Fundamentos de programación de Sistemas Embebidos
Entorno de desarrollo

Mg. Ing. E. Sergio Burgos

Universidad Nacional de Entre Ríos
Facultad de Ingeniería
Especialización en Sistemas Embebidos

31/08/2018

Desarrollo de aplicaciones

¿Qué necesitamos como mínimo para desarrollar una aplicación de escritorio en lenguaje C?

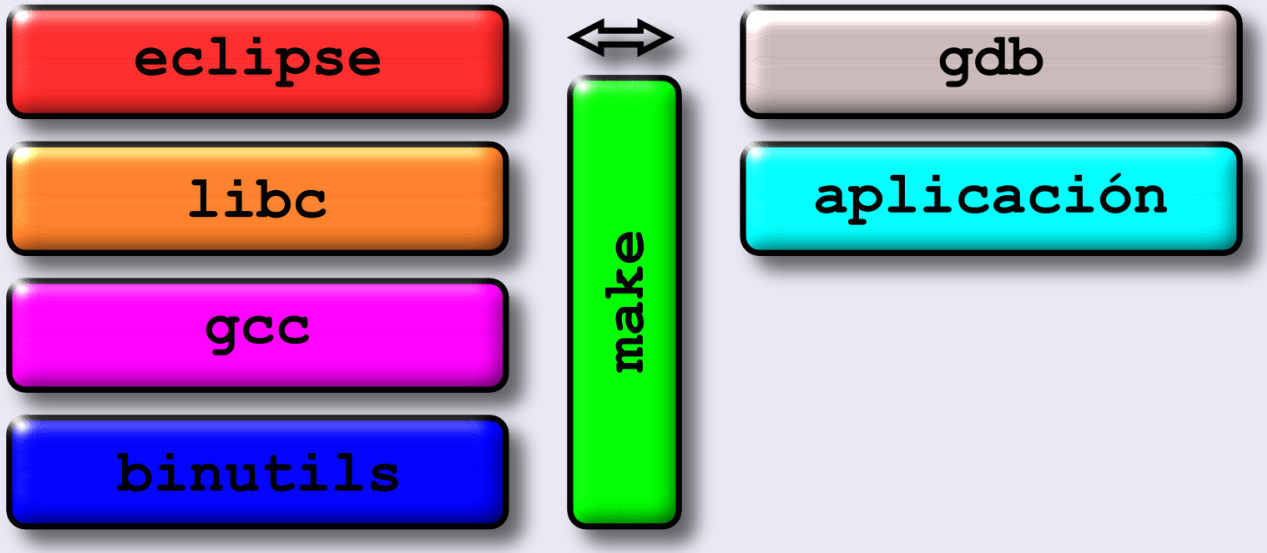
- Rta.: Un compilador

Surgen problemas:

- Gestión del proyecto
- Edición de los archivos de código fuente
- Depuración

Sería deseable contar con un entorno de desarrollo...

Desarrollo de aplicaciones con herramientas libres



El paquete binutils

Incluye herramientas de desarrollo a diferentes niveles. Entre ellas:

- `ld`: El enlazador de GNU.
- `as`: Compilador de lenguaje ensamblador.
- `addr2line`: Convierte direcciones a nombres de archivos y números de línea
- `ar`: Utilidad para la creación de librerías.
- `nm`: Muestra los símbolos presentes en un archivo objeto.
- `objcopy`: Copia y convierte archivos objeto.
- `objdump`: Muestra información contenida en un archivo objeto.
- `readelf`: Muestra información contenida en un archivo binario con formato 'ELF'.
- `size`: Informa los tamaños de las secciones de un archivo objeto o binario.
- `strip`: Elimina símbolos contenidos en un archivo binario.

`https://www.gnu.org/software/binutils/`

y todo lo demás

- gcc: 'The GNU C Compiler'
  Gcc ARM
  `https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads`
  GCC for windows (tdm-gcc)
  `http://tdm-gcc.tdragon.net/download`
  Ambos paquetes incluyen versiones apropiadas de `gdb`
- `Eclipse`:
  Primero, Java
  `https://www.java.com/en/download/linux_manual.jsp`
  Paquete oficial
  `https://www.eclipse.org/`
  Eclipse + CDT + Pluggins para ARM
  https://github.com/gnu-mcu-eclipse/org.eclipse.epp.packages/releases

Cómo construimos una aplicación

Supongamos que deseamos desarrollar una aplicación en lenguaje C. Hemos utilizado un editor de texto para lograr un archivo llamado 'main.c':

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  printf("hola mundo!");
  return 0;
}
```

¿Cómo podríamos generar el programa asociado utilizando el compilador gcc?

Rta.: Utilizando alguna de las siguientes secuencias de comandos:

```
# Sin información de depuración
$gcc -O2 main.c -o main

# Con información de depuración
$gcc -g -O0 main.c -o main
```

Cómo construimos una aplicación

Será necesario conocer algunos de los parámetros que reconoce GCC y permiten ajustar el proceso de compilación.

- `-On`: Establece el nivel de optimización, donde `-O0` implica el uso de ninguna optimización y `-O3` habilita la máxima cantidad de optimizaciones. Sin embargo, se recomienda utilizar `-O2`.
- *-g*: Agrega información de depuración, la que permite entre otras cosas, realizar ejecución paso a paso de la aplicación, saber el valor de variables y modificarlas en tiempo de ejecución. Si se utiliza este modificador se debe establecer el nivel de optimización a cero.
- `-I/path/to/dir`: agrega el *path* indicado a la ruta de búsqueda de los *headers* del entorno de desarrollo.
- `-c`: compila el archivo de código fuente indicado, no realizando el enlazado.
- `-o`: permite indicar el nombre del archivo resultante del procesado de compilación o enlazado. `gcc main.c -o salida` generará como resultado un programa llamado `salida`.

Cómo construimos una aplicación

En el caso anterior, debido a lo simple de la aplicación, realizamos el proceso de compilación y enlazado en un solo paso. `GCC` puede utilizarse para enlazar una aplicación a partir de los archivos de código objeto asociados. En ese caso puede tomar algunos modificadores especiales:

```
$gcc -O0 -g -Iinc/ -c util.c -o util.o
$gcc -O0 -g -Iinc/ -c main.c -o main.o
$gcc util.o main.o -L/usr/lib -lm -o app
```

- `-L`: Indica la ruta de búsqueda a las librerías que pueden requerirse durante el proceso de enlazado. Si se creara una librería con un fin particular, la ubicación de esta debería indicarse con este parámetro. Es similar al parámetro `-I` durante el proceso de compilación.
- `-l`: Indica que librería debe utilizarse durante el proceso de enlazado. En caso de utilizar enlazado estático, el nombre de la librería deberá ser precedido por 'lib' y tener extensión 'a'. Así la librería matemática se denomina `libm.a` y se incluye a través de `-lm`

Makefiles

Claro está que el desarrollo de aplicaciones requiere trabajar con multitud de archivos y repetir el proceso de construcción de las aplicaciones.

Para resolver esto existen diferentes herramientas para la gestión de proyectos de software. Una de las más utilizadas es `GNU make`.

Esta aplicación procesa archivos de texto conteniendo directivas para lograr diferentes efectos. Si bien ha sido desarrollada para la gestión de proyectos de software, puede utilizarse para automatizar diferentes tareas.

Makefiles

La estructura general de un archivo makefile contiene *objetivos*, *requerimientos* y las *acciones* necesarias para lograr los objetivos a partir de los requerimientos. Por ejemplo:

```
app: main.c
        gcc -O0 -g -c main.c -o main.o
        gcc main.o -o app
```

En este caso el objetivo es un archivo llamado app, el requerimiento es un archivo llamado `main.c` y, bajo ambos se encuentran la lista de acciones a ejecutar para lograr el objetivo.
Si las líneas anteriores se guardan en un archivo llamado `makefile`, y junto a él se encuentra el archivo `main.c`, tras ejecutar el comando `make` construiremos la aplicación.

Makefiles

```
sergio@sdf5: ~/Curso_Embebidos/Diapositivas/Modulo_03/ej/ej02
sergio@sdf5:~/Curso_Embebidos/Diapositivas/Modulo_03/ej/ej02$ ls -lah
total 16K
drwxrwxr-x 2 sergio sergio 4,0K ago 26 21:04 .
drwxrwxr-x 4 sergio sergio 4,0K ago 26 20:53 ..
-rw-rw-r-- 1 sergio sergio   96 ago 26 20:55 main.c
-rw-rw-r-- 1 sergio sergio   65 ago 26 21:04 makefile
sergio@sdf5:~/Curso_Embebidos/Diapositivas/Modulo_03/ej/ej02$ cat makefile
main: main.c
        gcc -O0 -g -c main.c -o main.o
        gcc main.o -o main
sergio@sdf5:~/Curso_Embebidos/Diapositivas/Modulo_03/ej/ej02$ make
gcc -O0 -g -c main.c -o main.o
gcc main.o -o main
sergio@sdf5:~/Curso_Embebidos/Diapositivas/Modulo_03/ej/ej02$ ls -lah
total 32K
drwxrwxr-x 2 sergio sergio 4,0K ago 26 21:04 .
drwxrwxr-x 4 sergio sergio 4,0K ago 26 20:53 ..
-rwxrwxr-x 1 sergio sergio 9,5K ago 26 21:04 main
-rw-rw-r-- 1 sergio sergio   96 ago 26 20:55 main.c
-rw-rw-r-- 1 sergio sergio 3,4K ago 26 21:04 main.o
-rw-rw-r-- 1 sergio sergio   65 ago 26 21:04 makefile
sergio@sdf5:~/Curso_Embebidos/Diapositivas/Modulo_03/ej/ej02$
```

Algunos detalles

- Cuando se invoca al comando make, por defecto, busca un archivo con el nombre `makefile` en el directorio donde se produce la invocación. Si existe, es interpretado. Si se quisiera trabajar con un archivo con otro nombre, es posible, pero el nombre del archivo a procesar debe indicarse con el parámetro `-f`. Por ejemplo:

  ```
  $make -f otroArchivo.mk
  ```

- Make utiliza la fecha y hora de creación de los requerimientos y los objetivos para detectar cambios. Por lo que si ambos están sincronizados entiende que no hay cambios y no repite el proceso de construcción.
- Si en un archivo `makefile` existen múltiples objetivos, el objetivo que se tratará de alcanzar será el primero.
  Es por esto que cuando se tienen diferentes archivos constitutivos de una aplicación se tiene un objetivo que es la aplicación en sí, que tiene como requerimientos otros objetivos. De este modo se encadena el proceso de construcción de las aplicaciones.
- Si un makefile tiene múltiples objetivos y se desea invocar un objetivo particular es posible hacerlo indicando el nombre del objetivo a alcanzar. Por ejemplo:

  ```
  $make -f unMakefile unObejtivo
  ```

Algunos detalles

Por ejemplo, si se separa el proceso de compilación del de enlazado, se podría trabajar con múltiples objetivos.

```
app: main.o
        gcc main.o -lm -o app

main.o: main.c
        gcc -c -O0 -g main.c -o main.o
```

Si se invocase solo `make` se construirá la aplicación `app`, pero si se utilizara como `make main.o` solo se produciría el proceso de compilación generando el archivo `main.o`.

Algunos detalles

Es común que un `makefile` tenga objetivos especiales que no resultan en archivos físicos. Por ejemplo, es común utilizar un objetivo llamado `clean` para eliminar los archivos resultantes del proceso de compilación. En este caso, estos objetivos se indican utilizando la palabra clave `PHONY`.

```
.PHONY: all clean

all: app

app: main.o
     gcc main.o -lm -o app

main.o: main.c
     gcc -c -O0 -g main.c -o main.o

clean:
    rm -rf main
    rm -rf main.o
```

De modo similar al caso del objetivo `clean`, también suele utilizarse un primer objetivo llamado `all` que realiza el proceso completo de compilación de una aplicación.

Algunos detalles

Make soporta el uso de variables. Las variables no tienen tipo y puede contener una palabra, una lista de palabras o el resultado de la ejecución de una aplicación. Si al momento de invocar al comando make se indica el valor de una variable, el valor con el que se haya inicializado en el script será reemplazado por el valor dado. Por ejemplo, si se tiene un makefile como:

```
to:="Mundo"
.PHONY: all
all:
	@echo "Hola $(to)!"
```

Puede ser invocado como:

```
$make
Hola Mundo!

$make to=Clase
Hola Clase!
```

Algunos detalles

- Al utilizar el carácter previo a una acción, hace que no se muestren por pantalla las acciones que se ejecutarán.
- El uso de variables permite lograr makefiles más flexibles, Por ejemplo, si en vez de invocar al compilador por su nombre, lo hiciéramos a través de una variable, podríamos cambiar el compilador utilizado a voluntad.

```
ccName=gcc

.PHONY: all clean

all: app

app: main.o
    @$(ccName) main.o -lm -o app
    @echo "Finalizado!"

main.o: main.c
    @echo -n "Compilando main.c..."
    @$(ccName) -c -O0 -g main.c -o main.o
    @echo "listo!"

clean:
    @rm -rf main
    @rm -rf main.o
```

Reglas implícitas y explícitas

En los ejemplos vistos se han utilizada reglas **explícitas**. Estas indican utilizando nombres de archivos específicos el proceso de construción de las aplicaciones.
Sin embargo, el uso de este tipo de reglas puede ser un problema cuando se trabaja en proyectos con gran cantidad de archivos de código fuente. En estos casos existe la posibilidad de utilizar reglas implícitas.
Este tipo de reglas permite indicar, de modo genérico, la forma en la que se debe lograr un objetivo particular.

```
ccName=gcc

.PHONY: all clean

all: app

app: main.o
    @$(ccName) main.o -lm -o app
    @echo "Finalizado!"

%.o: %.c
    @echo -n "Compilando $<..."
    @$(ccName) -c -O0 -g $< -o $@
    @echo "listo!"

clean:
    @rm -rf main
    @rm -rf main.o
```

Reglas implícitas y explícitas

Además de las variables que pueden definirse dentro de un makefile, existen un conjunto de variables predefinidas:

- `$@`: nombre del archivo deseado (objetivo), nombre de archivo con extensión.
- `$*`: nombre del archivo deseado (objetivo), solo nombre sin extensión.
- `$<`: nombre del archivo requerido incluyendo extensión.
- `$^`: la lista de todos los nombres de archivos requeridos.
- `$+`: similar a `$^` pero incluyendo duplicados (si existieran).
- `$?`: el nombre de todos los archivos que son requeridos y son más nuevos que el objetivo.
- `$(CURDIR)`: donde se está realizando el proceso de construcción.

Funciones

Existe un amplio conjunto de funciones definidas que pueden ser utilizadas para el desarrollo de `makefiles`.
En general, las funciones se invocan como:

```
resultado := $(nombreFuncion arg1,arg2,arg3)
```

Una función particularmente útil es `wildcard`. Esta función retorna la lista de archivos que cumplen con una condición particular en un determinado path o, si se da un nombre como argumento, retornará el nombre del archivo en caso de existir o una cadena vacía. Por ejemplo:

`srcLst:=$(wildcard src/*.c)`	Si dentro de la carpeta `src` existieran los archivos `main.c`, `util.c` y `main.h`, la variable `$(srcLst)` contendrá los valores `src/main.c src/util.c`
`ccName:=/usr/bin/gcc` `ccCmd:=$(wildcard $(ccName))`	Si existe el archivo indicado (`/usr/bin/gcc`), la variable `ccCmd` contendrá el valor `/usr/bin/gcc`. Si el archivo no existe, la variable no contendrá ningún valor.

Condicionales

El comienzo de los `makefile` suelen contener declaración de variables, invocaciones a funciones, invocaciones a comandos y, según el resultado obtenido, puede ser necesario modificar o establecer el valor de variables.
Para realizar estas tareas existe la posibilidad de utilizar estructuras condicionales:

```
ifeq (arg1, arg2)
  [acciones]
endif
```

```
ifeq (arg1, arg2)
  [acciones]
else
  [acciones]
endif
```

```
ifeq (arg1, arg2)
  [acciones]
else ifeq (arg1, arg3)
  [acciones]
else
  [acciones]
endif
```

```
ifneq (arg1, arg2)
  [acciones]
endif
```

```
ifneq (arg1, arg2)
  [acciones]
else
  [acciones]
endif
```

```
ifneq (arg1, arg2)
  [acciones]
else ifneq (arg1, arg3)
  [acciones]
else
  [acciones]
endif
```

Condicionales

Algunos ejemplos:

```
# Comprueba que un archivo exista
ccName:=/usr/bin/gcc
ccExist:=$(wildcard $(ccName))

ifeq (,$(ccExist))
  msg:="Error: $(ccName) [Fail]"
else
  msg:="$(ccName) [Ok]"
endif

.PHONY: all

all:
	@echo $(msg)
```

```
# Determina la ruta a un comando
ccName:=gcc
ccPath:=$(shell which $(ccName))

ifeq (,$(ccPath))
  msg:="$(ccName): not found"
else
  msg:="$(ccName): found $(ccPath)"
endif

.PHONY: all

all:
	@echo $(msg)
```

Otras funciones de interés

- `$(subst ori,alt,texto)`
  Retorna una nueva cadena, donde todas las ocurrencias de 'ori' es reemplazada por 'alt' en la cadena 'texto'.
- `$(strip $(texto))`
  Retorna una cadena de caracteres en la que se omiten los espacios en blanco contenidos al comienzo y al final del texto alojado en la variable $(texto).
- `$(sort $(lista))`
  Retorna el contenido de la variable lista, ordenado alfabéticamente, omitiendo cualquier resultado que pudiera contener.
- `$(shell $(cmd))`
  Retorna el resultado de ejecutar el comando contenido en la variable `cmd` desde el *shell* del sistema operativo utilizado.
  Puede utilizarse para realizar acciones o comandos en el encabezado de un `makefile`. Por ejemplo:
  `dummy:=$(shell mkdir -p out)`

Otras funciones de interés

- `$(dir $(fileNames))`
  Retorna el nombre de las carpetas asociadas a los nombres de archivos contenidos en la variable `$(fileNames)`
- `$(notdir $(fileNames))`
  Retorna el nombre de los archivos contenidos en `$(fileNames)`, quitando el nombre de la carpeta donde está almacenado.
- `$(basename $(fileNames))`
  Retorna los nombres de los archivos contenidos en la `$(filenames)` omitiendo las extensiones de los archivos en caso que la incluyan.
- `$(addsuffix sufijo,$(fileNames))`
  Retorna el resultado de agregar a cada nombre de archivo contenido en la variable `$(fileNames)` el sufijo indicado. Por ejemplo `$(addsuffix .o,src/main)` retornaría `main.o`.
- `$(addprefix prefijo,$(fileNames))`
  Retorna el resultado de agregar al comienzo de cada valor contenido en la variable `$(fileNames)` el prefijo indicado. Por ejemplo `$(addprefix src/,main)` retornará `src/main`.

Otras funciones de interés

- `$(error mensaje)`
  Finaliza la ejecución de un archivo `makefile` indicando una condición de error grave y mostrando por pantalla el mensaje indicado. Agrega junto a este el nombre del `makefile` y la línea donde se ejecutó la función `error`.
- `$(warning mensaje)`
  Funciona de modo similar al caso anterior, pero no finaliza la ejecución del `makefile`. Agrega al mensaje indicado el nombre del `makefile` y la línea donde se realizó la invocación.
- `$(info mensaje)`
  Permite mostrar el mensaje indicado a través de la salida estándar. No agrega ningún tipo de información adicional.

Comandos de utilidad

Desde un `makefile` es posible invocar cualquier comando o programa a través de la función `shell`. En algunos casos el uso de algunos parámetros resulta de utilidad.

- `mkdir -p path/deseado`
  Crea toda la estructura de directorios indicado. Si la estructura de directorios dada existe, no generará un error.
- `rm -rf path`
  Borrará de forma recursiva el `path` indicado. Si el `path` indicado no existe no producirá un error.
- `which comando`
  Retornará el `path` completo al comando indicado en el sistema de archivos. Si no se encuentra el comando o programa en las rutas de búsquedas predeterminadas retornará una cadena vacía.
- `cat archivo`
  Envía el contenido del archivo indicado a la salida estándar. Al ser invocado desde la función `shell` en un `makefile` el contenido del archivo puede ser asignado a una variable.
- `echo $(msg)`
  Envía a la salida estándar el texto contenido en la variable `$(msg)` agregando un fin de línea a continuación del mismo. Si no se desea este agregado puede utilizarse el modificador `-n`.

Utilizando multiples makefiles

Cuando un proyecto contiene múltiples archivos, suelen organizarse en varias carpetas de modo jerárquico, y cada carpeta contiene por un lado el código fuente (carpeta `src` con archivo `.c`) y por otra headers (carpeta `inc` con archivos `.h`).

En estos casos se utiliza más de un makefile para gestionar el proyecto. La vinculación entre ellos puede darse de diferentes maneras.

Un makefile puede incluir otro o un makefile puede producir la invocación de otro.

```
# archivo: conf.mk
export gcc:=/usr/bin/gcc
export ar:=/usr/bin/ar
```

```
# archivo: makefile
include conf.mk

.PHONY:all

all:
@echo $(gcc)
```

Ejemplos como este son muy comunes al momento de trabajar con sistemas embebidos. Un archivo establece la configuración del proyecto a partir de inicializar las variables de modo adecuado mientras que el otro efectivamente realiza el proceso de construcción.

Utilizando múltiples makefiles

La otra alternativa utilizar un makefile de nivel superior que realice la invocación a otros makefiles contenidos en subdirectorios. Del mismo modo que en el caso anterior las variables que se deseen compartir deberán ser exportadas (`export`) desde el makefile principal y/o establecidas durante la invocación.

`makefile` 'principal'

```
.PHONY: all
export saludo="Hola"
all:
    make -C lib msg="clase"
```

`makefile` contenido en la carpeta `lib`

```
.PHONY: all

msg="mundo"

all:
    @echo $(saludo) $(msg)
```

Aspectos de interés:

- Acceso a la variable `saludo` desde el makefile secundario definida en el `makefile` principal.
- Paso del parámetro `msg` al `makefile` secundario.

Construcción de librerías

El primer paso en la construcción de aplicaciones es la compilación. Finalizada esta se da el proceso de enlazado para obtener la aplicación final.
Si se tienen módulos de software que son utilizados por la aplicación suelen compilarse como librerías. Para esto se realiza el proceso de compilación y luego, se empacan los archivos objetos resultantes en una librería de enlazado estático. Esta librería puede ser utilizada posteriormente durante el proceso de enlazado de la aplicación sin necesidad de volver a compilar los archivos de código fuente.
Para lograr esto es necesario utilizar el comando `ar`, los argumentos típicos son:

- `c`: Crea un nuevo contenedor (librería).
- `s`: Agrega un índice al contendor.
- `r`: Inserta los archivos objetos indicados en la librería. Si se cuenta con múltiples definiciones de la misma entidad, serán sobre escritas.

Ejemplo:
```
ar rcs libutils.a calc.o datos.o io.o
```
Se creará una librería de enlazado estático llamada `libutils.a` a partir del código objeto contenido en los archivos `calc.o`, `datos.o` e `io.o`. Para usar esta librería en el proceso de enlazado de una aplicación se podría utilizar una línea como:
```
gcc main.c -lutils -L. -Iinc/ -o main
```