  main(**void**)<font></font>  {<font></font>     ...<font></font>     ctr = 0x0200;<font></font>     start\_timer();<font></font>  **while**(ctr != 0)<font></font>       // wait<font></font>         ;<font></font>     ...<font></font>  }<font></font>  <font></font> |

Existe la posibilidad de que el contexto principal salga de su ciclo while() cuando la variable **ctr** alcance el valor 0x00FF. Esto sucede porque el compilador no puede acceder de forma nativa a una variable de 16 bits de forma atómica en una CPU de 8 bits. Entonces, cuando **ctr** está, por ejemplo, en 0x0100, el compilador luego prueba el byte bajo para 0, lo que tiene éxito. Luego procede a probar el byte alto, pero en ese momento se activa el ISR y el contexto principal se interrumpe. El ISR disminuirá la variable de 0x0100 a 0x00FF, luego continúa el contexto principal. Ahora prueba el byte alto de la variable que (ahora) también es 0, por lo que concluye que la variable ha llegado a 0 y finaliza el ciclo.

## Macros de acceso atómico

[**La biblioteca atómica**](http://www.nongnu.org/avr-libc/user-manual/group__util__atomic.html) AVR-LIBC proporciona las macros ATOMIC\_BLOCK que insertan la protección de interrupción adecuada cuando se desea acceso atómico. Estas macros operan a través de la manipulación automática del bit de **estado de interrupción global (I) del registro SREG** . Las rutas de salida de ambos tipos de bloques se gestionan automáticamente sin necesidad de consideraciones especiales, es decir, el estado de interrupción se restaurará al mismo valor que tenía al entrar en el bloque respectivo. Usando las macros de este archivo de encabezado, el código anterior se puede reescribir como:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  dieciséis  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | <font></font>  #include <stdint.h><font></font>  #include <avr/io.h><font></font>  #include <avr/interrupt.h><font></font>  #include <util/atomic.h><font></font>  <font></font>  **volatile** uint16\_t ctr;<font></font>  <font></font>  ISR(TIMER1\_OVF\_vect)<font></font>  {<font></font>    ctr--;<font></font>  }<font></font>  ...<font></font>  **int** main(**void**)<font></font>  {<font></font>     uint\_16 ctr\_copy;<font></font>     ...<font></font>     ctr = 0x0200;<font></font>     start\_timer();<font></font>  **do**<font></font>     {<font></font>       ATOMIC\_BLOCK(ATOMIC\_RESTORESTATE)<font></font>       {<font></font>         ctr\_copy = ctr;<font></font>       }<font></font>     } **while**(ctr != 0);<font></font>       // wait<font></font>         ;<font></font>     ...<font></font>  }<font></font>  <font></font> |

La macro **ATOMIC\_BLOCK** instalará la protección de interrupción adecuada antes de acceder a la variable **ctr** , por lo que se garantiza que se probará de manera consistente. En este caso, el parámetro **ATOMIC\_RESTORESTATE** hace que ATOMIC\_BLOCK restaure el estado anterior del registro SREG, guardado antes de que se deshabilitara el bit indicador de estado de interrupción global. El efecto neto de esto es hacer que el contenido de ATOMIC\_BLOCK sea atómico garantizado, sin cambiar el estado del indicador de estado de interrupción global cuando se completa la ejecución del bloque.