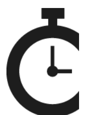
# Microcontroladores: CPU

PID_00247322

Francisco Hernández Ramírez
Màrius Montón Macián
Juan Daniel Prades García

Tiempo mínimo de dedicación recomendado: 2 horas



*Ninguna parte de esta publicación, incluido el diseño general y la cubierta, puede ser copiada, reproducida, almacenada o transmitida de ninguna forma, ni por ningún medio, sea éste eléctrico, químico, mecánico, óptico, grabación, fotocopia, o cualquier otro, sin la previa autorización escrita de los titulares del copyright.*

# Índice

**Introducción** ........ 5

**1. Harvard y von Neumann** ........ 6
1.1. Harvard ........ 6
1.2. Harvard modificada ........ 7
1.3. Von Neumann ........ 7

**2. Punto fijo** ........ 9
2.1. Signo y magnitud ........ 10
2.2. Complemento a 1 ........ 10
2.3. Complemento a 2 ........ 11

**3. Punto flotante** ........ 12
3.1. Precisión simple ........ 13
3.2. Precisión extendida ........ 14
3.3. Uso en microcontroladores ........ 14

**4. Interrupciones** ........ 15
4.1. Fundamentos conceptuales de las interrupciones ........ 17
4.2. Vectores de interrupción ........ 18
4.3. Rutinas de servicio a la interrupción (ISR) ........ 20
4.4. Interrupciones: consideraciones finales ........ 21

**Bibliografía** ........ 23

# Introducción

En este contenido se van a presentar las principales características de los microcontroladores, tanto a nivel de arquitectura como de módulos básicos, así como los periféricos más habituales.

Para una lectura más detallada y exhaustiva sobre el tema, [Stallings (2010)] es uno de los grandes libros de referencia de este campo.

# 1. Harvard y von Neumann

En cuanto a la organización de la CPU y su acceso a la memoria principal, hay dos grandes tipos de arquitecturas:

- Harvard: En esta arquitectura hay separación total entre el camino de datos y el camino de instrucciones. De este modo, las dos memorias están separadas y pueden incluso ser de diferente naturaleza o características.

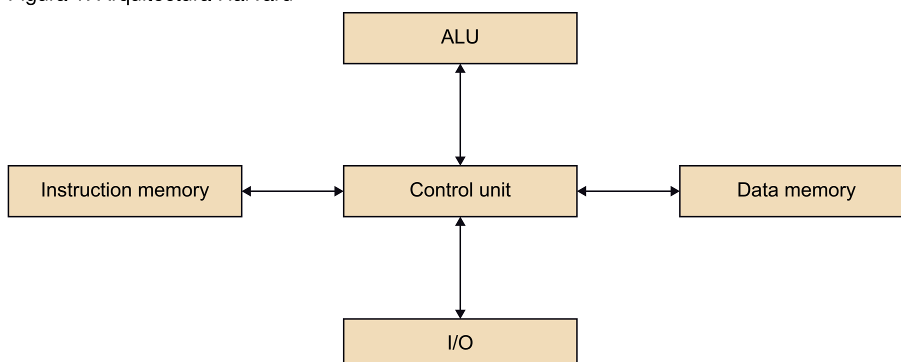
- Von Neumann: En esta arquitectura hay una misma memoria para almacenar datos y programa (instrucciones). Esta arquitectura simplifica el diseño de las distintas partes de la CPU.

## 1.1. Harvard

Como ya se ha comentado, esta arquitectura separa en dos la memoria que usa la CPU: una memoria es la destinada a almacenar los datos y la otra memoria, a almacenar las instrucciones (el programa). De este modo, la CPU necesita acceder primero a la memoria de instrucciones para recuperar la siguiente instrucción que ejecutar y luego, si es necesario, acceder a la memoria de datos para hacer las operaciones necesarias (figura 1).

El tener accesos de memoria separados ayuda a la optimización del acceso a cada una de ellas y abre la posibilidad de paralelizar los accesos, con lo que se puede hacer un acceso a la memoria de instrucciones mientras se está haciendo un acceso a la memoria de datos.

Figura 1. Arquitectura Harvard



Sin embargo, tener los dos tipos de memorias separados y con un tipo de acceso distinto complica la tarea del desarrollador, ya que, por ejemplo, el acceso a constantes (datos) en una parte de código (instrucciones) no se puede realizar.

No obstante, esta arquitectura no se suele utilizar en su forma más pura y se trabaja con la que se conoce como **arquitectura Harvard modificada**.

### 1.2. Harvard modificada

Esta variación de la arquitectura Harvard original es la más utilizada actualmente, y consiste en una serie de cambios respecto a la original:

- Una caché que separa los dos canales (datos y instrucciones) en un primer nivel, pero que los unifica en un segundo nivel. De esta manera, el acceso a un dato o instrucción pasa a ser transparente para el programador.

- Acceso a instrucciones como dato ofreciendo instrucciones especiales para acceder a datos que están almacenados en la parte de código (por ejemplo, variables definidas como constantes).

Con estos cambios en la arquitectura, que de hecho la acercan a una arquitectura tipo Von Neumann, se simplifica la codificación y el diseño de código.

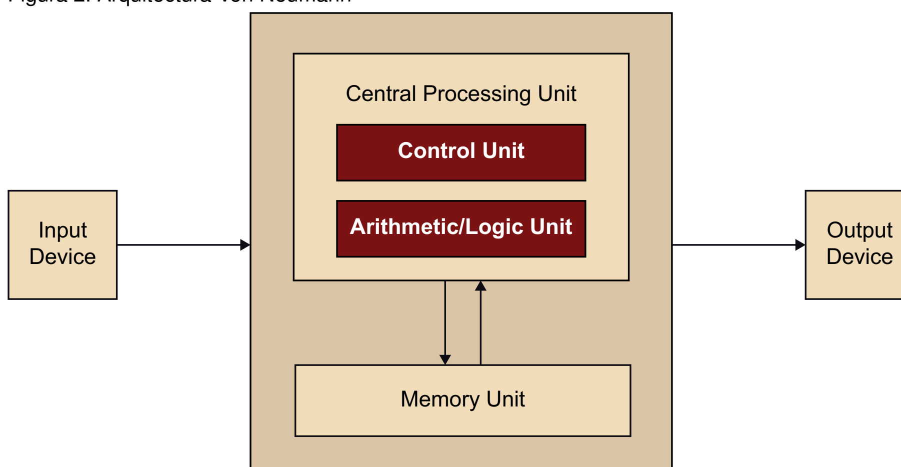
**Memoria caché**

Una memoria caché es una memoria muy rápida que almacena los datos o las instrucciones más utilizadas. Esta memoria se sitúa entre el procesador y la memoria principal, y permite mejorar el rendimiento de los accesos a memoria.

### 1.3. Von Neumann

Esta arquitectura de CPU tiene solamente una memoria para almacenar tanto datos como instrucciones (figura 2). Este diseño simplifica la parte de acceso a la memoria. Como contrapartida, el rendimiento de esta arquitectura es menor que el de la arquitectura Harvard, ya que en esta la CPU no puede paralelizar los accesos a memoria.

Figura 2. Arquitectura Von Neumann



Para mitigar este problema, es muy habitual el uso de memoria caché en este tipo de procesadores. Esta caché puede usar la arquitectura Harvard para paralelizar los accesos a memoria y mejorar aún más el rendimiento global del sistema.

Esta arquitectura es la seleccionada en el diseño de la mayoría de los procesadores de propósito general usados habitualmente.

## 2. Punto fijo

La representación de números enteros es la más sencilla y habitual de las representaciones de valores en computadores.

Esta representación funciona de manera muy similar a nuestra numeración en base 10, simplemente cambiando la base por la 2. De esta manera, el bit menos significativo (más a la derecha) representa el valor $2^0$, el siguiente bit el valor $2^1$, y así sucesivamente.

Entonces, el valor binario $10110_2$ representa el valor $22_{10}$. Veamos cómo se hace la conversión:

$$10110_2 \rightarrow 1{\times}2^4{+}0{\times}2^3{+}1{\times}2^2{+}1{\times}2^1{+}0{\times}2^0 = 1{\times}16{+}0{\times}8{+}1{\times}4{+}1{\times}2{+}0{\times}1 = 22$$

**Base 10, base 2**

La base en un sistema de numeración posicional es el valor que se usa como base de la potencia que se va incrementando en cada posición. La numeración habitual es en base 10, así cada posición contiene una potencia de 10: $345 = 3 \times 10^2 + 4 \times 10^1 + 5 \times 10^0$. En computadores la base habitual es la base 2.

De modo general, podemos decir que para una secuencia de bits $a_{n-1}a_{n-2}...a_1a_0$ el valor A que almacena es:

$$A = \sum_{i=0}^{N-1} 2^i a_i$$

Así, para representar números enteros positivos, la solución es sencilla, y cuantos más bits se utilizan, más rango de números se pueden representar. El valor máximo posible en este formato se puede calcular de la siguiente manera:

$$Num_{max} = 2^N - 1$$

donde *N* es el número de bits usado por la representación. Así, para el número de bits más habituales tenemos:

Tabla 1. Rangos para palabras de bits más usuales

| Bits | Valor mínimo | Valor máximo |
|---|---|---|
| 8 | 0 | 255 |
| 16 | 0 | 65535 |
| 32 | 0 | 4294967295 |
| 64 | 0 | 1,84467440737e+19 |

Elaboración propia.

Esta representación es clara y sencilla para números naturales (enteros positivos), pero no se pueden representar números con signo. Para ello, hay otras representaciones.

### 2.1. Signo y magnitud

La representación más sencilla para almacenar números enteros con signo es la de signo y magnitud. En esta representación se usa un bit para indicar el signo del valor y el resto de los bits se emplean para representar el valor absoluto del número.

Así, para 5 bits esta representación puede almacenar desde el valor $-15_{10} \rightarrow 11111_2$ hasta $+15_{10} \rightarrow 01111_2$.

Como se puede observar en la tabla 2, esta representación tiene el problema de que representa el 0 de dos formas distintas. Además, operar con esta representación es complejo debido a que las operaciones deben tener en cuenta un bit aparte (el de signo) para saber cómo operar. Por ello, esta representación apenas se usa en los microcontroladores actuales.

### 2.2. Complemento a 1

Otra propuesta de representación de números enteros es la llamada complemento a 1. En esta representación también se indica si el valor es positivo o negativo con el bit más significativo (más a la izquierda), pero se construye su representación de manera diferente.

Para construir una representación de un número negativo, se toma la representación entera del valor positivo y se invierten todos sus bits. Así por ejemplo, para representar el número $-9_{10}$ en C1 con 5 bits se usa su valor sin signo $9_{10} \rightarrow 01001_2$. El siguiente paso es invertir bit a bit esta representación binaria para obtener $10110_2$. Este es el valor en C1 del número $-9_{10}$.

De nuevo aparece la doble representación del 0 (siendo $0000_2$ y $11111_2$ los dos valores). Sin embargo, con esta representación las operaciones aritméticas se simplifican, ya que se pueden operar los números directamente.

### 2.3. Complemento a 2

Esta representación, igual que las anteriores, reserva el bit más significativo para indicar si el valor es positivo o negativo. La diferencia con las anteriores es, de nuevo, cómo se representa el valor en el resto de los bits.

En complemento a 2, si el número es negativo se almacena el valor que hay que restar del máximo representable para obtener el valor negativo. Así, tenemos la siguiente expresión si el bit más significativo es negativo:

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{N-2} 2^i a_i$$

En esta representación, como se ve en la tabla 2, no hay duplicidad del 0 y las operaciones aritméticas se simplifican enormemente.

Esta representación es la más utilizada en los microprocesadores y microcontroladores actuales para almacenar números enteros.

Tabla 2. Signo y magnitud, C1 y C2 para 5 bits

| Valor | SM | C1 | C2 |
|---|---|---|---|
| -16 | - | - | 10000 |
| -15 | 11111 | 10000 | 10001 |
| -14 | 11110 | 10001 | 10010 |
| ... | ... | ... | ... |
| -2 | 10010 | 11101 | 11110 |
| -1 | 10001 | 11110 | 11111 |
| -0 | 10000 | 11111 | - |
| +0 | 00000 | 00000 | 00000 |
| +1 | 00001 | 00001 | 00001 |
| +2 | 00010 | 00010 | 00010 |
| ... | ... | ... | ... |
| +14 | 01110 | 01110 | 01110 |
| +15 | 01111 | 01111 | 01111 |

Elaboración propia.

# 3. Punto flotante

Con la representación de punto fijo es posible representar valores numéricos enteros positivos o negativos centrados en el 0. Eligiendo correctamente una posición para el punto decimal, se pueden representar ciertos valores fraccionarios de forma precisa.

Sin embargo, esta representación tiene limitaciones, ya que no puede representar números muy grandes ni muy pequeños (fracciones pequeñas) y hay problemas con la pérdida de precisión según se acumulan cálculos. De hecho, trabajando con representación entera de 32 bits el mayor número representable es $2^{32} = 4,294,967,296$.

Una forma de solucionarlo es trabajando en notación científica. Usando esta notación en base 10, el valor 123,000,000,000 se puede escribir como $123 \times 10^9$ y el valor 0.000000000123 se puede escribir como $1{,}23 \times 10^{-10}$. Lo que se hace en esta notación es desplazar el punto decimal donde interesa y usar el exponente en base 10 para anotar dónde va el punto decimal.

Con esta notación se pueden representar números muy grandes y muy pequeños con muy pocos dígitos.

Lo mismo puede hacerse trabajando en base 2, donde representaríamos un número de la siguiente forma:

$$\pm S \times B^{\pm E}$$

donde:

- Signo positivo o negativos
- *S* mantisa del número
- *B* base siempre será 2
- *E* exponente positivo o negativo con un *bias*

Así, almacenando el **signo**, la **mantisa *S*** y el **exponente *E*** (la base es siempre 2 y no hace falta almacenarla) se puede representar un rango muy elevado de valores tanto muy grandes como muy pequeños.

Al exponente se le añade un *bias* para poder representar números positivos y negativos. Así, al valor almacenado en el exponente se le resta el *bias* para obtener el valor final que vamos a usar.

Si, por ejemplo, almacenamos un valor representado de esta forma en 16 bits, y elegimos:

- Signo: 1 bit
- *S*: 10 bits
- *E*: 5 bits (con un *bias* de 15)

Tendremos que el valor más pequeño que podemos representar será el

$1{,}0 \times 2^{-15} = 3{,}0517 \times 10^{-5}$ y el de mayor valor el $1023 \times 2^{16} = 67,043,328$

Un aspecto importante que hay que destacar es que en este formato no se representan más números que usando punto fijo. El número máximo de valores que se pueden representar es el mismo (en este caso $2^{16} = 65536$ valores). Lo que se hace con esta notación es «repartir» estos valores para que el rango de trabajo sea mayor.

También hay que destacar que, por tanto, no es posible representar todos los números dentro del rango, ya que se ha «repartido» su representación. Por ello una representación en punto flotante no será exacta en la mayoría de los casos debido a que ese valor en concreto no puede almacenarse y se almacenará el valor más próximo. Eligiendo bien los parámetros de la representación se puede minimizar el error relativo de representación para que pueda usarse en aplicaciones normales.

A día de hoy el uso y la representación de valores en punto flotante están estandarizados en el IEEE754 [IEEE (2008)], que define tanto la representación como la aritmética de este formato.

En este estándar se definen distintas variantes de almacenar valores en punto flotante. Por ejemplo, se definen valores especiales tales como infinito (positivo o negativo), NaN (*not a number*, para representar errores en los cálculos) o el 0.

Los formatos más usuales se detallan a continuación.

### 3.1. Precisión simple

En esta representación se usan 32 bits para almacenar un número en punto flotante. Los 32 bits se reparten como sigue:

- Signo: 1 bits
- Exponente: 8 bits con *bias* = 127

- Mantisa: 23 bits

En esta representación, el valor mínimo representable posible es $1{,}2 \times 10^{-38}$ y el valor máximo, $3{,}4 \times 10^{38}$.

En lenguaje C, para declarar una variable que contenga un valor en este formato debe declararse de tipo *float*.

### 3.2. Precisión extendida

En esta representación se usan 64 bits para almacenar un número en punto flotante. Los 64 bits se reparten como sigue:

- Signo: 1 bits
- Exponente: 11 bits con *bias* = 1023
- Mantisa: 52 bits

En esta representación, el valor mínimo representable posible es $2{,}2 \times 10^{-308}$ y el valor máximo, $1{,}8 \times 10^{308}$.

En lenguaje C, para declarar una variable que contenga un valor en este formato debe declararse de tipo *double*.

### 3.3. Uso en microcontroladores

Vista la representación de números en punto flotante, es fácil suponer que las operaciones aritméticas con este formato son bastante más complejas que usando formatos de punto fijo. Por este motivo tradicionalmente los dispositivos de bajo precio y consumo no disponían de unidades aritméticas que pudiesen trabajar en punto flotante.

Si bien todos los microcontroladores que se puedan llamar como tal pueden manejar tipos enteros de datos, la incorporación de unidades para el cómputo en coma flotante no ha sido generalizada hasta hace muy poco. Por ello, han existido librerías de software que implementan las operaciones en formatos de punto flotante sin ayuda específica de ningún módulo de hardware.

Hay que tener en cuenta que estas operaciones hechas por las librerías software son muy costosas en número de ciclos de operación, lo que provoca que, por ejemplo, para una suma de dos números en punto flotante sea necesarios varios centenares de ciclos de operación.

Por todo ello, normalmente se desaconseja el uso de variables de tipo *float* o *double* en el código si no es estrictamente necesario.

# 4. Interrupciones

Las interrupciones recogidas dentro del software de control de sistemas que funcionan en tiempo real son a menudo uno de los factores clave que permite su funcionamiento correcto en una aplicación determinada.

Podemos definir una interrupción como la señal que recibe el microprocesador para redirigir el flujo del programa que se está ejecutando en un momento determinado.

**Ejemplo**

Las señales que los periféricos de un ordenador envían a la CPU y que hacen que empiece una nueva tarea para, posteriormente, retomar la primera una vez finalizada la condicionada por el periférico sería un ejemplo típico de interrupción.

En dispositivos reales, la mayoría de los microprocesadores integrados en sistemas empotrados están conectados a componentes capaces de generar interrupciones muy variadas. En cambio, en cuanto al desarrollo de software, las propias interrupciones, y también las rutinas de interrupción de servicio o ISR* necesarias para gestionarlas de manera adecuada, son a menudo pasadas por alto por los programadores noveles, puesto que el error más pequeño en su configuración y en su diseño suele originar un fallo general de los sistemas muy difícil de corregir *a posteriori*. De hecho, estos programadores suelen usar el esquema más simple posible de programación para evitar utilizarlos y que se basa en preguntar de una manera recursiva a los dispositivos generadores de interrupciones, por ejemplo, los periféricos, si necesitan algún servicio del microprocesador. Obviamente, esta solución, conocida en inglés como *polled communication*, funciona bien cuando el procesador es rápido y potente, pero en el caso de sistemas empotrados podemos identificar las desventajas siguientes:

*** Sigla de *interrupt service routine*. A veces, también se emplea el término *interrupt service process* (ISP).**

- **Consumo de muchos recursos de microprocesador**, puesto que, independientemente de si el periférico solicita servicio o no, el procesador ha de preguntar de manera recursiva sobre su estatus.

- **Código de software poco ordenado**. A cada iteración del programa principal, este tiene que preguntar al periférico sobre su estatus y, en caso de originarse una necesidad de servicio, se debe incluir dentro del programa principal el código para tratarla. Todo ello acaba complicando bastante el código final del programa.

- **Estados de latencia elevados de los periféricos**. Si el microprocesador está ocupado haciendo otra función, el dispositivo periférico es ignorado hasta que el primero le pregunte sobre su estatus. En caso de que el microprocesador sea lento, este punto puede llegar a ser crítico para determinadas aplicaciones.

Por lo tanto, es evidente que la programación basada en *polled communication* no es la aproximación más adecuada para controlar sistemas empotrados con capacidades de cálculo limitadas. Todo ello hace imprescindible diseñar su software de control considerando la necesidad de incluir en él interrupciones.

La gestión correcta de interrupciones en sistemas en tiempo real requiere una relación estrecha entre software y hardware, lo que determina el lenguaje de programación más adecuado para programar las ISR. Por lo tanto, la pregunta directa que se deriva es: ¿qué es mejor, lenguaje máquina o de alto nivel? En el primer caso, se hace relativamente sencillo estimar el tiempo de ejecución de la ISR, puesto que para un microprocesador determinado con una capacidad de cálculo conocida *a priori*, cada instrucción necesita aproximadamente $x$ microsegundos para ser ejecutada (dependiendo de factores intrínsecos al hardware, como el reloj interno o los tiempos de espera entre estados). Así, podemos llegar a acotar el tiempo máximo que necesitará el sistema para gestionar la interrupción formada por $y$ instrucciones.

Por el contrario, la programación de ISR en un lenguaje de nivel medio-alto resulta mucho más problemática. De hecho, y a pesar de lo que a menudo afirman los proveedores de compiladores de estos tipos de lenguajes, no hay ninguna manera óptima de escribir una ISR en un lenguaje como C, puesto que no podemos conocer ni siquiera cuánto tiempo necesitará nuestra máquina para ejecutar una única línea de código, tal como demuestra J. Ganssle en su libro *The Art of Designing Embedded Systems*. Incluso para una instrucción simple como es una iteración FOR, el tiempo de ejecución puede ir desde unos pocos milisegundos hasta consumir muchos recursos de máquina desde un punto de vista de tiempo y memoria. Esta dispersión dependerá fundamentalmente de cómo el compilador realice la conversión al lenguaje máquina del código en lenguaje de alto nivel. Así pues, en una primera aproximación podemos decir que no hay ningún impedimento para desarrollar ISR en lenguajes de alto nivel, como C, pero su tiempo de ejecución deberá ser convenientemente verificado para comprobar si satisface las especificaciones requeridas al sistema empotrado que se está programando. En el peor de los casos, cuando haya un riesgo de generación rápida de interrupciones concatenadas, se recomienda optar directamente por el lenguaje máquina como solución para evitar el colapso del sistema. Huelga indicar que aquí el término *rápido* es totalmente cualitativo e indefinido, que depende del significado fundamentalmente de la aplicación, del dispositivo final o, incluso, de las preferencias personales del programador.

**Bibliografía recomendada**

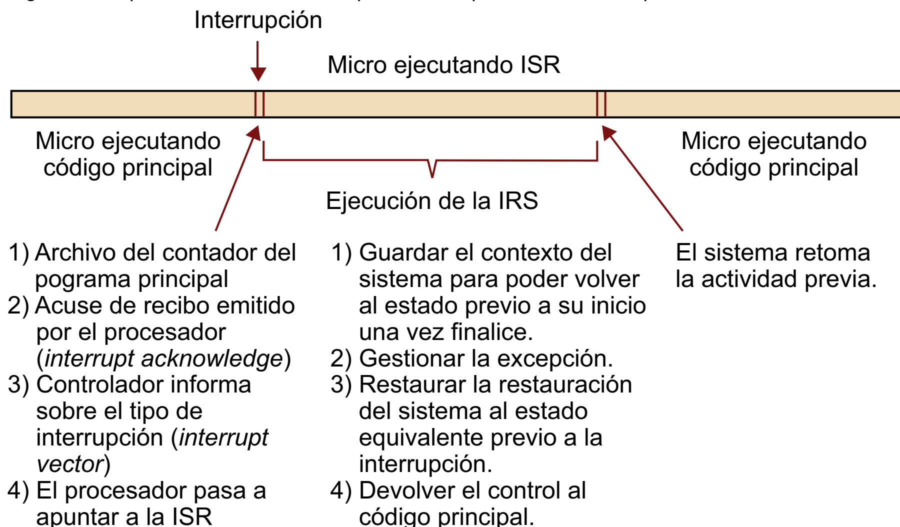
**J. G. Ganssle** (2000). *The Art of Designing Embedded Systems* (1.ª ed.). Woburn, Massachusetts: Newnes (Elsevier).

Vemos, pues, que la gestión optimizada de interrupciones se convierte en un trabajo solo al alcance de desarrolladores experimentados. En este texto introductorio, evidentemente el objetivo no es conseguir este nivel de conocimiento y complejidad, sino repasar la secuencia de eventos que acaban configurando una interrupción estándar, y también familiarizarnos con los conceptos que, independientemente del lenguaje de programación empleado, se usan para describirlas.

### 4.1. Fundamentos conceptuales de las interrupciones

La generación de una interrupción provoca que el sistema empotrado siga una serie de eventos a lo largo del ciclo asociado a su gestión, tal como se puede ver en la figura siguiente:

Figura 3. Esquema de las acciones típicas de un proceso de interrupción




Primero, la activación de una interrupción y el aviso posterior del controlador que la recoge en el microprocesador del sistema genera los eventos y las acciones siguientes en el hardware:

- **Archivo del contador del programa principal en ejecución en la pila del procesador**. La dirección de memoria que el procesador estaba ejecutando antes de la interrupción queda guardada junto con otra información, como, por ejemplo, los contenidos de sus registros internos. Este último punto dependerá del modelo de procesador utilizado.

- **Generación de un acuse de recibo (*interrupt acknowledge*) por el procesador**. El acuse de recibo es enviado al periférico o controlador que genera la interrupción solicitando un vector de interrupción (*interrupt vector*), cuyo contenido dependerá del tipo de interrupción. Un vector de interrupción no es más que una indicación de la dirección de memoria donde el microprocesador tiene que ir a buscar la ISR o el código de programa asociado a la interrupción y que se tiene que ejecutar para poderla gestionar.

- **Ejecución de la ISR**. El microprocesador, una vez recibido el vector de interrupción, apunta a la dirección de memoria donde está la ISR y ejecuta las funciones previstas para gestionar la interrupción.

Una vez que la interrupción ya sido tratada convenientemente, el procesador ejecuta las tareas siguientes:

- **Recuperación de la dirección de retorno y otra información archivada en la pila**. El procesador, para poder volver al estado previo a la generación de la interrupción, recupera la información archivada en la pila. Generalmente, la dirección de retorno (*return address*, en inglés) es la dirección de memoria donde está la instrucción posterior que se habría ejecutado según el programa principal si la interrupción no hubiera existido.

- **Continuación de la ejecución del programa principal**. El procesador continúa ejecutando el programa principal. Si el programador ha gestionado correctamente la interrupción, es evidente que este código principal ni siquiera notará que ha habido una interrupción por el medio, lo que por regla general habrá ocupado unos pocos milisegundos de CPU.

Analizando la secuencia de eventos anterior que constituyen una interrupción estándar, podemos afirmar que los puntos críticos del proceso son la generación del vector de interrupción y su interpretación correcta por parte del procesador y la ejecución de la ISR. En los apartados siguientes, veremos con más detalle estos dos aspectos y los requisitos generales que han de satisfacer los vectores y las ISR para evitar problemas con el funcionamiento final de dispositivos empotrados.

### 4.2. Vectores de interrupción

Todos los procesadores necesitan recibir un vector de interrupción cuando se les notifica la existencia de una interrupción para poder ejecutar la ISR correspondiente. Tal como hemos visto antes, los vectores indican al procesador dónde ir a buscar el servicio necesario para poder gestionar una interrupción de manera adecuada. Por regla general, los vectores son simplemente un número que el procesador traduce a una dirección de memoria donde está la primera instrucción de la ISR (podéis ver la tabla siguiente).

Tabla 3. Direcciones de memoria contenidas en los vectores asociados a cada una de las cuatro interrupciones que puede procesar un microprocesador 80188

| Tipo de interrupción | Dirección de memoria en el vector correspondiente |
|---|---|
| INT0 | 00030h |
| INT1 | 00034h |
| INT2 | 00038h |
| INT3 | 0003Ch |

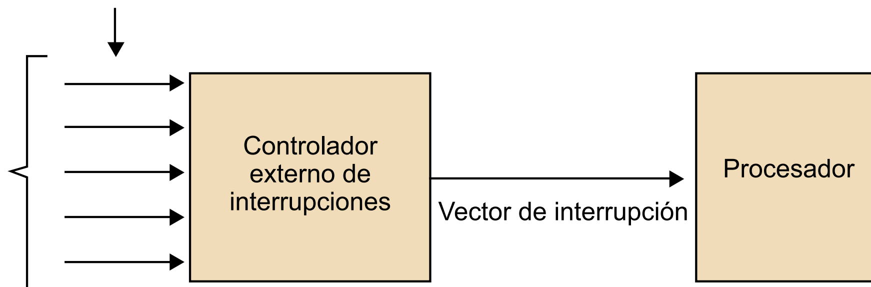
Sin embargo, algunos procesadores más antiguos también podían contener código ejecutable, pero esta aproximación cada vez es menos utilizada.

Actualmente, muchos procesadores están preparados para gestionar una interrupción especial y no ordinaria. Este tipo de interrupción, conocido como ***nonmaskable interrupt*** (NMI), se utiliza cuando se produce un error fatal en el sistema y no puede ser ignorado por el procesador, que, una vez que recibe un aviso de interrupción, solicita el vector correspondiente y finaliza el funcionamiento del dispositivo. A efectos prácti-

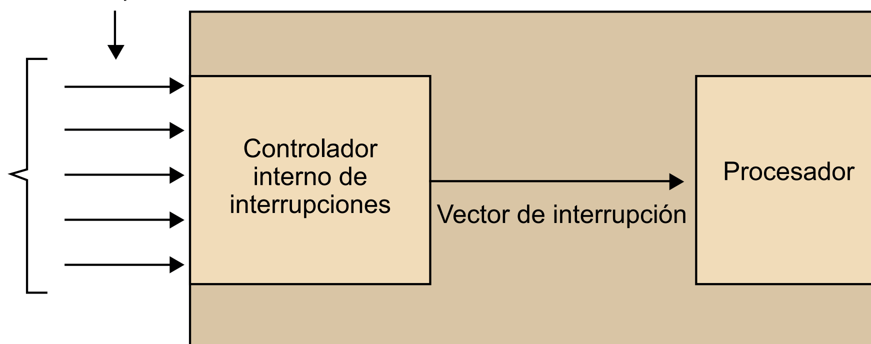
cos, esto puede provocar pérdidas de datos o situaciones no deseadas en determinadas aplicaciones; por lo tanto, muchos programadores prefieren diseñar sistemas empotrados que traten interrupciones críticas dejando el dispositivo inactivo a la espera de una revisión técnica.

Hay tres métodos para generar los vectores de interrupción solicitados por el procesador: a partir de un controlador externo, desde un controlador interno o desde los propios periféricos. La figura siguiente esquematiza el funcionamiento de cada uno de los tres modelos. A efectos prácticos, los tres producen el mismo resultado, a pesar de que en el hardware existen diferencias significativas, en las que no nos detendremos.

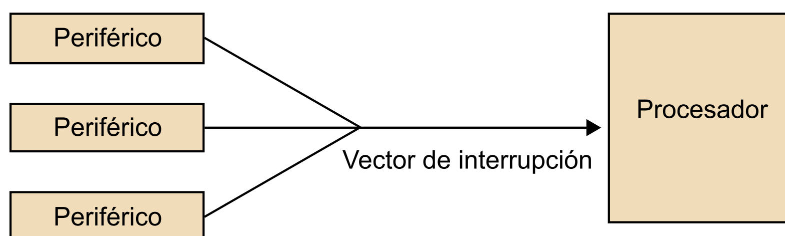
Figura 4. Mecanismos de generación de vector de interrupción




**Procesador con controlador externo de interrupciones**




**Procesador con controlador externo de interrupciones**




**Periféricos que solicitan interrupciones directamente al procesador**

**Ejemplo**

En el caso de controladores internos al procesador, no se llega a generar ningún vector de interrupción real, puesto que toda la información necesaria para gestionar la interrupción se trata dentro del mismo procesador, pero el programador acaba teniendo la impresión de que se ha usado un vector de interrupción virtual que modifica el funcionamiento del sistema.

### 4.3. Rutinas de servicio a la interrupción (ISR)

En el momento en el que se produce una interrupción, el procesador ejecuta una ISR. Para poder garantizar el funcionamiento correcto del sistema empotrado, todas las ISR deben satisfacer los tres puntos siguientes:

1) Ofrecer el servicio necesario para gestionar la interrupción generada.

2) Permitir al sistema aceptar nuevas interrupciones durante su ejecución.

3) Una vez finalizada la ISR, permitir y garantizar que el sistema vuelva al estado previo a la interrupción.

Los puntos primero y tercero son obvios y constituyen dos de las acciones básicas para poder gestionar una interrupción, tal como hemos visto anteriormente. En cambio, para justificar el segundo punto hemos de introducir el concepto de prioridad. Por regla general, todas las interrupciones tienen asignada una prioridad determinada. Este parámetro establece cuándo un procesador responde a la solicitud de interrupción de un periférico. Así, una interrupción de alta prioridad debería poder interrumpir la ejecución de una de prioridad más baja. Por el contrario, una de baja prioridad será ignorada siempre que una de alta esté siendo gestionada. Qué determina la prioridad dependerá del microprocesador que se emplee.

Así pues, para justificar la condición de que toda ISR ha de permitir aceptar nuevas interrupciones durante su ejecución, debemos pensar en el caso extremo de que una segunda interrupción de alta prioridad se genera mientras una primera de baja se está ejecutando. En caso de que el programador no lo haya previsto, lo más probable, exceptuando el caso de una NMI, es que la segunda sea ignorada hasta la finalización de la primera, puesto que la mayoría de los procesadores desconectan todos los interruptores y entradas desde los periféricos que lo pueden avisar de la existencia de nuevas solicitudes de interrupciones. El lector imaginará que esta situación puede originar situaciones no deseadas. Por este motivo, cada vez más procesadores permiten el tratamiento de interrupciones simultáneas o asíncronas.

En la aproximación más simple y tampoco recomendable, cualquier ISR finaliza si una nueva interrupción se genera durante su ejecución, independientemente de sus prioridades respectivas. Por el contrario, en el segundo esquema de funcionamiento, conocido como interrupciones imbricadas (*nested interrupts*), evidentemente más complicado desde el punto de vista de la programación, una ISR solo puede ser interrumpida por otra de más alta prioridad para después ser retomada una vez que la

**Excepciones NMI**

Las excepciones NMI son la excepción, puesto que por definición abortan el funcionamiento del sistema debido a un error fatal. Por lo tanto, solo satisfacen el primero de los tres puntos, puesto que tienen prioridad máxima.

**Ejemplo**

Por ejemplo, la familia de los 68.000 permite a un periférico que emite una interrupción evaluar su prioridad, y corresponde al programador la responsabilidad de gestionar los conflictos potenciales entre dos requisitos de igual grado de prioridad que puedan llegar al procesador. Por el contrario, el procesador Intel 8259 permite asignar al programador las prioridades de todas las posibles interrupciones que se pueden generar en el sistema.

segunda ha finalizado. Nos podemos imaginar que para poder retomar la primera ISR en su punto de interrupción hay que guardar registros y direcciones de memoria en la pila. Por lo tanto, que este modo de trabajar sea factible dependerá fundamentalmente de la capacidad y de los recursos del procesador, puesto que debemos recordar que en la pila tenemos guardada previamente la información relativa al programa principal del sistema que ha sido interrumpido. Además, no se puede obviar el grado de complejidad que se deriva del diseño de interrupciones imbricadas, lo que restringe su uso correcto a programadores experimentados y muy familiarizados con el tema.

### 4.4. Interrupciones: consideraciones finales

> Las interrupciones son señales que reciben los microprocesadores para redirigir el flujo del programa que se está ejecutando y de este modo satisfacer los requisitos de partes del sistema, como los periféricos. El código que se ejecuta para gestionar esta demanda se conoce como *rutina de interrupción de servicio* o ISR.

La programación de interrupciones es compleja y puede inducir fácilmente a errores críticos en los sistemas. Por este motivo, los desarrolladores noveles a menudo utilizan el método de programación alternativo basado en lo que se conoce como *polled communication*. Esta solución funciona bien con procesadores potentes, pero en el caso de los sistemas empotrados resulta necesario usar interrupciones para conseguir un funcionamiento óptimo.

No hay ningún impedimento en programar las ISR con un lenguaje de alto nivel como C, pero la solución óptima es usar lenguaje máquina. Todo ello dificulta su implementación.

Por regla general, cuando un procesador proporciona un servicio a una interrupción, sigue el proceso siguiente:

1) Un componente hardware genera una solicitud de interrupción que se envía al microprocesador.

2) El microprocesador prioriza las diferentes solicitudes que recibe y selecciona una.

3) El microprocesador responde al componente hardware con un acuse de recibo.

4) El componente hardware o un controlador responsable envía un vector de interrupción al microprocesador.

5) El microprocesador lee la dirección de memoria indicada en el vector, guarda la información necesaria para recordar su estado presente y salta a la ISR.

6) Se ejecuta la ISR y se gestiona la solicitud de interrupción.

7) La ISR finaliza y el microprocesador retoma su actividad inicial.

En aplicaciones reales, los programadores deben tener presentes las diferentes prioridades de las interrupciones que se pueden generar y también las características del software que integra su dispositivo para conseguir optimizar su funcionamiento.

# Bibliografía

**Ganssle, J. G.** (2000). *The Art of Designing Embedded Systems* (1.ª ed.). Woburn, Massachusetts: Newnes (Elsevier).

**IEEE** (2008). «IEEE Standard for Floating-Point Arithmetic». *IEEE Std 754-2008*, (n.º, págs. 1-70). doi: 10.1109/IEEESTD.2008.4610935.

**Stallings, W.** (2010). *Computer organization and architecture*. 8a ed.). Pearson Prentice Hall. ISBN 9780136073734.