U1 Desarrollo de software

1. Generalidades de los sistemas de información

Los sistemas de información o sistemas informáticos pueden ser extremadamente complejos, pero a la vez tienen características comunes. En este apartado se tratará de repasar algunas generalidades de los mismos relacionadas con el desarrollo de software. Como es ampliamente conocido, los sistemas informáticos están formados por hardware y software.

1.1 Software

Podríamos definir software como lo siguiente:

> **Software[1]:** conjunto de componentes lógicos de un sistema informático.

Pero ¿qué son los componentes lógicos? Son los elementos que, aun no siendo hardware, necesita el ordenador para funcionar. El hardware sin el software no se puede utilizar. El software define el funcionamiento del hardware, entendiendo el hardware como una máquina electrónica genérica, cuyo comportamiento no está definido previamente. En general, comprende datos e instrucciones (programas), pero también las ideas que los sustentan. El software necesita de algún medio material para tener existencia y desarrollar su funcionamiento. Ese soporte físico será el hardware.

A mí siempre me ha gustado una explicación que da un compañero: "Hardware es lo que se puede romper (haciendo un gesto con las manos, partiendo algo). El resto, es software."

[1] Basada en Wikipedia: https://es.wikipedia.org/wiki/Software

Tipos de software

¿Y qué elementos son software? Veamos algunos ejemplos y definiciones.

- El *sistema operativo* del ordenador es software. Ejemplos actuales: Windows, Linux, macOS, Android, iOS, … y sus versiones, Windows 8, Windows 10, Ubuntu Linux, Slackware, …. Gestiona los periféricos, la memoria, los programas en ejecución, los usuarios registrados en el sistema, …. Permite que el sistema informático sea utilizable. Por eso, en ocasiones se le llama *software de base*. Para manejar determinados elementos de hardware requiere de sus correspondientes *drivers*, que no son más que elementos de software con esta función, y pueden estar distribuidos con el sistema operativo o aparte de él (con el dispositivo que los necesita).
- *Firmware*: es el software que viene integrado en el hardware de fábrica (*firm*, en inglés). Aporta funcionamiento básico a los dispositivos. Los teléfonos móviles antiguos o el router de casa son dispositivos cuyo funcionamiento principal está basado en firmware. En ocasiones, el firmware puede ser actualizado o sustituido por otro; con el peligro que eso conlleva, claro: un equipo que se quede sin firmware válido perderá el funcionamiento.
- *Software empotrado o embebido*: es el que se encuentra en sistemas que no tienen sistema operativo. Por ejemplo, el que da funcionamiento a máquinas genéricas, como el hardware libre Arduino, o sistemas de control, ….

- *Aplicaciones*: es el software que ayuda al usuario, o que le permite realizar alguna tarea. Es decir, que le resulta de utilidad. Ejemplos son: el procesador de texto Microsoft Word, o el programa de edición de fotografías Adobe Photoshop, para ordenadores personales, pero también otras, como por ejemplo las de los dispositivos móviles: WhatsApp, Calendario, o juegos.
- *Utilidades*: este término puede ser similar al anterior (software de cierta utilidad, …). En ocasiones se utiliza para referirse a los programas usados para el mantenimiento del sistema u otras tareas genéricas: el desfragmentador, compresores de archivos, editor de texto plano, copias de seguridad, …
- *Software de desarrollo*: es el software que se utiliza para crear nuevo software. Compiladores, enlazadores, entornos de desarrollo integrados (IDE), herramientas CASE, ….
- *Los archivos*: los archivos de datos, sueltos, sin el programa que los trata, también son en esencia software.
- *Algoritmos y otra notación abstracta; la documentación del programa*: también pueden ser considerados como parte del software.

1.2 Hardware

Podríamos definir hardware como:

> **Hardware:** conjunto de elementos físicos de un sistema informático.

Como podemos imaginar, son infinidad de elementos: la pantalla, el ratón, pantalla táctil en el caso de los dispositivos móviles, pero también disco duro, procesador, memoria RAM, …; transistores, condensadores y LEDs. El conjunto de electrónica que forma el hardware puede funcionar de diferentes formas según el software que esté en ejecución. ¿Cómo es posible?

La mayoría de sistemas pueden caracterizarse con una misma estructura. Unos tendrán más procesadores (ahora, núcleos), más tipos de memoria, más o menos periféricos y de uno u otro tipo, pero cumplen con la estructura genérica de la figura 1.1. Está basada en tres grandes bloques: 1. Unidad Central de Proceso (CPU) o Procesador, 2. Memoria principal o RAM y 3. Todo lo demás: dispositivos de entrada/salida o periféricos. Los buses se encargan de comunicar esos elementos entre sí (datos, direcciones e información de control).

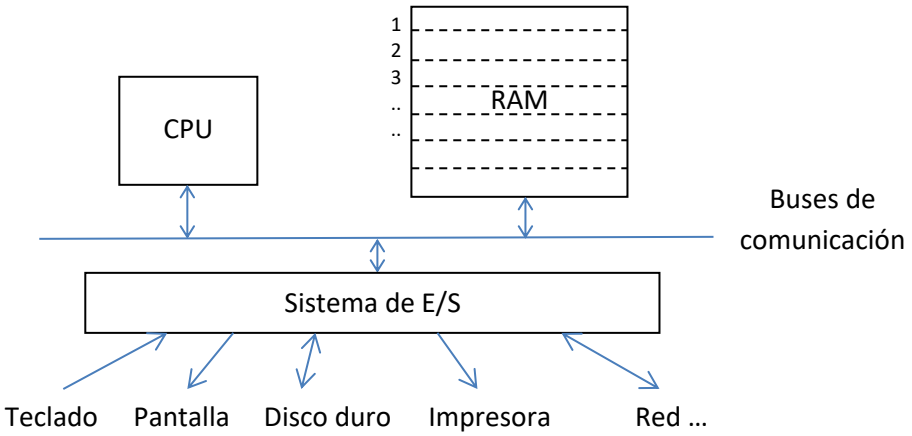
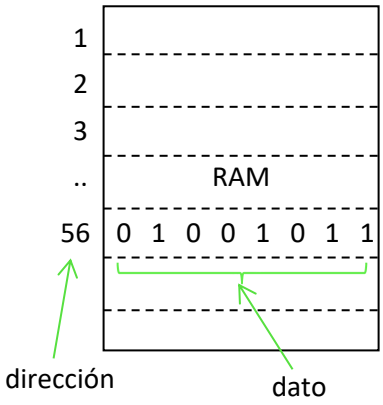


Figura 1.1. Tres bloques del hardware

La RAM

El funcionamiento básico de la *memoria principal* es muy sencillo. Está compuesto por celdas de bits, direccionables individualmente cada 8, 16, o, como en los ordenadores modernos, 32 o 64 bits. En cada dirección de memoria podemos escribir y leer datos, bits. Se llama RAM porque el orden en que se leen o escriben celdas es aleatorio (Ramdom Access Memory) y es una memoria de tipo volátil: cuando el ordenador se apaga pierde su contenido.



La CPU

La CPU o Unidad Central de Proceso es el corazón y el cerebro del ordenador. Es la que se encarga de ejecutar programas, y de efectuar cálculos. La primera de esas funciones está asignada a su subcomponente: la Unidad de Control. La segunda, a la ALU o Unidad Aritmético Lógica. Como su propio nombre dice, es capaz de hacer cálculos aritméticos y lógicos con los valores de los registros de la CPU. Los registros de la CPU son pequeñas unidades de almacenamiento, similares a la RAM, pero mucho más rápidos. Uno de ellos, el contador de programa, mantiene la dirección donde se encuentra la siguiente instrucción a ejecutar.

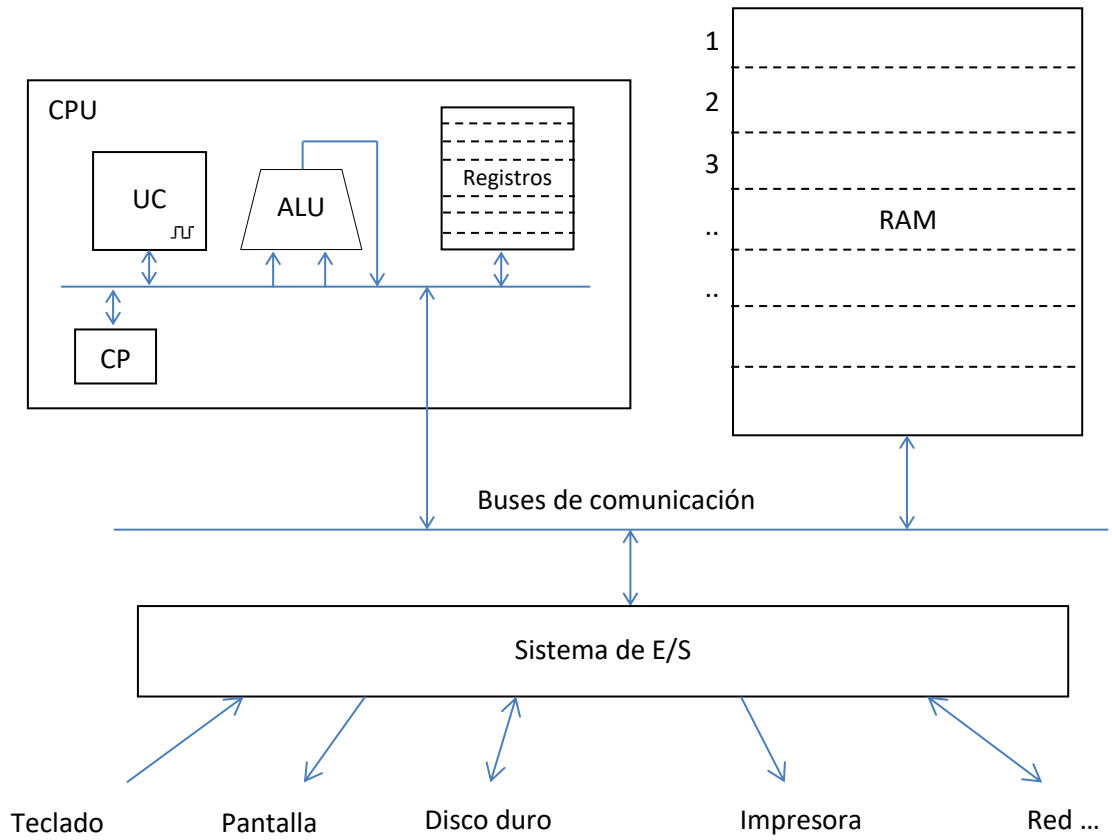


Figura 1.2. Ampliación con la estructura de la CPU

Entre los elementos de la CPU también hay buses de comunicación, igual que con el exterior. La UC sólo comprende un conjunto de instrucciones muy limitado, definido en el momento de

su construcción. Se le denomina *repertorio de instrucciones*. Suele ser suficientemente genérico para realizar cualquier tarea, puesto que contiene instrucciones de los tipos:

- Movimiento o copia de datos
- Aritméticas
- Comparación y comprobación (testeo)
- Manipulación de bits
- Desplazamiento y rotación
- Lógicas
- Control del flujo de ejecución del programa

Y en las CPU actuales otras más complejas, como:

- Operaciones multimedia
- Operaciones con múltiples datos de forma simultánea

1.3 Relación entre hardware y software

A alto nivel

Nada más encender un dispositivo comienza a ejecutarse su sistema operativo. En caso de no disponer de él, se ejecutará el firmware o sistema embebido que tenga almacenado. Una vez finalizado el arranque del sistema operativo, el usuario podrá hacer uso del dispositivo y ejecutar aplicaciones que tenga instaladas. También habrá procesos que estén ejecutándose en segundo plano, para mantener activos determinados servicios. Las aplicaciones deben ser específicas para la plataforma (sistema operativo sobre un hardware concreto) en que se ejecutarán. El sistema operativo es capaz de tratar con diferentes tipos de hardware a través de los *drivers*, o manejadores de dispositivo, que no son más que piezas de software con esta función, y específicas tanto del sistema operativo como del dispositivo.

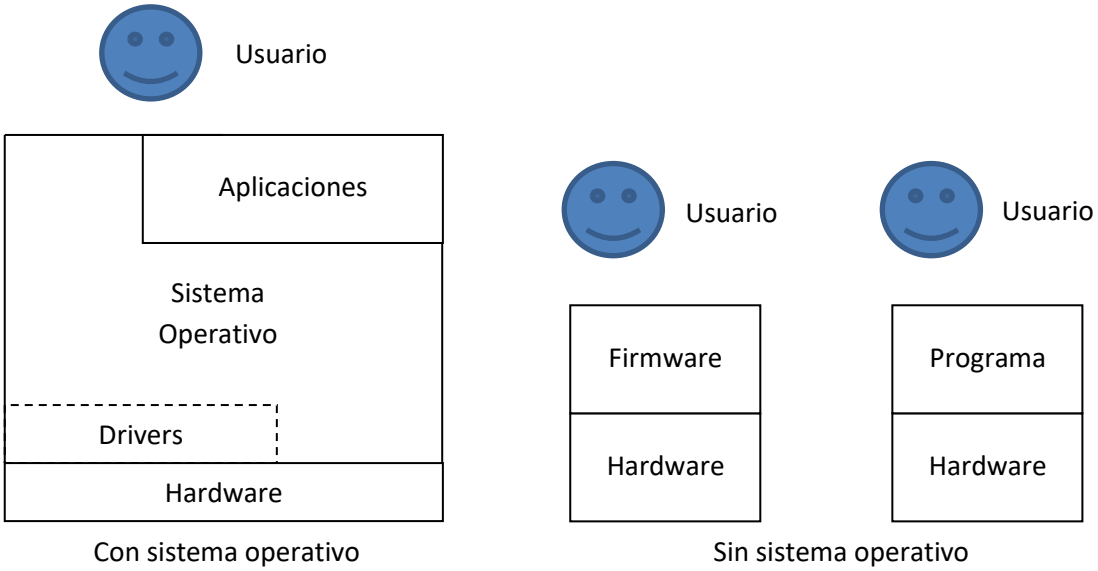


Figura 1.3. Relación entre hardware y software

Todo programa en ejecución debe estar, al menos en parte, en memoria principal. Al hecho de llevarlo desde el disco a la misma se le llama *cargar el programa*. Los programas cargados estarán compuestos de instrucciones en código máquina y datos expresados en binario.

A bajo nivel

El procesador repite continuamente lo que se denomina *ciclo de instrucción*. La instrucción que esté señalada por el registro denominado Contador de Programa es traída de memoria principal al procesador, en concreto a la Unidad de Control. Una vez le ha llegado a la Unidad de Control, ésta obedece la orden codificada en la instrucción, es decir, la ejecuta. Si es una orden de transferencia de datos entre la RAM y los Registros del procesador, así lo hará; si es una orden de utilización del sistema de Entrada y Salida se comunicará con el periférico correspondiente; si es una orden para hacer un cálculo, llevará datos de los Registros a la Unidad Aritmético Lógica y así conseguir el resultado deseado; etc.

Bien, pues aparte de la capacidad que tiene de atender interrupciones en el programa, eso es todo. A grandes rasgos, un ordenador no hace más que lo descrito. Por lo tanto, si se quiere que un programa funcione como se desea, lo único que se debe hacer es obtener de alguna forma el conjunto de instrucciones máquina y datos adecuados para el mismo.

1.4 Representando cualquier cosa: los bits

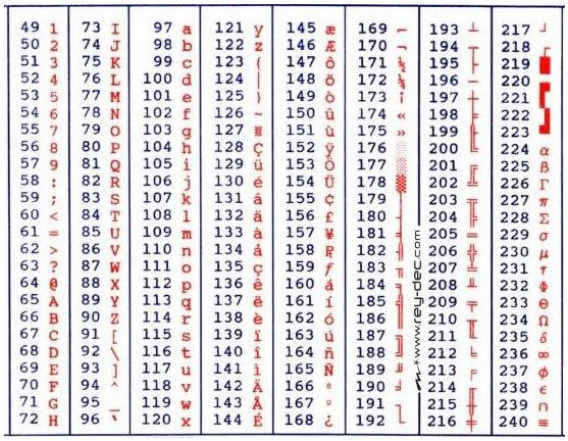
Un ordenador necesita almacenar y manejar instrucciones y datos, y los datos pueden ser de diferentes tipos: números enteros, caracteres simples, números reales, … y otros más complejos, es decir, cualquier tipo de dato: ¿es posible conseguirlo sólo mediante bits?

Un *bit* es el elemento básico de información. Su valor será o bien 0, o bien 1. La unión de ocho bits se denomina *byte*. Al tener ocho valores entre 0 y 1, su capacidad de representación es 2^8, es decir, 8 bits pueden producir hasta 256 combinaciones diferentes de ceros y unos. Si siguiéramos con las palabras, dobles palabras y palabras cuádruples, de 16, 32 ó 64 bits, respectivamente, su poder de representación sería de 2^{16}, 2^{32}, 2^{64}valores posibles. El primer resultado es de 65536 valores, el siguiente supera los cuatro mil millones de valores posibles, y el último podría identificar 18.446.744.073.709.551.616 cosas, exactamente (no quiero saber cómo se expresa de forma textual ☺). Con estos números lo que se ha buscado es mostrar la capacidad de representación de los bits: si queremos representar una serie de cosas, tan sólo necesitaremos utilizar el número de bits suficiente.

Algunos ejemplos:

- Números enteros: se hace por transformación directa entre base 2 y base 10: …0000 = 0; …0001 = 1; …0010 = 2; …0011 = 3; …0100 = 4; y así sucesivamente …; el signo positivo o negativo puede representarse con el primer bit: 0 es positivo y 1 negativo: 00000101 es un 5, 10000101 es un -5.
    - El signo en binario también se representa mediante complemento a 1 y complemento a 2 (= complemento a 1, + 1).
- Números reales: la representación es más compleja, pero p.ej. con el estándar IEEE 754 para coma flotante de 16 bits, se representa mediante 1 bit de signo, 7 de mantisa y 8 para el exponente. El exponente define el lugar de la coma que separa enteros y decimales en la mantisa. Así, el valor 53'2874 se puede representar mediante los bits: 0|10000100|1010101.

- Los caracteres sencillos se representan de forma muy sencilla: mediante tablas. Se asocia a cada símbolo, imprimible o no, un valor. Un ejemplo puede ser el código ASCII, o el Unicode. La siguiente es un fragmento de la tabla ASCII:



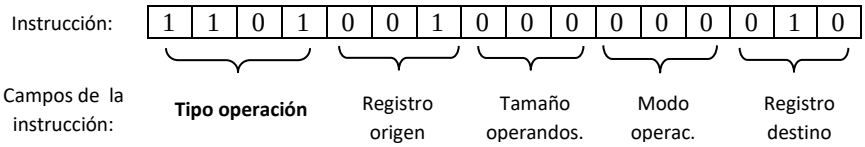
Y así, sucesivamente. Cada tipo de valor una forma de representación. Por ejemplo, agrupando varios valores de diferentes tipo, tendremos un valor de un tipo compuesto por esos campos. Y de esta forma podemos representar cualquier cosa. Las instrucciones máquina, las que entiende el procesador, también pueden almacenarse en RAM y en el disco duro, puesto que su formato original es un conjunto de campos, descrito cada uno en binario.

2. Programas y lenguajes de programación

2.1 Lenguaje máquina

Los programas en lenguaje máquina están formados un conjunto de instrucciones y datos en formato binario. Las instrucciones están definidas en binario. ¿Por qué? Las instrucciones almacenadas en memoria llegan a la Unidad de Control mediante los buses de datos, que nos los podemos imaginar como un conjunto de hilos. Si el bus es de 64 bits tendrá 64 hilos, y por cada uno de ellos se transmitirá o un 1 o un 0. El procesador tendrá 64 patillas para recibir cada bit, y en función de los valores recibidos comprenderá qué instrucción es y la ejecutará.

Veamos un ejemplo real, para el procesador de 16 bits M68000, de la marca Motorola. Su formato de instrucciones tiene varias opciones: o bien sólo contiene el código de la operación, o bien contiene el código de operación y un operando, o bien el código de operación y dos operandos. Así, por ejemplo, para el último caso:



es una instrucción que al ejecutarla suma el valor que haya en el registro R1 con el que haya en el registro R2 y el resultado lo deja en R2 (el segundo registro es a la vez origen y destino). El código de operación 1101 es para sumar, 001 significa que un operando es el registro R1 y 010 que el otro operando es el registro R2. Los campos de tamaño de los operandos y modo de operación no se explican ahora, por sencillez, dejándose a 0.

Algunos valores válidos de cada campo:

- Tipo operación: 1101; significa suma.
  Otros valores posibles: 1001 significa resta, 0100 significa mover bytes, …
- Registro origen: 001; significa 'registro1' o 'r1'.
  Otros valores posibles: 010 para 'registro2', 011 para 'registro3', …
- Tamaño operandos: 000; significa byte.
  Otros valores posibles: 001 significa palabra (word; 16 bits); 010 significa palabra larga (long word:32 bits)
- Modo operación: 000; significa 'Mediante registro de datos'
  Otros valores posibles: 001 significa 'Mediante registro de dirección'; 010 significa 'Relativo a registro de dirección'.
- Registro destino: 010; significa 'registro2' o 'r2'.
  Otros valores posibles: 001 para 'registro1', 011 para 'registro3', …

2.2 Lenguaje ensamblador

Ante la complejidad de lo que puede ser desarrollar código directamente en lenguaje máquina, en seguida se desarrollaron lenguajes simbólicos para facilitar la tarea de programación. Es más fácil recordar una palabra, por ejemplo ADD, que en inglés significa sumar, que el código 1101. El lenguaje más cercano a la máquina se denomina *ensamblador;* sin embargo, este programa ya no es comprensible por la máquina. Necesita ser traducido. Al proceso de traducción de un código escrito en lenguaje ensamblador a código máquina se le denomina *ensamblado.* El lenguaje ensamblador está compuesto, principalmente, por instrucciones similares a las del lenguaje máquina. Los lenguajes denominados macro-ensambladores incluyen alguna posibilidad más.

Veamos un ejemplo de lenguaje ensamblador para el mismo procesador M68000. Sumar los valores de los registros R1 y R2 y depositar el resultado en R2 podría codificarse así:

```
ADD.B D1, D2
```

Tamaño de los operandos: byte (B). Algunos otros símbolos de este lenguaje ensamblador:

- Tipo operación: ADD; significa suma.
  Otros valores posibles: SUB significa resta, MOVE significa mover bytes, …
- Registro origen: D1; significa 'registro1' o 'r1'.
  Otros valores posibles: D2 para 'registro2', D3 para 'registro3', …
- Tamaño operandos: B; significa byte.
  Otros valores posibles: W significa palabra (word; 16 bits); L significa palabra larga (long word: 32 bits)
- Modo de operación: En la sintaxis del registro destino: D2; significa 'Mediante registro de datos'
  Otros valores posibles: A1 significa 'Mediante registro de dirección A1'; (A1) significa 'Relativo a registro de dirección A1'
- Registro destino: D2; significa 'registro2' o 'r2'.
  Otros valores posibles: D1 para 'registro1', D3 para 'registro3', …

Ejercicios con lenguajes de bajo nivel:

1. Escribe un fragmento de código máquina que realice las acciones:

Sumar el valor que haya en el registro3 con el del registro1 guardando el resultado en el registro1.
Restar del valor del registro2 el valor del registro1. El valor del registro 2 queda actualizado.
Mover el valor del registro1 al registro3.

2. Codifica ahora las mismas instrucciones, pero en ensamblador.

2.3 Lenguajes de alto nivel

Como se puede deducir el lenguaje ensamblador, como el lenguaje máquina, es absolutamente dependiente del procesador que se programa. Habría que aprender un lenguaje ensamblador diferente por cada procesador sobre el que se quisieran hacer programas. ¿Cabría la posibilidad de crear lenguajes de programación independientes de la máquina y ejecutables en diferentes plataformas? La respuesta es que sí y así se hizo. A éstos, más cercanos al lenguaje del programador, se les denominó *lenguajes de alto nivel*, por tener un nivel de abstracción más alto respecto de la máquina que el ensamblador. Los lenguajes ensamblador y máquina quedarían entonces clasificados como *lenguajes de bajo nivel*.

En el momento histórico en el que se generaron supusieron un gran avance. Se dijo que eran "lenguajes de tercera generación". El código máquina sería de la primera y el ensamblador de la segunda. Algunos lenguajes de esta época son FORTRAN, COBOL, BASIC, C, PASCAL, etc. Posteriormente, se intentaron clasificar otros lenguajes como de cuarta o de quinta generación, aunque quizá sin el éxito deseado.

Sobre la idea de clasificar lenguajes entre alto y bajo nivel pueden establecerse grados. Por ejemplo, el lenguaje de programación C se considera de *medio-alto nivel*, ya que algunos de sus aspectos, como el tamaño de los datos del programa son dependientes de la plataforma de ejecución, lo que puede obligar a hacer adaptaciones en los programas.

Un ejemplo de programa en C, que imprime la frase 'Hola mundo!' en la interfaz de línea de comandos:

```
#include <stdio.h>

int main() {
    // Imprimir saludo
    printf("Hola Mundo!\n");
    return 0;
}
```

Figura 2.1 Código en C

2.4 Lenguajes compilados, a plataforma real

Está claro que si se programa en un lenguaje de alto nivel el código resultante no va a ser directamente ejecutable en ningún procesador: es necesario un proceso de traducción antes de poderlo ejecutar. Al proceso de traducir completamente un programa escrito en un lenguaje de alto nivel a código máquina se le llama *compilación*. En programación, compilar viene a ser sinónimo de traducir. El código escrito por el programador es el origen de la traducción, y se denomina *código fuente.*

En lenguajes como C, y otros, no todo el programa tiene que estar escrito en un único archivo de código fuente. Puede haber más de uno de esos archivos y además es común utilizar lo que se denominan *librerías estándar*, o tan sólo *librerías*. Este término, proviene del "false friend" inglés *library*, que significa biblioteca. Son recopilaciones de fragmentos de código de interés general, y que pueden ser aprovechados por los programas, ahorrando parte del trabajo.

Para conseguir un programa ejecutable deben compilarse cada uno de los archivos escritos en código fuente (en c, archivos con extensión .c) produciendo por cada uno de ellos un archivo

de *código objeto* (archivo con extensión .o ó .obj). Todos estos archivos deben unirse con los de las librerías (con extensión .a ó .lib) utilizadas por el programa para producir el *código ejecutable,* o tan solo, *ejecutable*. Al proceso de unión de unos códigos con otros se le denomina *enlazado* o *linkado* ('link', en inglés, significa 'unir'). Una de las extensiones de archivo para código ejecutable en entornos Windows es '.exe'.

En ocasiones, a todo el proceso se le denomina compilación, por brevedad. A los lenguajes que necesitan que sus programas sean compilados antes de ejecutarse se les denomina *lenguajes compilados.* El proceso, gráficamente:

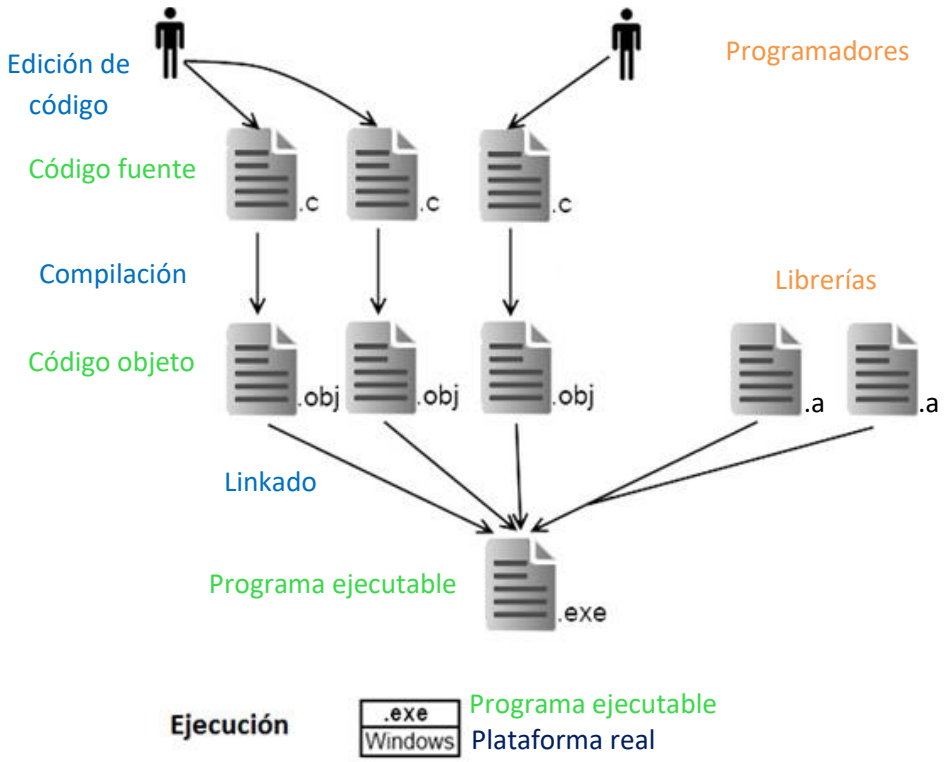


Figura 2.2 Proceso de compilación y ejecución en plataforma real

2.5 Lenguajes interpretados

Existe otra opción para conseguir ejecutar código de lenguajes de alto nivel diferente a la compilación. Se llama *interpretación*. En este caso la ejecución se produce directamente a partir del código fuente. No se requiere de archivos provenientes de traducciones, como el código objeto o el programa ejecutable. Lo que sí se requiere es de un programa que entienda y pueda ejecutar cualquier código escrito en el lenguaje determinado. A este programa se le denomina *intérprete*, al código fuente de estos lenguajes se le denomina *código interpretado* y a los lenguajes con esta característica, *lenguajes interpretados.*

Esta opción ralentiza la ejecución, puesto que el código que actúa como ejecutable es el mismo código fuente y, al no estar formado por instrucciones directamente ejecutables por el procesador, de alguna forma debe de ser traducido a estas en el momento de la ejecución, lo que requiere su tiempo. Por otra parte, estos lenguajes tienen como ventaja innata una mayor facilidad para la portabilidad de sus programas entre diferentes plataformas ya que, con tan sólo disponer del intérprete adecuado, esa plataforma podrá ejecutar sus programas.

Entre los lenguajes clásicos, por poner un ejemplo, BASIC era de tipo interpretado. Hubo un cierto vacío posterior de este tipo de lenguajes debido en parte al perjuicio de la velocidad, pero ahora parece que ha habido un nuevo repunte, quizá por la mayor velocidad de los dispositivos actuales, que compensa la desventaja. Entre los lenguajes interpretados actuales encontramos Javascript (extensión .js), HTML o PHP, propios de las aplicaciones web, o Python (extensión .py), exitoso lenguaje de propósito general.

Figura 2.3 Necesidad de intérprete

2.6 Lenguajes compilados a máquina virtual

Quizá haya una solución intermedia entre ambos tipos de lenguajes. ¿Es necesario perder tanto tiempo en la interpretación del código fuente? ¿Tenemos que perder las posibilidades multiplataforma de la interpretación? "Si todas las máquinas fueran iguales, podríamos ejecutar el código compilado en todas ellas…"

¿Habrá alguna máquina ideal sobre la que realizar la ejecución? Pues no, pero podemos inventarla. De hecho, en este mismo tema se ha ejemplificado una máquina genérica. La idea es la siguiente. Si ideamos esa máquina genérica podríamos generar código para la misma mediante compilación. Luego, esa máquina podríamos simularla en otra y ejecutar los programas que hemos compilado para ella.

A la máquina inventada se le llama *máquina virtual.* El código generado para esa máquina debe de ser un código muy parecido a un código máquina real, para que el proceso de traducción sea muy rápido. A ese código se le denomina *bytecode* o *código intermedio* y, puesto que es resultado de una compilación previa, su interpretación es mucho más rápida que la de los lenguajes puramente interpretados. Sin embargo, es lenta en comparación con la de los lenguajes compilados para la máquina real.

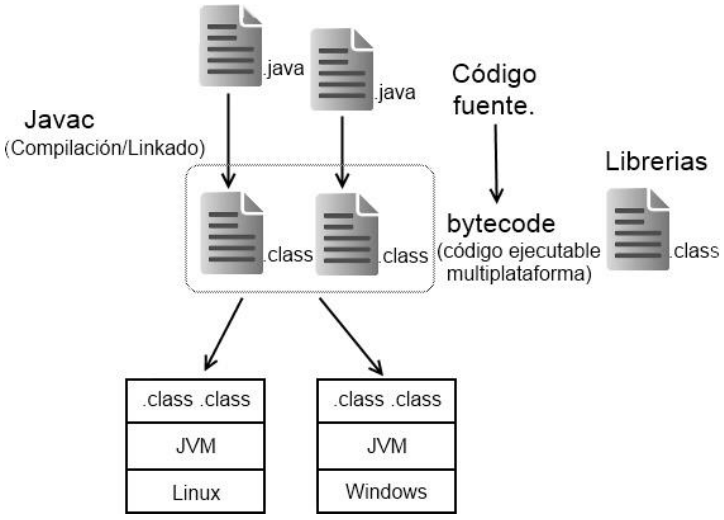


Figura 2.4 Compilación y ejecución en máquina virtual

Este sistema se popularizó mucho. Quizá encuentre en el lenguaje Java su mayor exponente. Microsoft adoptó la misma solución para su conjunto de lenguajes de la tecnología .NET: Visual Basic.NET, C++.NET, C#.NET, etc.: todos se ejecutan sobre un entorno virtual llamado CLR

(Common Language Runtime). C# es prácticamente una copia de Java. Los programas de código fuente de java suelen tener la extensión .java, sus bytecode .class y su entorno de ejecución es el JRE (Java Runtime Environment). El JRE incluye la JVM (Java Virtual Machine), ver figura 1.7, junto a las librerías (Figuras 1.7 y 1.8).

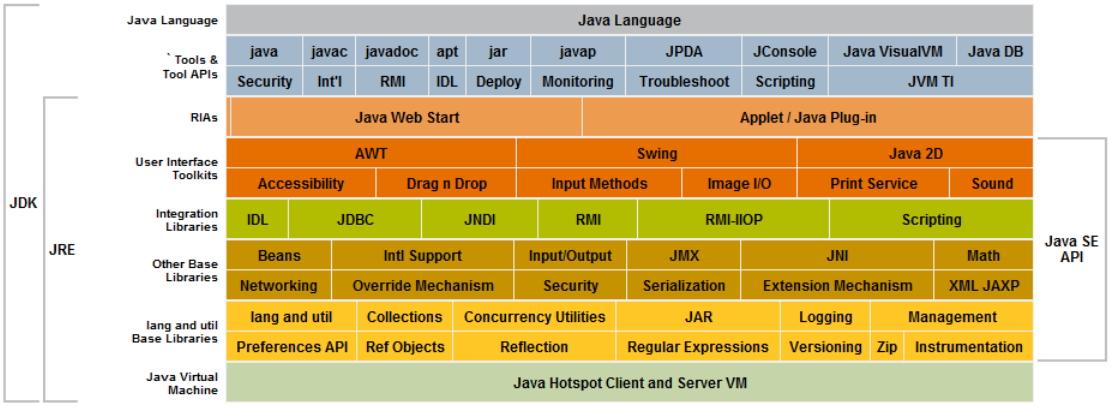


Figura 1.8. JDK, JRE y JVM

2.7 Paradigmas de programación

La forma de programar depende de la concepción que se tenga sobre cómo son, o deben de ser, los programas. A esta perspectiva se le denomina paradigma de programación. Repasemos algunos de ellos.

Paradigma imperativo

Está basado en la forma de funcionar real de los ordenadores. Los programas están compuestos por instrucciones y datos; el comportamiento correcto depende del orden en que se ejecutan las instrucciones, que definen los cambios de estado de las variables.

Al tremendo desorden que puede producirse al programar con saltos entre las instrucciones le trata de poner solución una disciplina que se denomina *programación estructurada*. El *paradigma procedimental* o *procedural* promueve la división del procedimiento general en unidades más pequeñas (procedimientos).

La mayoría de lenguajes que en la actualidad, extensamente, se consideran lenguajes de programación tienen características imperativas: C, Java, BASIC, C++, C#, aunque no sólo éstas.

Paradigma declarativo

Como contraposición al imperativo, los programas no están compuestos por órdenes sino por declaraciones. Es decir, el programador, en vez de especificar cómo obtener el resultado, describe cuál es el resultado que desea obtener y la máquina se encarga de producirlo. Se considera que están más cercanos al programador que los imperativos, y por lo tanto, deberían ser más sencillos de utilizar.

Ejemplos actuales de este tipo de lenguajes pueden ser HTML o SQL.

Paradigma funcional

En realidad, es un subtipo del paradigma declarativo. En el paradigma funcional los programas están compuestos por funciones al estilo matemático, con datos de entrada y salida. Programar supone definir las funciones necesarias, que muchas veces se hará en función de

otras. En este paradigma se usa mucho la recursividad, esto es, la definición de una función refiriéndose a sí misma. Ejecutar un programa supone realizar una invocación de la función que se desee, incluyendo unos valores como entrada.

Como ejemplos de lenguajes con características funcionales podemos mencionar a LISP, en el mundo de la inteligencia artificial, o los más utilizados actualmente Scala, Erlang, Haskell o R.

Paradigma lógico

También subtipo del paradigma declarativo, los programas del paradigma lógico están compuestos por una serie de hechos y reglas lógicas. Una vez comunicado al sistema este "conocimiento", es capaz de resolver preguntas de forma autónoma gracias a un motor lógico.

Como ejemplo prototípico de este paradigma tenemos a Prolog.

Paradigma orientado a objetos

En este paradigma los programas están compuestos por objetos de diferentes clases, que interaccionan entre sí por medio de mensajes para satisfacer los requisitos del usuario.

Es un paradigma actual, que engloba soluciones dadas por otros paradigmas y con un nivel de abstracción muy cercano a la forma que tiene el programador de ver las cosas en el mundo real.

La mayoría de lenguajes actuales importantes permiten programar desde el punto de vista de este paradigma. Ejemplos: Swift (para iOS y macOS), Java, Python, C++, C#, etc.

Lenguajes multiparadigma

Muchos de los lenguajes existentes son capaces de satisfacer los requisitos de programación de varios paradigmas. Será el programador el que elija desde qué enfoque programar.

3. Aplicaciones informáticas

Sea como fuere, el mayor ámbito en que se desarrolla la programación es el que tiene que ver con los usuarios estándar, realizando programas que los ayuden en sus tareas profesionales, en su vida diaria o que los entretengan, y que se instalan y ejecutan sobre algún sistema operativo previamente instalado. Vamos a ver algunos tipos de aplicaciones. En ocasiones, los términos aplicación y programa se utilizan indistintamente.

El primer tipo de aplicaciones a nombrar son las **aplicaciones de escritorio**. Ejecutadas sobre un sistema operativo, con entorno gráfico como el de los ordenadores personales, se relacionan con el usuario mediante elementos como ventanas, cuadros de diálogo, cuadros de texto y botones, y pueden abrirse varias ventanas o incluso aplicaciones al mismo tiempo, lo que en conjunto permite una interactividad muy alta y resulta un entorno agradable y muy intuitivo para el usuario. No se necesita estar conectado a la red para ejecutarlas, ya que, salvo las que tienen como requisitos comunicarse con otros ordenadores, utilizan principalmente los recursos de la máquina en que se ejecutan.

El segundo tipo son las aplicaciones con **interfaz de línea de comandos**. Sólo se produce entre el usuario y el programa un intercambio de líneas de texto, lo que por otra parte guía su flujo de ejecución. Este es el entorno que tenían los ordenadores antiguos y podría pensarse que

está en desuso, sin embargo, los administradores de sistemas y los programas que no requieran de una interfaz gráfica para funcionar, como pueden ser los servidores o los del lado del servidor, muchas veces lo utilizan.

Aplicaciones distribuidas. Son aplicaciones que requieren de software ubicado en diferentes ubicaciones geográficas. Todos sus elementos de software no están en el mismo equipo informático y para funcionar deben poder interactuar entre sí.

Aplicaciones web. Son un tipo de aplicaciones distribuidas que hacen uso, para funcionar, del protocolo de transferencia de páginas web. Las páginas web se originaron con un formato de presentación sencillo basado en el lenguaje HTML. Estas páginas estaban almacenadas en servidores y se accedía a ellas a través de programas cliente llamados navegadores. La característica principal de HTML es la capacidad de referirse a otras páginas y objetos mediante desde el propio documento, lo que puede visualizarse mentalmente como si fuera una telaraña (en inglés, web). El éxito de la web, su ámbito mundial y la impresión de ser servicio deslocalizado (accesible desde cualquier lugar) son factores que han provocado que, poco a poco, se fueran desplazando cada vez más servicios a "la nube", a los servidores web. Estas aplicaciones tienen unas capacidades multiplataforma prácticamente absolutas, ya que para acceder a ellas sólo se necesita que el sistema tenga acceso a Internet y un navegador web compatible.

Algunos lenguajes libres utilizados para desarrollar aplicaciones web son, además de HTML para la estructura de las páginas y CSS para el estilo, PHP para la ejecución en el lado del servidor y Javascript para la ejecución en el lado del cliente. Lenguajes de datos, XML y JSON. Entre otros.

Aplicaciones móviles. El auge del uso de dispositivos móviles con capacidades avanzadas, como smartphones y tablets, ha hecho que prácticamente cada usuario lleve uno encima, o tenga alguno cerca. Esto ha favorecido las necesidades de desarrollo de aplicaciones para estos dispositivos. Ya con sistema operativo, sus aplicaciones se suelen denominar con el diminutivo ***app***, por ser más pequeñas que las de ordenadores personales. A su vez suelen clasificarse en tres tipos:

- **Aplicaciones nativas.** Similares a las de escritorio, pero en los dispositivos móviles. Se ejecutan utilizando los recursos, e interfaz gráfica del dispositivo, principalmente. Sin uso de la web.
- **Aplicaciones web.** Lo mismo que las de los ordenadores personales, utilizan un navegador para acceder a la aplicación.
- **Híbridas.** Tienen un aspecto o interfaz de aplicación nativa, pero internamente utiliza la tecnología de la web para dar su servicio.

Sistemas operativos para dispositivos móviles que han resultado exitosos tenemos Android, como software libre, o iOS, de Apple. Sus aplicaciones nativas son incompatibles entre sí.

Clasificación por el ámbito de la aplicación

En este sentido, las aplicaciones se dividen, a muy grandes rasgos entre **aplicaciones verticales** y **aplicaciones horizontales**. Las primeras también se llaman de *propósito específico*, puesto que atienden necesidades de un campo muy concreto y suelen profundizar en este aspecto.

Ejemplo: un programa que maneja las máquinas de una fábrica. Las segundas también se les denomina *de propósito general*, ya que resultan de aplicación en ámbitos diversos, aunque suele ser de forma más superficial. Ejemplo: un procesador de textos, o una hoja de cálculo.

4. Licencias de software

El software desarrollado tiene autoría y, por lo tanto, derechos asociados: p. ej. derechos de uso, de acceso al código fuente, de modificación, o de distribución. Según la licencia otorgada por el autor para cada uno de esos derechos, nos encontramos diferentes tipos de aplicaciones y software:

- **Freeware**. Software de libre uso y distribución, sin costo alguno. No es lo mismo que software libre u open source. Mantiene los derechos sobre el código fuente. No tiene por qué tenerse acceso al mismo, ni tener derechos de modificación del programa.
- **Shareware**. Software de prueba con funcionalidades limitadas, o limitación en el tiempo de uso, con el objetivo de difundir la aplicación y que el usuario compre la versión completa.
- **Software de pago**. Software con objetivos económicos. Es necesario pagar para utilizarlo. Normalmente, no está permitida su distribución libre, ni el acceso a su código, ni su modificación.
- **Open source**, **software de código abierto** y **software libre** (no confundir con freeware): con algunas matizaciones, son licencias que dan acceso al código fuente y está permitida su modificación.
- Software bajo licencia **GNU GPL**. En general, de libre uso, libre distribución y acceso libre al código, pero, además, con la restricción de que si todo o parte del software es utilizado en alguna aplicación, ésta debe distribuirse también bajo licencia GNU.