Guion de prácticas 2

Bloque de LEDs

Marzo de 2016

DECSAI

Metodología de la Programación

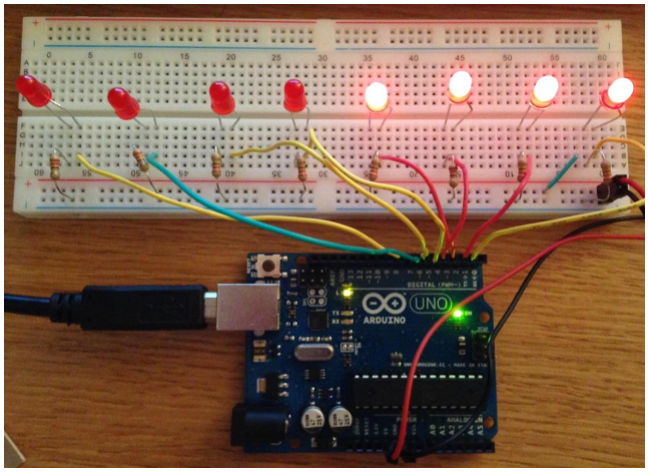
Curso 2015/2016

Índice

1. Definición del problema 5
2. Objetivos 5
3. Operadores a nivel de bit 6
4. Módulo bloqueLed 7
5. Tareas a realizar 9
6. Material a entregar 9

1. Definición del problema

En esta práctica simularemos un bloque de 8 LEDs consecutivos como el que se muestra en la figura siguiente.



Para manejar este bloque de LEDs debemos implementar ciertas operaciones básicas con los LEDs como encender/apagar LEDs determinados, o encender/apagar todos los LEDs simultáneamente.

Supondremos que cada LED puede estar sólo en dos estados, encendido y apagado, y, por tanto, el estado de cada LED se representa con un bit y los 8 LEDs se representan como un byte. En C++, las variables de tipo `unsigned char` ocupan exactamente un byte, por lo cual, el bloque de LED se representará como una variable de dicho tipo. Para facilitar su uso y abstraernos de la representación interna, podemos utilizar un "alias" para `unsigned char` y hacer:

```
typedef unsigned char bloqueLed
```

y de esta manera definir variables de tipo `bloqueLed`.

2. Objetivos

El desarrollo de esta práctica pretende servir a los siguientes objetivos:

- repasar conceptos básicos de funciones,
- practicar el paso de parámetros por referencia,
- practicar el paso de parámetros de tipo array,
- entender el uso de operaciones a nivel de bit,
- reforzar la comprensión de los conceptos de compilación separada y utilización de makefile.

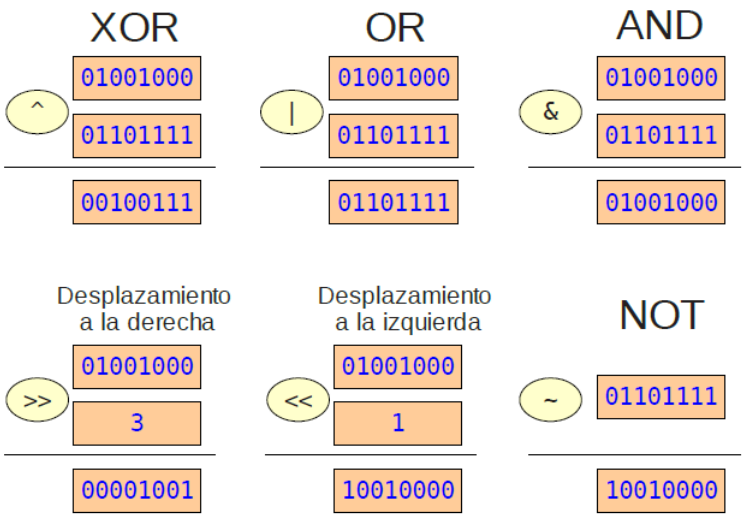


Figura 1: Ejemplos de operaciones a nivel de bit

3. Operadores a nivel de bit

Las funciones de manejo de LEDs concretos (encendido, apagado, consulta de estado, etc.) necesitarán acceder y modificar posiciones específicas de la variable de tipo `bloqueLed`. Es decir, tendremos que manipular los bits de la variable de manera independiente.

El lenguaje C++ ofrece un conjunto de operadores lógicos **a nivel de bit** para operar con diversos tipos de datos, en particular con **caracteres** y **enteros**. Los operadores son:

- operadores binarios, donde la operación afecta a los bits de dos operandos (de tipo `unsigned char`, en nuestro caso):
  - and $(op1 \ \& \ op2)$: devuelve 1 si los dos bits valen 1
  - or $(op1 \mid op2)$: devuelve 1 cuando al menos 1 de los bits vale 1
  - or-exclusivo $(op1 \ \hat{} \ op2)$: produce un valor 1 en aquellos bits en que sólo 1 de los bits de los operandos es 1
  - desplazamiento a la derecha un determinado número de bits (posiciones) $(op1 >> desp)$. Los $desp$ bits más a la derecha se perderán, al tiempo que se insertan bits con valor 0 por la izquierda
  - desplazamiento a la izquierda $(op1 << desp)$: ahora los bits que se pierden son los de la izquierda y se introducen bits a 0 por la derecha
- operador unario de negación $(\sim op1)$: cambia los bits de $op1$ de forma que aparecerá un 0 donde había un 1 y viceversa.

La Fig. 1 muestra un ejemplo de dichas operaciones.

4. Módulo bloqueLed

El módulo a implementar contendrá las siguientes funciones (puedes ver una explicación con más detalle en el fichero `bloqueLed.h` incluído en la carpeta `include`).

```
typedef unsigned char bloqueLed;

// enciende el led pos del bloqueLed b
void on(bloqueLed &b, int pos);

// apaga el led pos del bloqueLed b
void off(bloqueLed &b, int pos);

// devuelve el estado del LED pos
//(encendido = true, apagado = false)
bool get(bloqueLed b, int pos);

// enciende todos los leds
void encender(bloqueLed &b);

// apaga todos los leds
void apagar(bloqueLed &b);

//enciende los leds según la configuracion de v.
// el tamaño de v debe ser 8
void asignar(bloqueLed &b, const bool v[]);

//asigna en v el estado de cada LED
void volcar(bloqueLed b, bool v[]);

// devuelve en posic un vector con las posiciones
// de los leds que están encendidos.
// En cuantos se devuelve el número de leds encendidos
//(número de elementos ocupados en el vector posic).
void encendidos(bloqueLed b, int posic[], int &cuantos);

// devuelve un string con una secuencia de 0's y 1's
// correspondiente al estado de cada LED. Se debe
// implementar utilizando la función "get"
string bloqueLedToString(bloqueLed b);
```

Como consultar y modificar el estado de un LED

Las operaciones de consulta, apagado y encendido de LEDs se traducen a operaciones a nivel de bits. La consulta se corresponde con una lectura y el encendido/apagado con una operación de asignación.

Las operaciones de asignación y lectura de bits se pueden dividir en dos pasos: a) generar una “máscara” (un byte con una secuencia deter-

minada de 0's y 1's) y b) aplicar un operador lógico.

Por convención, asumiremos que el bit más a la derecha representa el primer LED y tiene asignada la posición 0, mientras que el bit de más a la izquierda ocupa la posición 7.

Consulta del estado de un LED

Supongamos que queremos averiguar si el LED en la posición 5 se encuentra encendido. Esto es equivalente a averiguar si el bit en la posición 5 del bloque de LEDs `b` está en 1. Para ello, debemos seguir los siguientes pasos.

1. Crear una máscara (un byte específico) que contenga sólo un 1 en la posición de interés. En este caso: **0010 0000**. Para ello:
   a) Partimos del valor decimal **1** (su codificación en binario es **0000 0001**) [1]
   b) lo rotamos a la izquierda el número de posiciones deseadas (en este caso 5). Utilizando el operador de rotación a izquierda hacemos: `unsigned char mask = 1 << 5`.
2. hacer una operación **AND** entre `b` y `mask`.
3. Si el resultado es distinto de cero, entonces el bit 5 es un 1 y por tanto, el LED está encendido. Caso contrario es un cero y el LED está apagado.

Apagar y Encender de un LED

Apagar un LED implica poner a 0 el bit correspondiente. Supongamos que deseamos apagar el LED de la posición 2. Para poner el bit `k`=2 del bloque de LEDs `b` a cero, se deben seguir los siguientes pasos.

1. generar la máscara `mask` con valor 1111 1011 Para ello:
   a) generar primero una máscara 0000 0100 como hemos explicado antes.
   b) aplicarle el operador de negación NOT.
2. hacer un AND entre `mask` y `b` y guardar el resultado en `b`.

Encender un LED implica poner a 1 el bit correspondiente. Por ejemplo, para poner el bit `k`=2 a 1 del bloque de LEDs `b` haremos:

1. generar la máscara `mask` con valor 0000 0100.
2. aplicar el operador OR entre `mask` y la variable `b`. El resultado se guarda en `b`.

[1] En C++ se pueden escribir literales en hexadecimal precediéndolos por `0x`, p. ej. 32 decimal es `0x20`, o en octal precediéndolos por `0`, p. ej. `040`. Desde C++14, los literales binarios pueden representarse precediéndolos con `0b`, p. ej. `0b00100000`

Secuencias de Animación

El bloque de LEDs puede mostrar una "animación" si se encienden y apagan los LEDs en un orden determinado y se muestran sus valores. A continuación se muestran dos ejemplos.

Ejemplo 1	Ejemplo 2
11111111	11111111
01111111	01111110
10111111	00111100
11011111	00011000
11101111	00000000
11110111	00011000
11111011	00111100
11111101	01111110
11111110	11111111

5. Tareas a realizar

- Implementar las funciones indicadas en el fichero **bloqueLed.h**
- El módulo **test.cpp** ya incluye instrucciones para probar la funcionalidad básica del bloqueLed. Extienda el módulo mostrando como utilizaría las instrucciones `on`, `off` y `bloqueLedToString` para generar las secuencias de animación indicadas en la sección anterior.
- Crear el archivo **makefile** con las órdenes necesarias para generar el ejecutable (**test**) a partir de los archivos fuentes. Debe crear también los objetivos **clean** y **mrproper**.
- Escribir un informe donde consten los nombres y DNI de los integrantes del grupo, los problemas que hayan podido surgir durante el desarrollo de la práctica, capturas de pantalla, etc. Este informe, en formato pdf, se guardará en la carpeta `doc`.

Al completar la implementación debería obtener una salida como la mostrada en la Fig. 2.

6. Material a entregar

Cuando esté todo listo y probado el alumno empaquetará la estructura de directorios anterior en un archivo con el nombre **bloqueLed.zip** y lo entregará en la plataforma **decsai** en el plazo indicado. No deben entregarse archivos objeto (**.o**) ni ejecutables. Para asegurarse de esto último conviene ejecutar **make mrproper** antes de proceder al empaquetado.

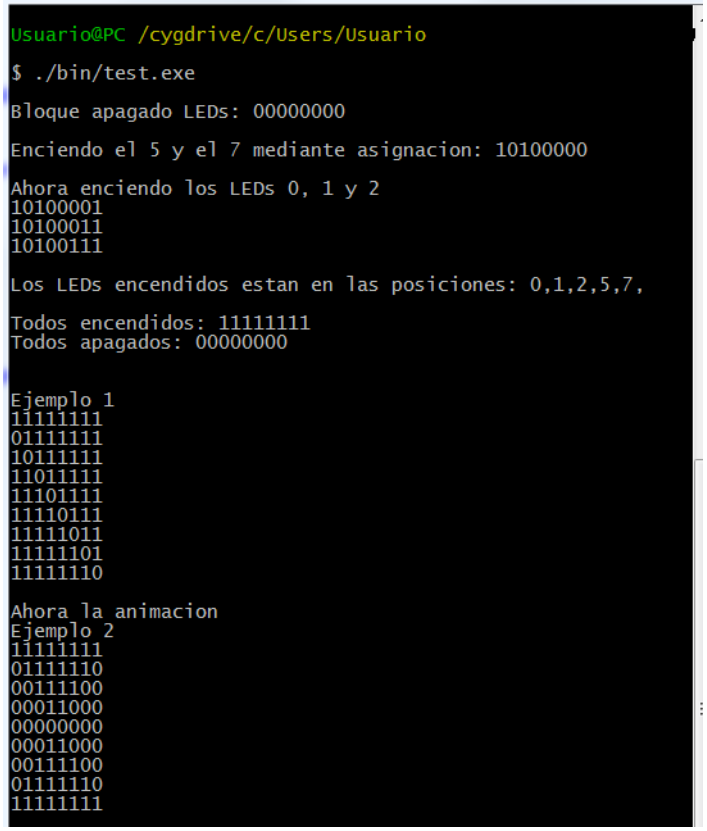


Figura 2: Salida esperada del programa.

El fichero **bloqueLed.zip** debe contener la siguiente estructura:

```
bloqueLed
├── Makefile
├── src
│   ├── bloqueLed.cpp
│   └── main.cpp
├── include
│   └── bloqueLed.h
├── obj
├── bin
└── doc
    └── informe.pdf
```

El alumno debe asegurarse de que ejecutando las siguientes órdenes se compila y ejecuta correctamente su proyecto:

```
unzip bloqueLed.zip
cd bloqueLed
make
bin/main
```