Introducción a la compilación en ambiente Unix

Luis C. Pérez Tato

27 de julio de 2012

Índice

1. Introducción **1**
1.1. ¿Porqué tengo que compilar el programa? 1

2. Compilación **2**
2.1. Estructura del código fuente . . . 2
2.2. Bibliotecas de funciones 3
2.2.1. Ejemplo de función de biblioteca 4
2.3. Problemas que pueden presentarse al compilar el programa . . 4
2.3.1. Archivos de cabecera . . . 4
2.3.2. No se encuentra un archivo de cabecera 5

3. Enlace **6**
3.1. Problemas que pueden presentarse al enlazar el programa . . . 6
3.1.1. Archivos de biblioteca . . 6
3.1.2. No se encuentra la definición de una función . . . 6
3.1.3. No se encuentra un archivo de biblioteca 6

4. La herramienta «Make» **7**

5. Sistemas de compilación integrados (CMake,. . .) **7**
5.1. Empleo de CMake 7
5.2. Autoconf 7

6. Las instrucciones de compilación **7**

7. Conclusión **7**

1. Introducción

Este artículo se redacta pensando en aquellas personas que tienen conocimientos de informática a nivel de usuario y desean aprender a compilar e instalar programas que se distribuyen en forma de código fuente. En particular se pretende facilitar la instalación del programa Xc[1].

Con ese fin se introducen algunos conceptos básicos acerca de la generación de código ejecutable para, posteriormente, explicar cómo se resuelven los problemas más frecuentes a la hora de compilar un programa en un ambiente ligeramente diferente al de la máquina en que se creó.

Aunque frecuentemente se emplea la palabra *compilar* para referirse a la obtención de un programa ejecutable a partir del código fuente del mismo, en realidad dicho proceso requiere realizar dos operaciones: *compilación* y *enlace*. Más adelante se explica brevemente en qué consisten estas operaciones.

En las páginas siguientes se intentará transmitir al lector los conocimientos básicos necesarios que le permitan iniciarse en el uso de programas distribuidos en forma de código fuente.

1.1. ¿Porqué tengo que compilar el programa?

Puede que uno se pregunte acerca de la necesidad de compilar el programa en lugar de emplear un cómodo programa de instalación que oculte al usuario la complejidad del proceso de generar código ejecutable para una máquina determinada.

Cuando un programa está diseñado para ejecutarse en un sólo sistema operativo que co-

[1] Xc es un programa de análisis de estructuras mediante elementos finitos de código abierto

rre sobre un sólo tipo de máquina (por ejemplo OS X y Mac) es relativamente sencillo generar un ejecutable válido para dichas máquinas y un programa de instalación que se encargue de colocar el archivo ejecutable y otros archivos que sean necesarios para la ejecución, en los lugares adecuados.

Sin embargo, si lo que se pretende es que el programa se pueda instalar en una familia de máquinas tan amplia como sea posible la solución anterior no resulta tan sencilla de aplicar. El problema radica en la variedad de arquitecturas para las que hay que resolver la instalación. Por ejemplo considerando sólo las arquitecturas de hardware más populares tendremos:

- Intel IA-32.
- Intel IA-64.
- AMD64.

Por otra parte sobre esa arquitectura podríamos encontrar distintas distribuciones de Linux (Ubuntu, Linux Mint, Debian, Red Hat, Suse, Fedora,...) u otros sistemas de similares características HP-UX, IBM-AIX, Solaris, etcétera.

El coste de elaborar soluciones para la instalación, aunque sólo fuera para unas pocas de entre las combinaciones posibles, es muy grande[2].

Si en lugar de ello lo que se hace es distribuir el código fuente y dejar al usuario la tarea de compilarlo el problema se simplifica bastante y la tarea que se traslada al usuario no es demasiado complicada en la mayor parte de los casos[3]

Como puede ver el argumento es muy similar a los que emplea una conocida cadena de distribución de muebles para justificar que sea el cliente el que tenga que transportarlos hasta su casa y montarlos.

[2]Las distintas distribuciones de Linux resuelven este problema empleando a personas (los denominados «package maintainers») que se dedican a adaptar un programa determinado (como por ejemplo Gimp `www.gimp.org`) a la distribución en cuestión.

[3]La tarea será tanto más complicada cuanto más diferente sea la máquina en la que se pretende compilar el programa de aquella en la que se creó el mismo. Por ejemplo si se intenta compilar un programa diseñado para Unix en una máquina Windows el dolor de cabeza está prácticamente garantizado.

2. Compilación

Compilar, en nuestro contexto, consiste en «traducir» el código fuente escrito en un lenguaje de alto nivel (C, C++, Fortran,...) a una secuencia de instrucciones directamente interpretable por el procesador (Intel, Sparc, Motorola,...) (*lenguaje de máquina*).

Para hacerse una idea de qué aspecto tiene el lenguaje de máquina puede examinarse la figura 1.

2.1. Estructura del código fuente

El código fuente de un programa estará formado por el del *programa principal* y las bibliotecas de funciones que acompañan a éste. Para dar una idea de en qué consiste esto en los párrafos siguientes se desarrolla un ejemplo que, aun siendo trivial, contiene los elementos básicos que forman el problema. Por otra parte en la figura 2 se intentan representar las relaciones de dependencia del programa con el sistema operativo y el hardware en el que se pretende ejecutar.

Supongamos que tenemos un archivo de nombre `hola.cc` con el siguiente contenido:

```
#include <iostream>
int main()
  {
    std::cout << "Hola a todos."
              << std::endl;
    return 0;
  }
```

como el nombre del archivo lleva la extensión «cc» suponemos que se trata de un programa escrito en C++ (si estuviera escrito en FORTRAN debería llevar la extensión «for», en Python la extensión «py»,...). Por otra parte como en el programa se define la función «main» sabemos que el archivo corresponde al *programa principal* ya que en C++ (y en C) la función que se ejecuta al lanzar la ejecución del programa es precisamente ésta.

Si examinamos la primera línea del código en la que se lee `#include <iostream>` deducimos que el programa emplea el archivo de cabecera denominado «iostream» y, con un poco de esfuerzo se puede averiguar (aunque en general no será necesario) que en un sistema Debian Linux versión 6 esto implica usar la biblioteca de

funciones «LibCXX 3.4» que forma parte del sistema operativo.

Así que el programa `hola.cc` está formado por un único archivo y (en mi sistema) sólo usa la biblioteca de funciones LibCXX versión 3.4. Si el lector usa alguna variante de Unix y quiere compilar este programa, con el compilador de GNU, puede hacerlo mediante:

```
g++ hola.cc -o hola
```

Para ilustrar el caso general, en el que el programa a compilar emplea bibliotecas de funciones cuyo código fuente se suministra con el mismo, escribiremos el archivo `bibHola.h` que contendrá lo siguiente:

```
//bibHola.h

#ifndef BIBHOLA_H
#define BIBHOLA_H

#include <string>

void print_msg(const std::string &msg);

#endif
```

Este será el *archivo de cabecera* de nuestra biblioteca en el que se declara una sola función a la que hemos llamado `pring_msg`.

La implementación de la función la escribiremos en un archivo al que llamaremos `bibHola.cc` y aquí definiremos lo que hace la función:

```
//bibHola.cc

#include "bibHola.h"
#include <iostream>

void print_msg(const std::string &msg)
  {
    std::cout << msg << std::endl;
  }
```

Por último modificaremos el contenido del archivo `hola.cc` como sigue:

```
#include "bibHola.h"
int main(void)
  {
    print_msg("Hola a todos.");
    return 0;
  }
```

Ahora el programa principal usa la función `pring_msg` de nuestra raquítica biblioteca de funciones «bibHola».

Ahora el proceso para generar el ejecutable un poco más complicado. Primero deberemos compilar la biblioteca de funciones. Para ello tecleáríamos:

```
g++ -c bibHola.cc -o bibHola.o
```

y a continuación generaremos el ejecutable mediante:

```
g++ bibHola.o hola.cc -o hola
```

El ejemplo descrito hasta aquí, a pesar de su simplicidad, ilustra las principales tareas que deben realizarse para compilar cualquier programa. En los apartados siguientes se desarrollan un poco más algunos aspectos de lo tratado aquí de forma tan superficial.

2.2. Bibliotecas de funciones

Un microprocesador es capaz, por sí solo, de ejecutar instrucciones sencillas como la suma, resta y producto de números. También existen microprocesadores que incorporan rutinas para procesar operaciones no tan sencillas como evaluar funciones trigonométricas[4], logaritmos, etcétera.

El resto de las operaciones que realiza un programa (mostrar texto y figuras en un monitor, leer y escribir en discos duros, memorias USB y periféricos de todo tipo,...) se realizan con el auxilio de lo que llamamos *funciones de biblioteca*. Ese es el motivo de que cuando se adquiere un elemento de hardware nuevo (tarjeta de vídeo, escáner,...) venga acompañado de un disco con sus «drivers», es decir, con las funciones de biblioteca que permitirán al sistema operativo de la máquina y a otras aplicaciones manejar el dispositivo (mostrar gráficos 3D, escanear documentos,...).

Estas funciones de biblioteca no son más que secuencias de instrucciones básicas que sirven para facilitar la ejecución de una operación relativamente compleja (por ejemplo calcular el determinante de una matriz), o acceder a determinada característica de determinado hardware (por ejemplo borrar lo que se muestra en pantalla).

[4] A los usuarios más veteranos de ordenadores de tipo PC les resultará familiar el término «coprocesador matemático» que popularizó Intel en los años 80.

2.2.1. Ejemplo de función de biblioteca

Para dar una idea un poco más aproximada de lo que se intenta describir en este apartado pondremos un ejemplo trivial.

Supongamos una función que se encarga de cambiar el color de un píxel en la pantalla. Esta función podría recibir como argumentos las coordenadas x e y del píxel en cuestión y los valores RGB[5]. Su declaración en C++ sería semejante a esta:

```
int color_pixel(int x,int y,float r
, float g, float b);
```

Cuando el compilador se encuentra con una llamada a esta función como, por ejemplo, la siguiente:

```
ok= color_pixel(400,300,
0.5,0.5,0.5); //gris.
```

generará las instrucciones del microprocesador que sirvan para:

1. Colocar en la pila[6] los cinco valores que empleará la función, en este caso: 400, 300, 0.5, 0.5, 0.5.

2. Colocar a continuación la llamada a la función. Esto hará que, cuando se ejecute el programa, el código de dicha función recoja los cinco argumentos de la pila y ejecute, empleando esos valores, las instrucciones que sean necesarias (llamadas a la tarjeta gráfica,...).

3. Generar el código para, en su caso, almacenar el valor devuelto por la función en alguna variable cuyo nombre le habremos indicado. En este caso: `ok`.

Una vez que termina el proceso de compilación tendremos, por cada uno de los archivos fuente que hayamos compilado, un *archivo objeto* que contendrá las instrucciones del programa en código máquina con llamadas a funciones de biblioteca intercaladas. Estos archivos suelen tener la extensión `.o` en sistemas Unix y Mac OS, y `.obj` en los sistemas operativos de Microsoft[7].

2.3. Problemas que pueden presentarse al compilar el programa

En lo que sigue se supone que tratamos de compilar un programa que se creó y probó con éxito en otra máquina y por tanto está libre de errores sintácticos. Dicho de otro modo, los problemas de los que se hablará aquí son los debidos a las diferencias que pueden existir entre la máquina en la que pretendemos compilar el programa y aquella en la que se desarrolló y probó el mismo.

2.3.1. Archivos de cabecera

Para que el compilador pueda generar el código que servirá para llamar a las funciones de biblioteca necesita conocer:

- El nombre de la función a llamar.
- El número de argumentos que recibirá la función y su orden.
- El tipo de cada uno de los argumentos (entero, coma flotante, cadena de caracteres,...).
- El tipo de valor que, en su caso, devuelve la función.

Para verificar que la llamada a la función puede hacerse correctamente el compilador necesita acceder a la declaración de la función (en la que se definen los datos de la lista anterior). En C y C++ esto se resuelve empleando lo que se llama *archivos de cabecera*. Estos archivos contienen la declaración de las funciones y variables de la biblioteca de funciones a la que corresponden.

Por ejemplo en el cuadro 1 puede verse la declaración de la función `getMemoryUsed` que no recibe argumentos y devuelve un entero largo sin signo. Una llamada a esta función dentro del código fuente podría ser:

[5] Estos valores son sencillamente las intensidades de color rojo (red: R), verde (green: G) y azul (blue: B) que definen el color a mostrar en el píxel.

[6] Los que sean usuarios de calculadoras HP con notación polaca inversa (RPN) ya saben lo que es una pila, los que no tengan esa suerte pueden consultar la referencia [2].

[7] Como se ha dicho anteriormente, compilar bajo Windows un programa desarrollado en Unix es una tarea muy difícil salvo que se trate de un programa extremadamente simple o el desarrollador previera dicha posibilidad. En este documento se citan distintos sistemas operativos para que sea lo más general posible.

```
//memoria.h

#ifndef MEMORIA_H
#define MEMORIA_H

unsigned long getMemoryUsed(void);

#endif
```

Cuadro 1: Ejemplo de archivo de cabecera (muy pequeño).

```
unsigned long memOcupada= 0;
memOcupada= getMemoryUsed();
```

Con ambas informaciones (la declaración y la llamada) el compilador puede establecer la correspondencia entre los argumentos en la llamada y en la declaración. También puede establecer la correspondencia entre el valor devuelto por la función y la variable a la que se asigna (`getMemoryUsed`).

Para poder hacer todo esto el compilador necesita tener acceso al archivo de cabecera, es decir conocer su ubicación en el sistema de archivos (por ejemplo `/usr/include/math.h`). Estos archivos se colocan en distintos directorios dependiendo del sistema operativo que estemos usando y de la forma en que se haya instalado la biblioteca cuya interfaz[8] definen (paquete binario, compilada por el usuario,...).

Aunque, como veremos más adelante, existen herramientas para facilitar la solución de este problema, para que el compilador pueda acceder a determinado archivo de cabecera es necesario asegurarse de que:

1. La versión *de desarrollo*[9] de la biblioteca está instalada en el sistema.

2. El compilador conoce la ubicación del archivo de cabecera.

Veamos esto con un ejemplo. Supongamos que queremos compilar un programa que hace uso de la biblioteca OpenGL y para ello llama a funciones declaradas en el archivo (`gl.h`) de dicha biblioteca. Una vez instalada la biblioteca OpenGL, este archivo puede encontrarse, dependiendo del sistema que usemos, en las ubicaciones que se dan en la tabla 3.

El procedimiento que se emplea para indicar al compilador la ubicación de los archivos de cabecera varía de unos sistemas a otros. Por ejemplo si estamos empleando el compilador GNU C++ se emplea la opción `-I` seguida del directorio en el que se encuentran los archivos de cabecera. En el caso de que estuviéramos empleando este compilador en una máquina HP-UX usaríamos:

```
g++ -I/opt/graphics/OpenGL/include
   programa.cc
```

Si además necesitáramos emplear otro archivo de cabecera que estuviera en el directorio `/usr/local/include`.

```
g++ -I/opt/graphics/OpenGL/include
  -I/usr/local/include programa.cc
```

2.3.2. No se encuentra un archivo de cabecera

De lo anterior se deduce que cuando el compilador emite un mensaje de error como:

```
error: blas.h: No existe el
 fichero o el directorio
```

puede deberse a dos razones:

1. Realmente el archivo `blas.h` no está instalado en el sistema. Para cerciorarnos de ello en los sistemas Unix podemos usar el comando:

   ```
   locate blas.h
   ```

   Si efectivamente el archivo no está instalado probablemente se deba a que no se ha instalado la biblioteca a la que pertenece el archivo. En sistemas Debian esto puede averiguarse empleando el comando:

   ```
   apt-file search blas.h
   ```

   Si en su sistema no dispone de una utilidad similar puede utilizar búsquedas de Google para inferir a qué biblioteca pertenece el archivo.

[8] El archivo de cabecera al definir el modo en que se puede llamar a las distintas rutinas de la biblioteca a la que pertenece constituye su *interfaz*.

[9] La que proporciona los archivos de cabecera de la biblioteca.

2. El archivo sí está instalado pero el compilador no puede localizarlo. En este caso el problema es que falta la opción `-I` que indica al compilador la ubicación del archivo. Si se tiene acceso al comando que lanza la ejecución del compilador bastará con añadir esta opción. Por el contrario si se emplean medios más sofisticados para compilar el programa como make, CMake o automake y no conocemos su funcionamiento será necesario contactar con el desarrollador para indicarle el problema.

3. Enlace

Una vez que disponemos del código objeto del programa, para generar el código ejecutable, se deberá colocar en el sitio adecuado el código ejecutable de las funciones de biblioteca a las que llamamos. A esta operación se la llama *enlazar*. Cuando el código de las funciones de biblioteca se incorpora al programa ejecutable decimos que en enlace es *estático*. Por el contrario cuando lo que se coloca en el programa ejecutable es el código necesario para ejecutar el código de la función que residirá en una biblioteca de enlace *dinámico*.

De un modo u otro, cuando el *linker* termina su tarea obtendremos un archivo que contendrá un programa ejecutable. Estos archivos no suelen tener extensión en los sistemas Unix y tienen la extensión `.EXE` en los sistemas operativos de Microsoft.

3.1. Problemas que pueden presentarse al enlazar el programa

3.1.1. Archivos de biblioteca

Los archivos de biblioteca son aquellos en los que se almacena el código ejecutable de las funciones que componen la misma. En los sistemas Linux tienen las extensiones `.o` ó `.a` cuando se trata de bibliotecas estáticas y la extensión `.so` (normalmente acompañada de un número de versión) cuando es una biblioteca de enlace dinámico. En los sistemas Windows las bibliotecas de enlace dinámico tienen la extensión `.dll` y las de enlace estático la extensión `.lib`.

3.1.2. No se encuentra la definición de una función

Aunque el programa enlazara correctamente en el sistema que se creó es posible que en nuestro sistema la definición de alguna función esté en un archivo de biblioteca distinto. En este caso el enlazador emitirá un mensaje parecido al el que sigue (el texto importante es el que reza «undefined reference to»):

```
blas_example.c:(.text+0xaf): undefined
 reference to ‘ddot_’
```

En general para localizar el archivo de biblioteca en el que se define una función comenzaremos por localizar el archivo de cabecera en el que se declara la función. Localizado éste, es frecuente que (en los sistemas Unix) el archivo que contiene los binarios correspondientes tenga el mismo nombre precedido del prefijo `lib`. Por ejemplo las funciones declaradas en el archivo de cabecera `gmp.h` están definidas en el archivo `libgmp.a` ó `libgmp.so`. Cuando esto no sucede así, es necesario consultar la documentación de la implementación de la biblioteca en cuestión en nuestro sistema.

3.1.3. No se encuentra un archivo de biblioteca

Como en el caso de los archivos de cabecera cuando el enlazador no encuentra el archivo de biblioteca emite un mensaje de error como el siguiente:

```
/usr/bin/ld: cannot find -lblas
```

Con ello nos indica que no encuentra el archivo `libblas.a`. Igual que en el caso de los archivos de cabecera, el hecho de que el enlazador no encuentre el archivo puede deberse a que efectivamente no esté instalado (aunque en este caso es más difícil porque lo normal es que se haya instalado junto con el archivo de cabecera) o a que esté en un directorio que el enlazador no encuentra. En el primer caso la solución consistirá en averiguar si la instalación de la biblioteca es defectuosa.

En el segundo caso la solución consiste en indicar al enlazador el directorio en el que se encuentra el archivo mediante la opción `-L`. Por ejemplo si se encuentra en `/usr/local/lib` agregaríamos este directorio al comando que lanza la compilación del programa.

```
gcc -lblas -L/usr/local/lib
  blas_example.c
```

Como ocurría con los archivos de cabecera, cuando se emplean sistemas de compilación integrados como CMake o autoconf, el procedimiento a emplear es distinto.

4. La herramienta «Make»

La herramienta «make» es una utilidad de uso común, sobre todo en entornos Unix, que sirve para automatizar la generación de programas ejecutables y bibliotecas a partir del código fuente. A tal fin se escribe un archivo denominado «makefile» en el que se especifican las reglas a seguir para obtener el código fuente. Como ejemplo mostramos aquí el archivo `makefile` que se emplearía para automatizar las tareas que se describieron en el apartado 2.1:

```
all: hola

hola: hola.o bibHola
	g++ hola.cc bibHola.o -o hola
bibHola: bibHola.o
	g++ -c bibHola.cc
clean:
	rm -f hola hola.o bibHola.o
```

En la referencia [4] se puede encontrar más información sobre esta herramienta.

5. Sistemas de compilación integrados (CMake,...)

Los sistemas de compilación integrados como «automake» o «CMake» suponen un paso más (respecto a make) en la automatización de la generación del programa. Su finalidad principal evitar los problemas descritos acerca de la ubicación de los archivos de cabecera y de biblioteca. Para ello están dotados de una serie de rutinas mediante las que se determina la ubicación de estos archivos en el sistema y se generan los comandos de compilación adecuados para el programa en el sistema de que se trate.

La ventaja de estos programas es que permiten al desarrollador abstraerse de las particularidades de los distintos sistemas operativos. Su principal desventaja es que añaden una capa más al proceso oscureciéndolo un poco.

5.1. Empleo de CMake

Cuando se emplea este sistema para compilar el programa los requerimientos del mismo en lo que ser refiere a las bibliotecas que emplea y otros detalles se especifican en un archivo denominado `CMakeLists.txt`. Con el contenido de este archivo el programa `cmake` genera el archivo `makefile` (ver apartado 4) con todas las opciones `-I` y `-L` necesarias.

Naturalmente el `makefile` generado es tan bueno como lo sea la especificación que haga el desarrollador en `CMakeLists.txt`. El olvido de alguna biblioteca necesaria para compilar el programa dará lugar al correspondiente error. Llegados a este punto cabe modificar el archivo `CMakeLists.txt` para subsanar el error si se tienen conocimientos para hacerlo ó indicar al desarrollador el mensaje de error obtenido para que pueda corregirlo.

5.2. Autoconf

Como en el caso anterior este programa genera un archivo `makefile` a partir de una macro denominada `configure` que es la encargada de recopilar la información acerca de la ubicación de las bibliotecas y archivos de cabecera en el sistema.

En caso de error el procedimiento a seguir es el descrito en el apartado anterior.

6. Las instrucciones de compilación

Generalmente el código fuente de un programa viene acompañado de un archivo de nombre similar `install.txt` o `readme.txt` en el que se indican, más o menos minuciosamente, los pasos a seguir para compilar el programa.

Una vez conocida, siquiera de forma somera, la forma en que se compila un programa en ambiente Unix resultará más sencillo interpretar correctamente esas instrucciones y averiguar cómo corregir los fallos más simples cuando estos se produzcan.

7. Conclusión

Con la información que se proporciona en el presente artículo se pretende facilitar el uso

de software libre distribuido en forma de código fuente a personas cuyos conocimientos sobre programación son escasos.

Si además sirve para que alguna de esas personas se anime a colaborar en proyectos de software libre o a iniciar el suyo propio el objetivo estará sobradamente cumplido.

Si alguien desea hacer algún comentario o sugerencia puede escribirme a la dirección: l.perez@xcingenieria.com.

```
-u 100 la
0CFD:0100 BA0B01      MOV     DX,010B
0CFD:0103 B409        MOV     AH,09
0CFD:0105 CD21        INT     21
0CFD:0107 B400        MOV     AH,00
0CFD:0109 CD21        INT     21
-d 10b 13f
0CFD:0100                                48 6F 6C 61 2C        Hola,
0CFD:0110  20 65 73 74 65 20 65 73-20 75 6E 20 70 72 6F 67   este es un prog
0CFD:0120  72 61 6D 61 20 68 65 63-68 6F 20 65 6E 20 61 73   rama hecho en as
0CFD:0130  73 65 6D 62 6C 65 72 20-70 61 72 61 20 6C 61 20   sembler para la
0CFD:0140  57 69 6B 69 70 65 64 69-61 24                     Wikipedia$
```

Figura 1: Lenguaje de máquina del Intel 8088. El código de máquina en hexadecimal se resalta en rojo, el equivalente en lenguaje ensamblador en magenta, y las direcciones de memoria donde se encuentra el código, en azul. Abajo se ve un texto en hexadecimal y ASCII (tomada de la referencia [1]).

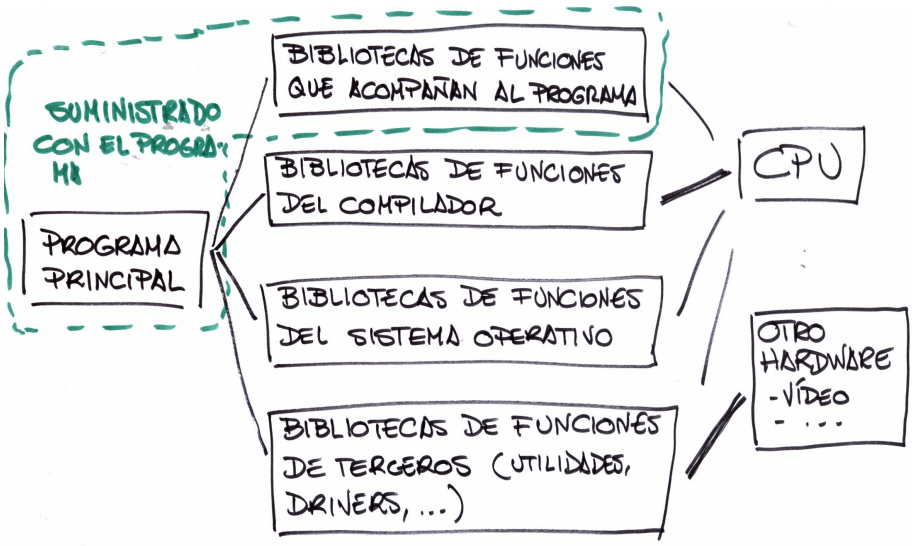


Figura 2: Dependencias del código de un programa de ordenador.

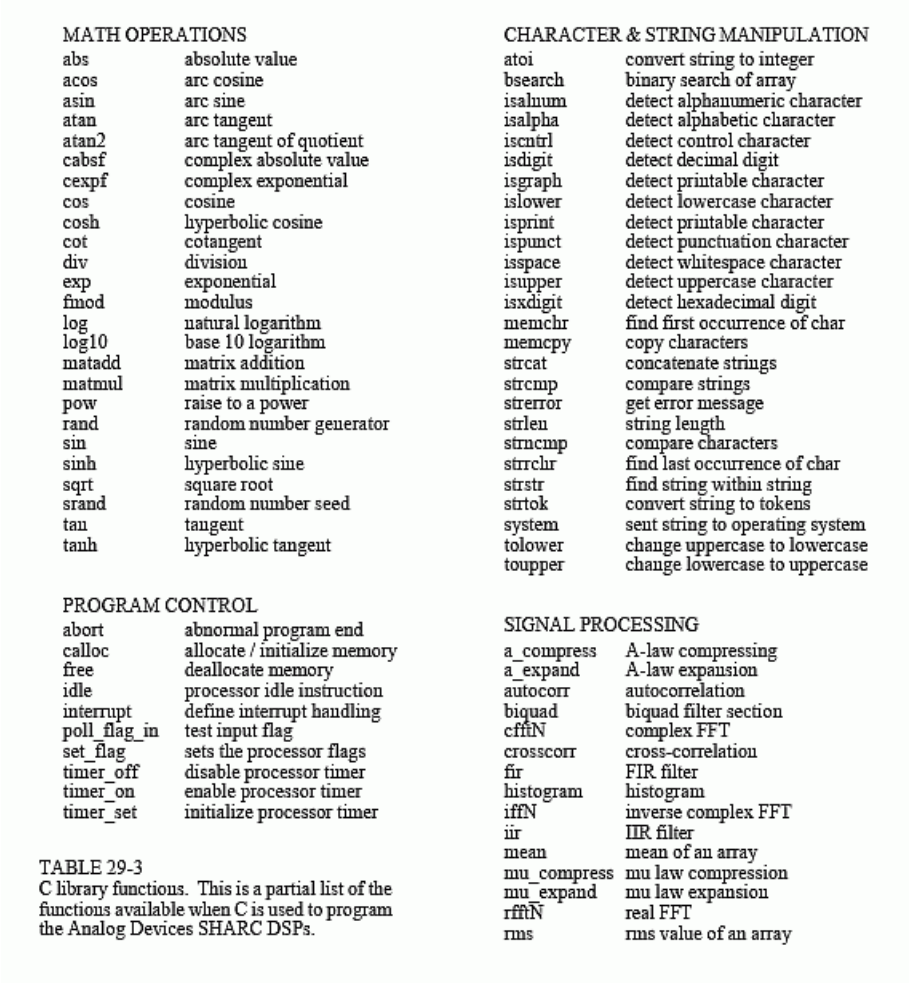
Referencias

[1] Colaboradores de Wikipedia. Compilador [en línea]. Wikipedia, La enciclopedia libre, 2012 [fecha de consulta: 3 de abril del 2012]. Disponible en <http://es.wikipedia.org/w/index.php?title=Compilador>.

[2] Colaboradores de Wikipedia. Pila (informática) [en línea]. Wikipedia, La enciclopedia libre, 2012 [fecha de consulta: 3 de abril del 2012]. Disponible en <http://es.wikipedia.org/w/index.php?title=Pila_(informática)>.

[3] Conceptos de Informática. P. Bishop. Editorial Anaya 1990.

[4] Colaboradores de Wikipedia. Make [en línea]. Wikipedia, La enciclopedia libre, 2012 [fecha de consulta: 4 de abril del 2012]. Disponible en <http://es.wikipedia.org/w/index.php?title=Make>.



MATH OPERATIONS

abs	absolute value
acos	arc cosine
asin	arc sine
atan	arc tangent
atan2	arc tangent of quotient
cabsf	complex absolute value
cexpf	complex exponential
cos	cosine
cosh	hyperbolic cosine
cot	cotangent
div	division
exp	exponential
fmod	modulus
log	natural logarithm
log10	base 10 logarithm
matadd	matrix addition
matmul	matrix multiplication
pow	raise to a power
rand	random number generator
sin	sine
sinh	hyperbolic sine
sqrt	square root
srand	random number seed
tan	tangent
tanh	hyperbolic tangent

PROGRAM CONTROL

abort	abnormal program end
calloc	allocate / initialize memory
free	deallocate memory
idle	processor idle instruction
interrupt	define interrupt handling
poll_flag_in	test input flag
set_flag	sets the processor flags
timer_off	disable processor timer
timer_on	enable processor timer
timer_set	initialize processor timer

TABLE 29-3
C library functions. This is a partial list of the functions available when C is used to program the Analog Devices SHARC DSPs.

CHARACTER & STRING MANIPULATION

atoi	convert string to integer
bsearch	binary search of array
isalnum	detect alphanumeric character
isalpha	detect alphabetic character
iscntrl	detect control character
isdigit	detect decimal digit
isgraph	detect printable character
islower	detect lowercase character
isprint	detect printable character
ispunct	detect punctuation character
isspace	detect whitespace character
isupper	detect uppercase character
isxdigit	detect hexadecimal digit
memchr	find first occurrence of char
memcpy	copy characters
strcat	concatenate strings
strcmp	compare strings
strerror	get error message
strlen	string length
strncmp	compare characters
strrchr	find last occurrence of char
strstr	find string within string
strtok	convert string to tokens
system	sent string to operating system
tolower	change uppercase to lowercase
toupper	change lowercase to uppercase

SIGNAL PROCESSING

a_compress	A-law compressing
a_expand	A-law expansion
autocorr	autocorrelation
biquad	biquad filter section
cfftN	complex FFT
crosscorr	cross-correlation
fir	FIR filter
histogram	histogram
iffN	inverse complex FFT
iir	IIR filter
mean	mean of an array
mu_compress	mu law compression
mu_expand	mu law expansion
rfftN	real FFT
rms	rms value of an array

Cuadro 2: Algunas funciones de la biblioteca estándar de C.

Sistema	**Ubicación**
Windows	C:\Program Files\Microsoft SDKs\Windows\v7.0A\Include\gl\gl.h
Debian Linux	/usr/include/GL/gl.h
Mac OS X	/usr/include/OpenGL/gl.h
Solaris	/usr/X11/include/GL/gl.h
HP-UX	/opt/graphics/OpenGL/include/GL/gl.h

Cuadro 3: Ubicaciones del archivo gl.h en distintos sistemas operativos.