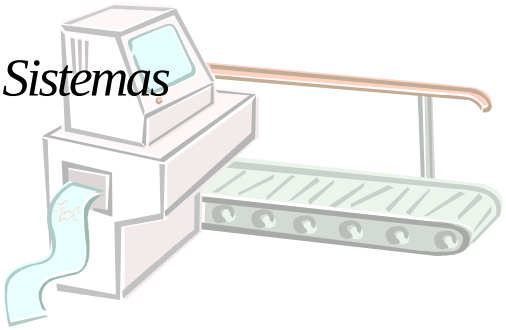
e-mail: eliseo.garcia@uah.es
Despacho: N-245
Departamento de Automática

Práctica 6

Entorno de Desarrollo Linux

Desarrollo de Aplicaciones para Sistemas Industriales



Entorno de Desarrollo Linux

Introducción

En este documento se tratará de la instalación del Sistema Operativo Raspbian (una versión de Debian adaptada para Raspberry Pi) en una Raspberry Pi utilizando una tarjeta de memoria SD (dicha tarjeta debe tener un tamaño nunca inferior a 4Gb) y de la instalación de las herramientas necesarias para desarrollar aplicaciones sobre ella (Entorno de Desarrollo).

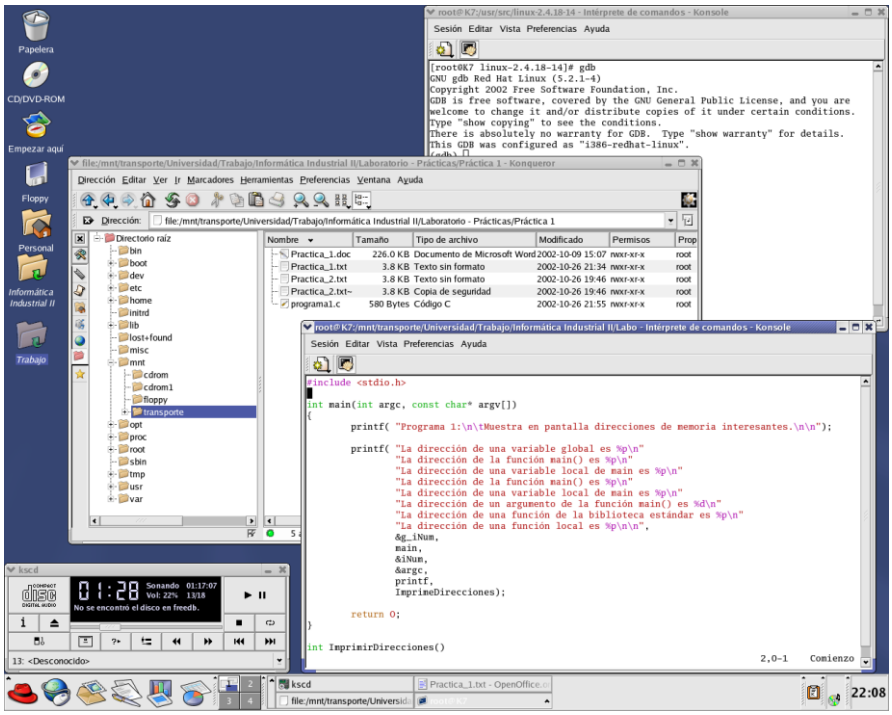
Raspberry Pi es una computadora de bajo costo construida en una placa del tamaño de una tarjeta de crédito, desarrollada en Reino Unido por la Fundación Raspberry Pi, con el objetivo de estimular la enseñanza de ciencias de la computación en las escuelas. Utiliza un chip Broadcom BCM2835, el cual contiene un procesador (CPU) ARM1176JZF-S a 700 MHz (aunque el firmware incluye unos modos "Turbo" para que el usuario pueda hacer overclock hasta 1 GHz sin perder la garantía), un procesador gráfico (GPU) VideoCore IV, y 512 MB de memoria RAM.

Esta computadora es excelente tanto para proyectos de investigación con sistemas embebidos, automatización de sistemas, creación de servidores minimalistas, etc. como para simplemente jugar en casa.

Tradicionalmente UNIX ha sido considerado como uno de los SS.OO. mejor dotados para el desarrollo de software. Cuenta con un conjunto de herramientas de desarrollo con bastante solera. No en vano el exitoso lenguaje C nació fuertemente ligado a UNIX. El hecho de conservar compatibilidad hacia atrás permite a los desarrolladores centrarse más en aspectos de diseño que en la forma de usar las herramientas.

Desde que a finales de los '80 las utilidades GNU de la *Free Software Foundation* alcanzaran un nivel de madurez suficiente, se han establecido como un estándar de facto para el desarrollo de software y el mantenimiento del sistema, y han trasgredido las fronteras del propio UNIX que las vio nacer. En la actualidad se dispone de ellas en casi cualquier plataforma informática de entidad suficiente.

El compilador cruzado de C **`gcc`** es con total seguridad su aplicación principal. Está portado a más de 10 tipos diferentes de procesador, y adaptado para generar código para muchos SS.OO. La posibilidad de contar con un potente compilador de C compatible ANSI/ISO ha permitido migrar a su vez toda una legión de aplicaciones de soporte: diferentes tipos de intérpretes, utilidades de interfaz de usuario, editores de texto, analizadores, ensambladores, bibliotecas de programación, etc. Diferentes *front-ends* para **`gcc`** permiten desarrollar aplicaciones en Objective-C, C++, Pascal, Fortran y Java.



Gracias a la existencia gratuita y acompañada de código fuente de estos programas se ha propiciado la aparición de otros muchos proyectos de software de diferente envergadura entre los que podemos citar el núcleo Linux, el S.O. FreeBSD (creados ambos al estilo UNIX), el sistema de ventanas XFree86, los gestores GNOME y KDE, etc.

Instalación

Ahora que ya sabemos de qué se trata Raspberry Pi, lo siguiente es instalarle un sistema operativo, por supuesto vamos a instalar GNU/Linux. Existen muchas distribuciones a medida de la Raspberry Pi, pero en este laboratorio vamos a utilizar Raspbian.

Raspbian es una distribución (por supuesto libre) basada en Debian optimizada para correr sobre el hardware de la Raspberry Pi. Procedamos, pues, a mostrar cómo se realiza la instalación de este sistema operativo en una tarjeta de memoria micro-SD.

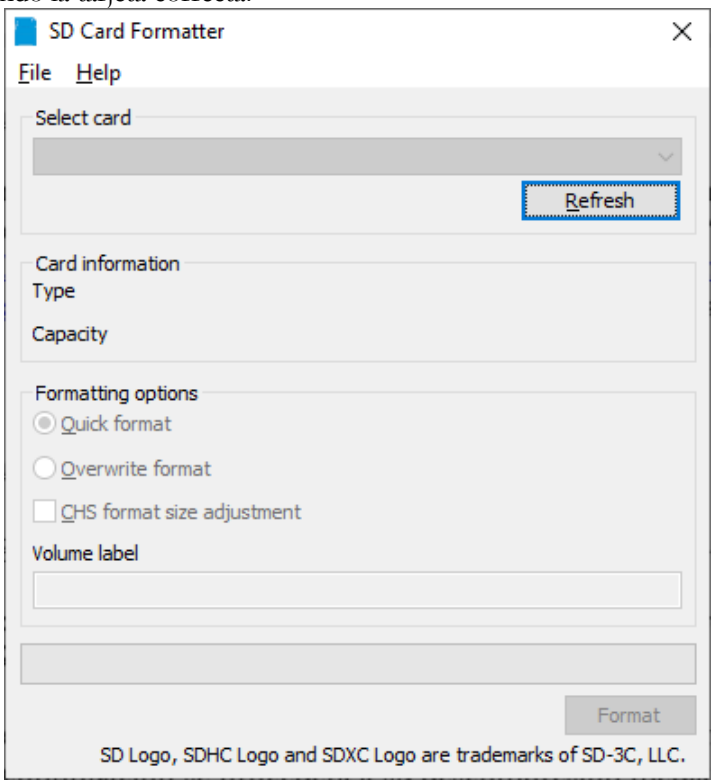
En primer lugar, se debe descargar el sistema operativo Raspbian. Ese sistema operativo se ofrece a todo usuario en la web oficial de raspberrypi, que es http://www.raspberrypi.org. La Raspberry Pi Foundation nos da además la posibilidad de descargarnos e instalar el ultimo Sistema Operativo Raspbian disponible para cada una de los modelos de Raspberry Pi de una forma sencilla, a través de un software llamado Imager (a través de su web https://www.raspberrypi.com/software/) disponible para Windows y Linux.

Debido a que las herramientas que vamos a usar para el desarrollo del laboratorio muestran incompatibilidades con la última versión de Raspbian para nuestra tarjeta Raspberry Pi 1 B+, vamos a usar una versión de Raspbian anterior, por lo que no podremos utilizar la herramienta Raspberry Pi Imager. La versión que vamos a utilizar en el laboratorio, compatible con nuestra tarjeta Raspberry Pi B+ y con nuestras herramientas de trabajo, la podemos obtener en el enlace http://downloads.raspberrypi.org/raspbian/images/raspbian-2015-11-24/2015-11-21-raspbian-jessie.zip

Para la instalación del Sistema Operativo que usaremos en el laboratorio en la tarjeta SD, podemos utilizar dos métodos en función del Sistema Operativo que tengamos instalado en nuestro equipo de trabajo

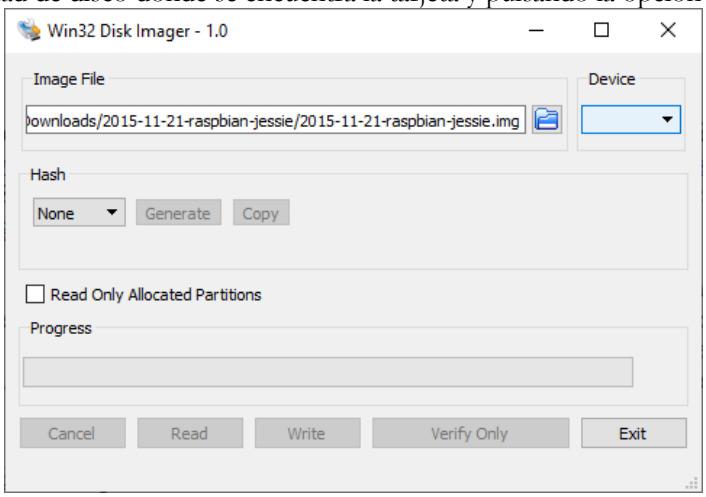
- Desde un Sistema Operativo Windows

  En primer lugar, nos aseguramos que la tarjeta está correctamente formateada. Para realizar el formateo de la tarjeta SD de una manera sencilla, podemos utilizar el programa "SD Card Formatter", disponible gratuitamente en https://www.sdcard.org/downloads/formatter/sd-memory-card-formatter-for-windows-download/. Una vez instalado, insertamos la tarjeta SD que se utilizará, y se ejecuta el programa, seleccionando la tarjeta correcta.



A continuación, nos descargaremos el fichero que contiene el sistema operativo utilizando el enlace http://downloads.raspberrypi.org/raspbian/images/raspbian-2015-11-24/2015-11-21-raspbian-jessie.zip. Se obtendrá un fichero comprimido zip, que procederemos a descomprimir, obteniendo un fichero .img que es ya la imagen de Sistema Operativo a instalar en la tarjeta.

Para copiar el fichero en la tarjeta, se puede utilizar el programa win32diskimager, que se encuentra disponible en https://win32diskimager.org/. Una vez descargado e instalado, se inserta la tarjeta SD en el equipo y se ejecuta el programa, escogiendo el fichero .img previamente descargado y descomprimido, seleccionando la unidad de disco donde se encuentra la tarjeta y pulsando la opción write.



Una vez finalice la copia, ya tendremos en la tarjeta el sistema operativo que utilizará nuestra tarjeta Raspberry Pi. Expulse el dispositivo e insértelo en la tarjeta.

- Desde un Sistema Operativo Linux

Nos podemos descargar el fichero que contiene el sistema operativo mediante el comando

```
$ wget http://downloads.raspberrypi.org/raspbian/images/raspbian-2015-11-24/2015-11-21-raspbian-jessie.zip
```

con lo que se procede automáticamente a la descarga. El fichero que obtenemos es un fichero comprimido con zip, con lo que a continuación se procederá a su descompresión mediante el comando

```
$ unzip ficherodescargado.zip
```

Tras la descarga se obtendrá un fichero imagen con extensión .img. Este fichero contiene ya una imagen del Sistema Operativo, y únicamente se deberá copiar en la tarjeta SD. Para eso se inserta dicha tarjeta en una ranura SD del ordenador (o en un adaptador microSD- USB, en el caso de que el ordenador no la tenga) y se realiza la copia. Fíjese al insertar la tarjeta el punto de montaje que se crea, es decir, que archivo del directorio /dev/ se utiliza para hacer accesible la tarjeta (suele ser /dev).

Para la copia, podemos utilizar el comando dd, que realiza la copia desde un origen a un destino. Este comando se diferencia del comando cp en que la copia se realiza en bloques. Tras la ejecución de este comando, todos los datos almacenados en la tarjeta se perderán, con lo que se recomienda previamente guardar esos datos en otro dispositivo que no sea la tarjeta microSD.

```
$ sudo dd if=ficherodescomprimido.img of=/dev/xxxx
```

Tras la ejecución de este comando ya tendremos en la tarjeta el sistema operativo que utilizará nuestra tarjeta Raspberry Pi. Desmonte o expulse el dispositivo e insértelo en la tarjeta.

Configuración

A continuación, se va a proceder a la configuración de ciertos aspectos del Sistema Operativo. Procedemos instalar teclado, ratón, monitor y red a la placa, y a continuación damos alimentación en el puerto microusb.

Tras una carga inicial, escribiendo en consola el comando raspi-config se nos presenta en la pantalla una utilidad de configuración (esa utilidad se llama raspi-config y se puede ejecutar en cualquier momento).

De entre las opciones que tenemos, vamos a destacar las siguientes

Expansión del sistema de archivos.

Seleccionando esta opción, solicitaremos que el sistema haga uso del total de la capacidad de la tarjeta micro-SD. Esta opción es importante porque nos permitirá tener más capacidad de almacenamiento para nuestro trabajo en Raspberry Pi. Este nuevo tamaño del sistema de ficheros estará accesible para nosotros la siguiente vez que iniciemos el sistema.

Cambio de la contraseña.

Raspbian define un usuario por defecto, el usuario *pi*. Su contraseña por defecto es *raspberry*. Mediante esta opción cambiaremos la contraseña.

Seleccionar arranque en un escritorio.

Permite que el usuario arranque directamente en un entorno de ventanas. Si no seleccionamos esta opción, se arrancará en un terminal y podremos acceder al entorno de ventanas sin más que ejecutar el comando **startx**

Configuración Regional.

Permite configurar Raspberry Pi para que tenga en cuenta nuestra región es aspectos como idioma, reloj, configuración del teclado, … Dentro de las opciones de configuración de teclado, nos permite también habilitar la combinación de las teclas `Ctrl+Alt+Backspace` para finalizar el servidor de ventanas X.

Habilitar cámara.

Se habilita esta opción en el caso de que vayamos a trabajar con la cámara que ofrece Raspberry Pi.

Opciones avanzadas.

En este apartado se permite la configuración de una serie opciones adicionales. Para nuestro caso, debido a que vamos a acceder a Raspberry Pi desde nuestro PC, es muy útil tener habilitado en nuestro sistema objetivo (Raspberry Pi) un servidor ssh que nos permita ejecutar comandos en ella y realizar conexiones desde el PC. Por lo tanto, seleccionamos esta opción, y en el menú siguiente seleccionamos Habilitar el servidor SSH.

Tras configurar estas opciones, seleccionaremos el botón Finish, con lo que se procederá al reinicio del sistema para aplicar las configuraciones modificadas.

Tras el arranque del sistema, vamos a proceder a configurar la red. Para eso, desde el interfaz de comandos (en caso de que hayamos seleccionado arrancar en él por defecto) o desde el programa LXTerminal, modificaremos el fichero /etc/network/interfaces. Esto lo tenemos que hace con los permisos de super-usuario, para lo cual utilizaremos el comando sudo.

```
$ sudo nano /etc/network/interfaces
```

Nosotros vamos a conectarnos a la red Ethernet mediante una dirección IP estática, con lo que deberemos modificar este fichero añadiendo las líneas.

```
allow-hotplug eth0
iface eth0 inet static
    address  172.29.a.b
    netmask  255.255.252.0
    gateway  172.29.20.1
```

(preguntar al profesor del laboratorio el valor concreto de a y b para cada alumno y cada puesto).

Guardar este archivo y cerrarlo. Tras el siguiente reinicio ya estaría disponible esta configuración automáticamente, pero si queremos activarla sin necesidad de reinicio, se pueden ejecutar los siguientes comandos.

```
$ sudo ifdown eth0
$ sudo ifup eth0
```

Entorno de Desarrollo

Como conocemos, un entorno de desarrollo contiene todas las herramientas necesarias para la creación de aplicaciones. Las herramientas más habituales de un entorno son el Editor (permite la creación de programas siguiendo la sintaxis especificada por un Lenguaje de Programación), Compilador (realiza la traducción del programa editado a su equivalente en lenguaje máquina), Depurador (permite la validación del programa realizado en un entorno pseudoreal) y Lanzador (pone en ejecución el programa desarrollado en el sistema final). A continuación, se procede a describir los elementos más habituales que ofrece Linux para cada una de las fases

El editor

La edición de textos es uno de los pilares básicos sobre los que necesariamente se asienta la codificación de nuevos programas. Actualmente, cada distribución de Linux acompaña entre sus herramientas uno o varios editores de textos que permite la realización de programas.

El editor vi es uno de los más populares dentro del mundo UNIX. En la actualidad se suele usar una versión más moderna compatible llamada vim. Sus principales bazas son la compatibilidad hacia atrás y su potencia como editores desde el punto de vista de la funcionalidad, además de que permite la edición de textos desde consola, sin uso de un entorno de ventana de X-Window.

El compilador de C `gcc` (GNU C Compiler)

El programa **`gcc`** es la utilidad que emplearemos para generar ejecutables a partir de código C. Desde este podremos preprocesar directivas, traducir el C a ensamblador, ensamblar, y enlazar, todo en uno y de forma transparente. La sintaxis es:

```
gcc [opciones] fichero...
```

Por ejemplo, edite el siguiente programa, y llámelo `programa0.c`:

```
#include <stdio.h>

int main()
{
```

```
        puts("Este es un programa de C.\n");

        return 0;
}
```

Para compilarlo tan solo será necesario introducir lo siguiente:

```
$ gcc programa0.c
```

lo que creará un archivo ejecutable llamado `a.out` que podremos cargar de la forma habitual:

```
$ ./a.out
Este es un programa de C.
```

El formato de salida del compilador es compatible con el S.O. para el que se esté compilando. En el caso de GNU/Linux el formato será ELF, que es comprendido por el programa cargador del sistema.

Si queremos especificar un nombre distinto al que se toma por defecto, lo podemos especificar con el modificador `-o` de la siguiente forma:

```
$ gcc -o mensaje programa0.c
```

lo que indica que el nombre del archivo de salida sea `mensaje` en lugar de `a.out`.

Si quisiéramos que gcc sólo generase el código objeto correspondiente a este único archivo sin enlazarlo (linkarlo) con ningún otro, usaríamos el modificador –c.

```
$ gcc –c -o programa0.o programa0.c
```

Para enlazarlo con la biblioteca usaremos el siguiente comando:

```
$ gcc -o programa0 -lm programa0.o
```

Depuración de programas con gdb (GNU De-Bugger)

Los depuradores son utilidades que ayudan a eliminar fallos de los programas en tiempo de ejecución, una vez que se han eliminado los fallos de sintaxis en tiempo de compilación. Por lo general se suele asumir que los fallos de los programas se encuentran únicamente en el código, ya sea por un diseño deficiente o una codificación incorrecta. La verdad es que eso no siempre es así. A menudo los errores están presentes en los documentos o en los datos externos que maneja el programa, normalmente porque no se adapten a un convenio estipulado. Esto no exime de la responsabilidad de desarrollar programas robustos que puedan reponerse ante datos de entrada anormales.

La aplicación `gdb` se ha erigido por méritos propios como uno de los mejores depuradores a nivel de código fuente existentes. Es apto tanto para depurar código como datos. Aunque se trate de un programa que funciona en consola de texto, está preparado para comunicarse con otros programas que le pueden proporcionar una interfaz más amigable.

La Herramienta make

La herramienta *make* facilita la compilación de proyectos, y es una herramienta estándar instalada en todo sistema UNIX. Este programa permite efectuar una compilación inteligente de programas, en función de los archivos modificados que necesitan realmente ser compilados. Según se indica en la manual de make, el propósito de esta utilidad es determinar automáticamente que piezas de un programa necesitan ser recompiladas y de acuerdo a un conjunto de reglas llevar a cabo las tareas necesarias para alcanzar el objetivo definido, que normalmente es compilar un programa. No obstante, la sintaxis de los archivos *Makefile* es relativamente compleja de escribir porque utiliza reglas de reescritura, aunque aquí no las veremos todas en detalle.

Al ser ejecutado, *make* utiliza el archivo *makefile* o *Makefile* situado en el directorio actual, analiza su contenido y lanza las ordenes indicadas. Esta herramienta es muy práctica porque permite, entre otras cosas, recompilar sólo lo necesario. Si algún fichero de código sufre alguna modificación, sólo será recompilado aquel que fue modificado, más todos los programas que dependan de él. Por supuesto, es necesario indicarle a make la dependencia de programas, lo cual se realiza en el archivo *Makefile*.

Un archivo Makefile es un archivo de texto en el cual se distinguen cuatro tipos básicos de declaraciones:

- Comentarios.
- Variables.
- Reglas explícitas.
- Reglas implícitas.

Comentarios

Al igual que en los programas, contribuyen a un mejor entendimiento de las reglas definidas en el archivo. Los comentarios se inician con el carácter #, y se ignora todo lo que continúe después de ella, hasta el final de línea.

```
# Este es un comentario
```

Variables

Se definen utilizando el siguiente formato:
nombre = dato

De esta forma, se simplifica el uso de los archivos Makefile. Para obtener el valor se emplea la variable encerrada entre paréntesis y con el carácter *$* al inicio, en este caso todas las instancias de *$(nombre)* serán reemplazadas por *dato*. Por ejemplo, la siguiente definición.

```
SRC = main.c
```

origina la siguiente línea:

```
gcc $(SRC)
```

y será interpretada como:

```
gcc main.c
```

Sin embargo, pueden contener más de un elemento *dato*. Por ejemplo:

```
objects = programa_1.o programa_2.o programa_3.o programa_4.o programa_5.o

programa: $(objects)
        gcc –o programa $(objects)
```

Hay que notar que make hace distinción entre mayúsculas y minúsculas.

Reglas explícitas

Estas le indican a make qué archivos dependen de otros archivos, así como los comandos requeridos para compilar un archivo en particular. Su formato es:

```
archivoDestino: archivosOrigen
        comandos  # Existe un carácter TAB (tabulador) antes de cada comando.
```

Esta regla indica que, para crear *archivoDestino*, make debe ejecutar *comandos* sobre los archivos *archivosOrigen.* Por ejemplo:

```
main: main.c funciones.h
        gcc -o main main.c funciones.h
```

significa que, para crear el archivo de destino *main*, deben existir los archivos *main.c* y *funciones.h* y que, para crearlo, debe ejecutar el comando:

```
gcc -o main main.c funciones.h
```

Reglas implícitas

Son similares a las reglas explícitas, pero no indican los comandos a ejecutar, sino que make utiliza los sufijos (extensiones de los archivos) para determinar que comandos ejecutar. Por ejemplo:

```
funciones.o: funciones.c funciones.h
```

origina la siguiente línea:

```
$(CC) $(CFLAGS) -c funciones.c funciones.h
```

Existe un conjunto de variables que se emplean para las reglas implícitas, y existen dos categorías: aquellas que son nombres de programas (como CC) y aquellas que tienen los argumentos para los programas (como CFLAGS). Estas variables son provistas y contienen valores predeterminados, sin embargo, pueden ser modificadas, como se muestra a continuación:

```
CC = gcc

CFLAGS = -Wall -O2
```

En el primer caso, se ha indicado que el compilador que se empleará es gcc y sus parámetros son *-Wall -O2.*

El ejemplo que ilustra este apartado se puede utilizar también en los ejercicios relacionados con compilación de proyectos en estas prácticas. Este ejemplo usa varios archivos: *complejo.c*: módulo que implementa los cálculos sobre números complejos; c*omplejo.h:* interfaz de prototipos del módulo *complejo.c*; *dibujoc.c:* módulo de representación gráfica de los números complejos; *dibujo.h:* interfaz de los prototipos del módulo *dibujo.c; main.c:* archivo principal. El archivo *Makefile* asociado es entonces el siguiente:

```
# Ejemplo de archivo makefile para números complejos
CC=gcc
RM=/bin/rm
# Opciones de compilación
CFLAGS:-g
# Archivos
SRC=complejo.c dibujo.c main.c
OBJ=$(SRC: .c=.o)
PROGRAMA=complejo
LIB=-lm

# Reglas de generación
$(PROGRAMA) : $(OBJ)
$(CC) $(CFLAGS) $(OBJ) -o $(PROGRAMA) $(LIB)

# Propiedad
clean:
$(RM) -f $(OBJ) $(PROGRAMA)

# Dependencias
complejo.o : complejo.c complejo.h
dibujo.o : dibujo.c dibujo.h complejo.h
main.c : main.c dibujo.h complejo.h
```

La primera parte de este archivo está constituida por declaraciones diversas que permiten una cierta flexibilidad de uso en el caso en que se quiera, por ejemplo, cambiar de compilador. Seguidamente, se definen las opciones que deberán utilizarse en la compilación. Aquí se trata de la opción *-g* para poder depurar eventuales errores.

Viene a continuación la enumeración de los distintos archivos que deben compilarse. La variable OBJ está formada por una regla que evita tener que enumerar todos los archivos objeto sabiendo que se trata de los mismos nombres que para los archivos fuente, aparte de la extensión.

La última parte es la regla de creación del ejecutable. Esta regla se ejecuta únicamente si todos los archivos objeto han sido generados. Al fin del archivo se encuentran las dependencias que permitirán a *make* compilar sólo los archivos modificados.

Por ejemplo, supongamos que se hayan generado todos los archivos objeto, pero desde la última compilación un desarrollador ha modificado el archivo *dibujo.h*. En la próxima compilación, sólo se recompilarán ciertos archivos:

```
# make
gcc -g -c dibujo.c -o dibujo.o
gcc -g -c main.c -o main.o
gcc -g complejo.o dibujo.o main.o -o complejo –lm
```

Gracias a la definición de dependencias, el archivo *complejo.c* no se recompila. Para conocer las dependencias de un programa, basta con lanzar la orden *gcc -MM dibujo.c*. Se obtiene así la lista de archivos de cabecera utilizados por este módulo.

La herramienta Makefile permite también el uso de comodines. Por ejemplo, se muestran a continuación los más comunes

- $(VAR) hace referencia al contenido de la variable VAR
- $@ hace referencia al nombre del objetivo
- $< hace referencia a la primera dependencia (después de los dos puntos)
- $^ hace referencia a la lista completa de dependencias

- $*. En la definición de una regla implícita, tiene el valor correspondiente al texto que reemplaza el símbolo %.
- $?. El nombre de todos los prerrequisitos.

Compilación nativa y cruzada

En el proceso de desarrollo de un sistema GNU/Linux embebido se debe realizar en un primer paso las configuraciones correspondientes de los componentes que formarán parte del sistema embebido (kernel, los módulos del mismo, el sistema de archivos raíz, etc), para a continuación realizar programas que vayan a ejecutar sobre dicho sistema.

Cuando se realiza el desarrollo de una aplicación para un sistema no embebido, usualmente lo compilamos en una computadora tipo PC, ya que dicha aplicación será ejecutada por un sistema de características de hardware similares. Otra práctica habitual ocurre cuando deseamos añadir soporte extra al kernel Linux, ya sea para hardware o para protocolos de red, en este caso realizamos la compilación en la misma computadora donde luego será ejecutado. Este tipo de compilación, en donde el software es compilado y ejecutado en una misma arquitectura de hardware se denomina **compilación nativa**.

Esta alternativa es la más utilizada, y se la denomina de este modo aún si la aplicación ha sido compilada en un equipo diferente al que luego la ejecutará siempre y cuando las arquitecturas de hardware sean compatibles. Un ejemplo de esto son las aplicaciones compatibles para microprocesadores AMD e Intel.

En el caso de los equipos que frecuentemente se utilizan para sistemas embebidos, poseen recursos de hardware limitados y específicamente diseñados para realizar cierto tipo de tareas. Por este motivo, en general, no es una buena alternativa realizar el desarrollo de las aplicaciones directamente en el sistema en el que se acabará ejecutando la aplicación, ya que sus recursos son limitados.

En este punto es donde surge la necesidad de realizar la compilación del sistema embebido en una computadora cuyos recursos no sean escasos. Es común hoy en día tener un PC con capacidades de procesamiento de 3 GHz y 1GB de memoria RAM, lo que nos supone un gran ahorro de tiempo al momento de realizar pruebas y cambios de configuraciones, solo basta diseñar un "ambiente" que le permita al compilador producir código ejecutable para un microprocesador diferente al que se está utilizando.

La compilación de software que será ejecutado en una arquitectura diferente, por ende, incompatible, a la que está produciendo el código ejecutable se denomina **compilación cruzada** y este "ambiente" que permite realizar la misma se denomina **entorno de compilación cruzada** o **entorno de desarrollo cruzado**.

En este entorno se identifica al equipo que realiza el desarrollo y la compilación mediante el término **Host** o **Huesped,** y al dispositivo donde se ejecutará el software desarrollado, como sistema **Target** u **Objetivo**.

Entorno de compilación cruzada

En nuestro caso, nuestro sistema Huesped será un ordenador de los que consta el laboratorio de la asignatura. Este sistema tiene unas características medias (procesador, memoria RAM, disco duro, conexión a internet, …) y una serie de aplicaciones instaladas (editores, sistema de escritorio, …) que le permiten trabajar de forma cómoda con él

Para implementar un entorno de compilación cruzada en él es necesario un conjunto de librerías, herramientas y binarios. En la bibliografía relacionada con sistemas GNU/Linux embebido a este conjunto de componentes se los denomina toolchain components. Estos componentes son:

- Compilador C : compilador de C básico, generador de código objeto (tanto del kernel como de aplicaciones)
- Librería C: implementa las llamadas al sistema mediante APIs.
- Binutils: conjunto de programas necesarios para la compilación, enlazado, ensamblado y depuración de código. Entre otros, los binarios principales son: ld (GNU linker), as (GNU assembler).

Nuestro sistema objetivo es la plataforma Raspberry Pi. Como conocemos, esta plataforma utiliza un chip Broadcom BCM2835, el cual contiene un procesador (CPU) ARM1176JZF-S a 700 MHz (aunque el firmware incluye unos modos "Turbo" para que el usuario pueda hacer overclock hasta 1 GHz sin perder la garantía), un procesador gráfico (GPU) VideoCore IV, y 512 MB de memoria RAM. Queda patente que los recursos de este sistema son con diferencia mucho menores que los del sistema huésped.

Comunicación ente el sistema huésped y el sistema objetivo

Para realizar la comunicación entre el sistema huésped y el sistema objetivo se empleará una herramienta, que es el programa ssh. Para tal fin, en la instalación del Sistema Operativo Raspian se habilita en Raspberry Pi un servidor SSH. *SSH* es un programa que permite acceder a otro ordenador a través de la red, ejecutar comandos en la máquina remota y mover ficheros entre dos máquinas. Provee autenticación y comunicaciones seguras sobre canales inseguros.

El comando **ssh** ofrece comunicación encriptada y segura entre dos sistemas sobre una red no segura. Para iniciar una sesión en otra máquina usando **ssh**:

```
$ ssh usuario1@servidor.dominio.es
The authenticity of host 'servidor.dominio.es (192.168.0.2)' can't be
established.
RSA key fingerprint is 97:4f:66:f5:96:ba:6d:b2:ef:65:35:45:18:0d:cc:29.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'servidor.dominio.es' (RSA) to the list of known
hosts.
usuario1@servidor.dominio.es's password:
$
```

Nota: La primera vez que realizas la conexión debes aceptar la firma del otro host. De esta manera se establece una relación de confianza que se traduce en archivar la clave pública de este servidor en el fichero $HOME/.ssh/known_hosts.

La sintaxis básica del comando **ssh** es:

```
ssh user@hostname [command]
```

El comando es opcional. Si se especifica en lugar de obtener un shell se ejecuta el comando en la máquina remota. Por ejemplo, podríamos hacer un ls en la máquina remota y observar su salida:

```
ssh usuario1@servidor.dominio.es ls
```

O realizar alguna operación más elaborada como realizar una copia en local de un directorio remoto, como en el ejemplo:

```
ssh usuario1@servidor.dominio.es "tar cf - /home/usuario1" | tar xvf -
```

Una de las funcionalidades que le da mayor potencia al **ssh** es la redirección de las X.

Para realizar la transferencia de ficheros entre una maquina remota y una local se usa el comando scp. Este comando Utiliza **ssh** para la transmisión de la información, por lo que ofrece la misma seguridad

que el **ssh**. De la misma manera utiliza los métodos de autenticación de **ssh**. Este es un ejemplo de uso del **scp** para copiar desde la máquina local a una remota:

```
$ scp /tmp/file usuario1@servidor.dominio.es:/tmp
```

También podemos copiar ficheros entre dos máquinas remotas:

```
$ scp usuario1@anotherhost:/tmp/file usuario1@servidor.dominio.es:/tmp
```

La sintaxis del comando es:

```
scp [-pqrvBC46] [-F ssh_config] [-S program] [-P port] [-c cipher]
         [-i identity_file] [-o ssh_option] [[user@]host1:]file1 [...]
         [[user@]host2:]file2
```

Puedes consultar las opciones en la página man de **scp**, estas son las más habituales:

- -p: conserva las propiedades del archivo. Permisos del archivo, fecha de última de modificación.
- -r: copia recursiva de directorios

La sintaxis para especificar el origen o destino de los archivos tiene la forma **[[user@]host:]file** donde:

- **user**: es el usuario de la máquina. Si no se especifica es el actual.
- **host**: es la máquina origen o destino del archivo. Si no se informa es la máquina local.
- **file**: fichero o directorio a copiar. Por defecto es el directorio HOME del usuario. En caso de ser un directorio deberás especificar la opción -r.

  En nuestro caso, en la instalación de raspbian sobre raspberry pi, se crea un usuario por defecto que es el usuario *pi*, con contraseña *raspberry*. Al habilitarse el servicio ssh, este usuario puede ser el que se utilice para la conexión con ssh.

Instalación del toolchain

A continuación, vamos a proceder a la instalación del compilador cruzado en el sistema huésped que usaremos para construirnos el kernel para el sistema objetivo, nuestra Raspberry Pi. Para eso, descargaremos dicho compilador del servidor Github mediante el siguiente comando:

```
$ git clone https://github.com/raspberrypi/tools
```

Este compilador y sus herramientas serán descargadas y desempaquetadas en el subdirectorio tools y ocuparán alrededor de 1Gbyte. Dentro de ese directorio existen varias versiones de la herramienta de compilador cruzado.

- tools/arm-bcm2708/arm-bcm2708hardfp-linux-gnueabi
- tools/arm-bcm2708/arm-bcm2708-linux-gnueabi
- tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabif-raspbian

La ultima contiene unos patches (los patches linaro) por lo que elegiremos esta versión para la compilación cruzada. Para tal fin, fijaremos la variable de entorno CCPREFIX al directorio que contiene la tercera versión y, a continuación, testearemos su correcto funcionamiento invocando el comando de compilación del este compilador cruzado (gcc) utilizando dicho prefijo:

```
$ export CCPREFIX=~/raspberry/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabif-raspbian/bin/arm-linux-gnueabihf-
$ ${CCPREFIX}gcc –v
```

Tras la ejecución de este último comando, en la última línea debería aparecer un mensaje con el número de versión de la herramienta que se encuentra instalada.

Practica planteada

1) Realícese la instalación del sistema operativo Raspbian en la tarjeta Raspberry Pi del laboratorio.
2) Realizar un programa llamado `Suma` que realice la suma de dos numero recibidos por el teclado y saque el resultado por la pantalla. Realícese la compilación nativa de dicho programa (el ejecutable se ejecutará en el mismo sistema donde se ha compilado, sistema PC del laboratorio), y compruebe el correcto funcionamiento. Realizar un Makefile para la compilación del código.
3) Instálese el entorno de compilación cruzada en el PC del laboratorio. Realícese la compilación del programa `Suma`, realizado en el punto anterior, utilizando la herramienta gcc (que produce la compilación para el sistema PC), y también la compilación con ${PREFIXCC}gcc, que realiza la compilación cruzada (para el sistema RaspberryPi). Realícense sendos Makefile para automatizar la compilación. Cópiense los resultados de ambas compilaciones a la maquina destino (RaspberryPi), y procédase a la ejecución de ambas. Coméntese los resultados obtenidos y su justificación.