Instituto Tecnológico de Costa Rica
Área Académica de Ingeniería en Computadores
Introducción a los Sistemas Embebidos
Prof. Ing. Jeferson González Gómez

Tutorial: Introducción a drivers en Linux - II

1. Introducción

Este tutorial presenta la continuación del enfoque introductorio a la creación y uso de controladores de dispositivos (*drivers*) en Linux. En el desarrollo de este tutorial se creará un driver para el módulo temporizador (*timer*) de una tarjeta Rasbperrypi 2, utilizando los métodos estándar de Linux para la inicialización, apertura, liberación, lectura, escritura y finalización en controladores de dispositivos, así como las herramientas de construcción cruzada que provee el proyecto *Yocto* (ver tutoriales 4 y 5).

Para el desarrollo de este tutorial será necesario contar con la hoja de datos del fabricante del Sistema en Chip (SoC) de la tarjeta Raspberrypi 2 (Broadcom BCM2835), que contiene información del funcionamiento y configuración los diferentes periféricos dentro del SoC. Dicha hoja de datos, puede descargarse de: `https://cdn-shop.adafruit.com/product-files/2885/BCM2835Datasheet.pdf`

2. Funcionamiento del módulo temporizador

Un módulo temporizador o *timer* es un dispositivo de hardware que permite llevar diferentes tiempos y conteos dentro de un sistema en chip o computador, en general. Un *timer* genérico permite realizar conteo de tiempo, tanto ascendente como descendente, a frecuencias determinadas (típicamente programables, escalando la frecuencia de reloj del sistema). Adicionalmente, el *timer* permite la carga de un valor de conteo objetivo, de manera programática, y puede generar una interrupción al procesador al cumplirse el tiempo de conteo programado.

El funcionamiento *timer* del SoC BCM2835 se encuentra descrito de las páginas 196 a la 199 de la hoja de datos del fabricante. En esta sección se presentará un resumen de las funcionalidades más importantes del mismo.

El *timer* del SoC, que se basa en un modelo ARM AP804, tiene dos modos de funcionamiento: *timer* tradicional, que permite realizar conteo descendente a partir de un valor de carga determinado programáticamente, y modo de corrida libre, en el que el temporizador de 32 bits aumentará el conteo desde 0 hasta su máximo valor $(2^{32} - 1)$ y regresará a 0, para nuevamente iniciar el conteo, de manera continua. Este tutorial se centra en el modo de corrida libre.

Como la mayoría de dispositivos de E/S dentro de un SoC, el temporizador interactúa con el procesador por medio de registros, que son accedidos por medio de una dirección base (*base address*) y un desplazamiento (*offset*). Para el caso del BCM2835 la dirección base para el *timer* corresponde a $0x7E00B000$, esta dirección corresponde a una dirección base del SoC, sin

Address offset[8]	Description
0x400	Load
0x404	Value (Read Only)
0x408	Control
0x40C	IRQ Clear/Ack (Write only)
0x410	RAW IRQ (Read Only)
0x414	Masked IRQ (Read Only)
0x418	Reload
0x41C	*Pre-divider (Not in real 804!)*
0x420	*Free running counter (Not in real 804!)*

Figura 1: Registros del módulo temporizador del BCM2835

embargo, el sistema operativo lo asigna a una dirección distinta, por medio de un *offset* a todas las direcciones SoC. Para el caso de este tutorial, la dirección base del *timer* será: $0x3F00B000$. Los registros del *timer* (subsección 14.2) se muestran en la Fig.1

Para los objetivos de este tutorial, se abarcarán solamente los registros de control (**TCR** : *timer control register*) y de corrida libre (**FRCR**: *free running counter register*), accesibles en los *offsets* $0x408$ y $0x420$, respectivamente. Es importante destacar que al crear un driver para un dispositivo físico, será necesario posteriormente reservar espacio de memoria para escribir o leer del mismo, a través de los registros para tal fin, por lo que siendo que cada registro es de 32 bits, se deberá reservar desde la dirección base, un total de $0x424$ direcciones.

2.1. Registro de control - TCR

El registro de control del temporizador (TCR), permite configurar el *timer* para los diferentes modos de operación. En la Fig.2 se muestran los bits que componen el TCR y su funcionalidad. Para este tutorial serán importantes los bits 9 y 23-16. El 9 permite iniciar el modo de corrida libre, estableciéndolo en 1, mientras que los bits 23-16, permiten establecer el pre-escalar, que determina la frecuencia a la que debe contar el temporizador. La frecuencia final de conteo está dado por

$$Fc = \frac{sys_{freq}}{1 + pre - escalar} \tag{1}$$

donde sys_{freq} corresponde a la frecuencia del sistema. Para el caso de Linux, en raspberrypi 2, este valor es de 600MHz. Esta información puede ser verificada con el comando:

```
# cat /sys/devices/cpu/cpu0/cpufreq/cpuinfo_cur_freq
```

Es importante resaltar que en *reset* el registro TCR está inicializado en $0x003E0020$ por lo que el bit 9 se encuentran en 0 por defecto, y el pre-escalar se encuentra en 0x3E

2.2. Registro de contador de corrida libre - FRCR

El registro FCRC (Fig.3), accesible en el *offset* $0x420$, es de solo lectura, y contiene el valor del conteo, en modo de corrida libre.

Name: Timer control		Address: base + 0x40C	Reset: 0x3E0020
Bit(s)	**R/W**	**Function**	
31:10	-	<Unused>	
23:16	*R/W*	*Free running counter pre-scaler. Freq is sys_clk/(prescale+1)* ***These bits do not exists in a standard 804! Reset value is 0x3E***	
15:10	-	<Unused>	
9	*R/W*	*0 : Free running counter Disabled* *1 : Free running counter Enabled* ***This bit does not exists in a standard 804 timer!***	
8	*R/W*	*0 : Timers keeps running if ARM is in debug halted mode* *1 : Timers halted if ARM is in debug halted mode* ***This bit does not exists in a standard 804 timer!***	
7	R/W	0 : Timer disabled 1 : Timer enabled	
6	*R/W*	***Not used**, The timer is always in free running mode.* *If this bit is set it enables periodic mode in a standard 804. That mode is not supported in the BC2835M.*	
5	R/W	0 : Timer interrupt disabled 1 : Timer interrupt enabled	
4	R/W	<Not used>	
3:2	R/W	Pre-scale bits: 00 : pre-scale is clock / 1 (No pre-scale) 01 : pre-scale is clock / 16 10 : pre-scale is clock / 256 11 : pre-scale is clock / 1 *(Undefined in 804)*	
1	R/W	0 : 16-bit counters 1 : 23-bit counter	
0	*R/W*	***Not used**, The timer is always in wrapping mode.* *If this bit is set it enables one-shot mode in real 804. That mode is not supported in the BCM2835.*	

Figura 2: Registro de control de *timer*

Name: Free running		Address: base + 0x420	Reset: 0x000
Bit(s)	**R/W**	**Function**	
31:0	R	Counter value	

Figura 3: Registro corrida libre del *timer*

3. Driver para módulo temporizador

En esta sección, se realizará la implementación de un *driver* sencillo para el módulo *timer*, en modo de corrida libre. Para esto, se partirá de un código base (<timer_base>), mostrado a continuación.

```
/* Necessary includes for device drivers */
#include <linux/init.h>
#include <linux/version.h>
#include <linux/module.h>
#include <linux/kernel.h> /* printk() */
#include <linux/slab.h> /* kmalloc() */
#include <linux/fs.h> /* everything... */
#include <linux/errno.h> /* error codes */
#include <linux/types.h> /* size_t */
#include <linux/proc_fs.h>
#include <linux/fcntl.h> /* O_ACCMODE */
#include <linux/uaccess.h> /* copy_from/to_user */
#include <linux/ioport.h>
#include <asm/io.h>

/*Local defined macros*/
#define BUF_LEN 80
#define DEVICE_NAME "timer"
#define SUCCESS 0

static unsigned RANGE =  0x404;
static unsigned TIMER_BASE =  0x3f00B000;
static char msg[BUF_LEN]; /* The msg the device will give when asked */
static char *msg_Ptr;
static int Device_Open = 0; /* Is device open?  */
u8 *addr;

MODULE_LICENSE("GPL v2");

/* Declaration of timer.c functions */
int timer_open(struct inode *inode, struct file *filp);
int timer_release(struct inode *inode, struct file *file);
ssize_t timer_read(struct file *filp, char *buffer,
                   size_t length, loff_t * offset);
ssize_t timer_write(struct file *filp, const char *buffer,
                    size_t length, loff_t * offset));
void timer_exit(void);
int timer_init(void);


/* Structure that declares the usual file */
/* access functions */
struct file_operations timer_fops = {
  read: timer_read,
```

```
  write: timer_write,
  open: timer_open,
  release: timer_release
};

/* Declaration of the init and exit functions */
module_init(timer_init);
module_exit(timer_exit);

/* Global variables of the driver */
/* Major number */
int timer_major = 60;
```

En el código, inicialmente se procede a incluir los encabezados necesarios para utilizar las funciones en espacio de kernel, como reserva de memoria, impresiones en consola del sistema, códigos de error, etc. Posteriormente se definen los macros y variables globales a utilizar en el *driver*. La variable RANGE contiene el rango de direcciones a reservar y TIMER_BASE se asigna a la dirección base en la que se encuentra asignado el módulo *timer* en el sistema, como se describió anteriormente. Luego, se declaran las funciones de operación del dispositivo: abrir, liberar, leer, escribir, salir e inicializar, necesarias para todo driver de dispositivo, como se describió en el tutorial 6. Finalmente, se define la estructura que relaciona las operaciones sobre archivos en espacio de usuario, con las funciones sobre el dispositivo, en espacio de kernel. Adicionalmente se especifica el nombre de las funciones de inicializar y salir, y se define en número *Major* para el dispositivo.

A continuación se procederá a implementar cada una de las funciones para operar sobre el dispositivo.

Inicialización del dispositivo

Como se describió en el tutorial anterior, la función de inicialización (*init*) del controlador, debe reservar el espacio de memoria asignado del dispositivo, así como realizar las escrituras necesarias en el dispositivo para que este pueda ser utilizado por las demás funciones. A continuación se muestra la implementación de la función de inicialización:

```
/*********Functions*********/

/* Device init*/
int timer_init(void) {
    int result;
    /* Registering device */
    result = register_chrdev(timer_major, DEVICE_NAME, &timer_fops);
    if (result < 0) {
     printk("<1>timer: cannot obtain major number %d\n", timer_major);
     return result;
    }
    /*Reserve timer memory region */
    if(request_mem_region(TIMER_BASE, RANGE, DEVICE_NAME) == NULL) {
     printk("<1>timer: cannot reserve memory \n");
     unregister_chrdev(timer_major, DEVICE_NAME);
     return -ENODEV;
```

```
    }

    printk("<1>Inserting timer module... :)\n");

    /*Generate new address required for iowrite and ioread methods*/
    addr = ioremap(TIMER_BASE, RANGE);

    /* Set free running frequency to half of sys_freq*/
    u32 cmd = 0x00010020;

    /*Write to Timer Control Register*/
    iowrite32(cmd, (addr+ 0x408));

    /*Check it has been configured*/
    printk("timer: Timer control set to 0x%08x\n", ioread32(addr+ 0x408));

  return SUCCESS;
}
```

En el código, inicialmente se registra el dispositivo y se liga con el nombre del archivo correspondiente (DEVICE_NAME). Luego, se procede a reservar la región de memoria correspondiente al temporizador. Esto se realiza por medio del método *request_mem_region*, que toma como argumentos, respectivamente, la dirección base del dispositivo, el rango de direcciones a reservar, y el nombre del recurso a quién será asignada esta región de memoria. Una vez reservado el espacio de memoria, se procede a inicializar el dispositivo, por medio de una escritura al registro de control, para habilitar el modo de corrida libre. Para realizarlo, se debe primero generar un re-mapeo de la dirección base y rango de direcciones, a una dirección virtual que pueda ser manejada por el programa, esta acción la realiza el método *ioremap*. Posteriormente se genera el comando para configurar el pre-escalar (bits 23-16) a 0x01, de forma que el conteo se realice a la mitad de la frecuencia del sistema, según la Ec.1. El comando a escribir es entonces *0x00010020*. La escritura del comando se realiza mediante el método *iowrite32*, tomando como argumentos el comando y la dirección a escribir. Cabe destacar que la dirección a escribir, en este caso, corresponde a la nueva dirección virtual, obtenida previamente, más el *offset* correspondiente al registro de control (0x408). Finalmente, se lee del registro de control, por medio del método *ioread32*, para verificar que el temporizador se haya configurado correctamente.

Remover el dispositivo

Para remover el dispositivo debe implementarse el método *module_exit*. La implementación se muestra a continuación:

```
void timer_exit(void) {
  /* Freeing the major number and memory space */
  unregister_chrdev(timer_major, DEVICE_NAME);
  release_mem_region(TIMER_BASE, RANGE);
  printk("<1>Removing timer module\n");
}
```

Dado que en la inicialización se reservó una región de memoria para el driver, en la etapa de remover debe liberarse dicho espacio, por medio del método así como liberar el registro del major. Esto se realiza, en el código, con el método *release_mem_region*

Abrir el dispositivo

La apertura del dispositivo se realiza cada vez que hay escrituras o lecturas al mismo. Para el caso de este *driver* no se implementará ninguna funcionalidad adicional. La implementación se muestra a continuación:

```
int timer_open(struct inode *inode, struct file *filp) {
printk("<1>Opening timer module\n");
/*Avoid reading conflicts*/
if (Device_Open)
return -EBUSY;

Device_Open++;
    return SUCCESS;
}
```

En el código, se utiliza una bandera (*Device_Open*) para evitar aperturas simultáneas al dispositivo.

Cerrar el dispositivo

El cierre del dispositivo se realiza al finalizar las lecturas o escrituras al dispositivo. En algunos dispositivos puede ser útil aplicar configuraciones en esta etapa, para reiniciar a valores por defecto registros, etc. Para el caso del módulo *timer* no se aplicará ninguna funcionalidad extra. El código se muestra a continuación:

```
int timer_release(struct inode *inode, struct file *file){
printk("<1>Releasing  timer module...\n");
Device_Open--;      //make ready for the next caller*/
return SUCCESS;
}
```

En el código, se libera la bandera para permitir posteriores aperturas al dispositivo.

Escribir al dispositivo

En el método de escritura se implementará una funcionalidad que permite iniciar ('1') y detener ('0') el *timer* en el modo de corrida libre. Este proceso se debe realizar escribiendo al bit 9 del registro de control, ubicado en el *offset* 0x408. El comando para iniciar el conteo, será entonces *0x00010220*, en el que se mantiene el pre-escalar de 1, y se habilita el bit 9 del registro. Para el caso en que se detiene el conteo, el comando será *0x003E0020*, mismo que vuelve el registro de control a su valor por defecto, en el que el bit 9 se encuentra en 0, y el pre-escalar regresa a 0x3E. Cabe destacar que el *timer* no tiene una lógica de *reset*, por lo que cualquier lectura posterior a haber detenido el conteo, retornará el mismo valor. El código se muestra a continuación:

```
ssize_t timer_write(struct file *filp, char *buffer,
                    size_t length, loff_t * offset){
    printk("<1>Writing to timer module...\n");
    u32 cmd;
    if (buffer) {
         /* if ascii 0*/
         if (buffer[0] == 48) {
            /*Stop timer - bit 9 OFF*/
```

```
            cmd = 0x003E0020;
            iowrite32(cmd, (addr+ 0x408));
            printk("timer: OFF \n");
            }
         else {
            /*if ascii 1*/
            if (buffer[0] == 49){
                /*Start timer - bit 9 ON*/
                cmd = cmd = 0x00010220;
                iowrite32(cmd, (addr+ 0x408));
                printk("timer: ON \n");
             }
         }
    }
    /*Buffer contains only 1 char (0-1)
    returns "1 byte written"*/
return 1;
}
```

En el código, se utiliza el caracter '0' (48, en decimal) para detener el conteo y el caracter '1' (decimal 49), para iniciarlo. Por lo tanto, en **espacio de usuario**, deberá escribirse '0' o '1' al archivo correspondiente (/dev/timer) para iniciarlo o detenerlo. Las escrituras en **espacio de kernel**, se realizan con el método *iowrite32*, descrito previamente. Dado que el *buffer* que se usa desde espacio de usuario hacia espacio de kernel es de 1 byte ('0' o '1'), el método *timer_write* debe retornar 1 (1 byte), aunque la escritura con el dispositivo real sea de una longitud mayor.

Lectura del dispositivo

La lectura del valor del conteo se realiza en el registro FRCR, en el *offset* 0x420. El método para lectura del dispositivo (*timer_read*) debe leer, por medio de *ioread32* en este *offset* y copiar la respuesta obtenida a espacio de usuario, para que sea recibida por los métodos de lectura habituales (cat, fread, etc). La función de lectura *timer_read*, se muestra a continuación:

```
ssize_t timer_read(struct file *filp, char *buffer,
                   size_t length, loff_t * offset){
    int index;
    u32 res;
    msg_Ptr = msg;
    int nbytes = 8;

    if (*offset == 0) {

         /* Reading FRCR*/
         res = ioread32(addr+0x420);

         /*Mask to get bytes of message*/
         msg[3] = res & 0xFF;
         msg[2] = (res >> 8) & 0xFF;
         msg[1] = (res >> 16) & 0xFF;
         msg[0] = (res >> 24) & 0xFF;
```

```
        /*Convert to Hex ASCII */
        sprintf(msg, "%02hhx%02hhx%02hhx%02hhx",
                msg[3],msg[2],msg[1], msg[0]);

        /*Copy to user buffer*/
        copy_to_user(buffer,msg,nbytes);

        *offset+=1;

        printk("timer: timer count = %d \n", res);
        return nbytes;
    } else
        return SUCCESS;
}
```

En el código, luego de la lectura del FRCR, se realiza un desplazamiento (>>) y enmascarado con 0xFF, esto, para obtener de manera independiente cada uno de los bytes que conforman la respuesta (valor de conteo). Posteriormente, cada uno de estos bytes, en hexadecimal, se convierte a caracteres, para ser retornados a espacio de usuario, como una cadena imprimible de 8 bytes, mediante el método *copy_to_user*.

Compilación del driver

Una vez diseñado el controlador del módulo temporizador, se procederá a realizar su compilación cruzada. Para esto se utilizará las herramientas que brinda el proyecto Yocto. A continuación se describen los pasos para realizar la compilación.

Archivo Makefile

Como primer paso para la compilación del *driver*, se creará un archivo Makefile, el cual será invocado de manera automática por la herramienta de construcción de Yocto. El Makefile, que me muestra a continuación, es similar al realizado en el tutorial anterior:

```
obj-m := timer.o

SRC := $(shell pwd)

all:
[tab][tab]$(MAKE) -C $(KERNEL_SRC) M=$(SRC)
modules_install:
[tab][tab]$(MAKE) -C $(KERNEL_SRC) M=$(SRC) modules_install

clean:
[tab][tab]rm -f *.o *~ core .depend .*.cmd *.ko *.mod.c
[tab][tab]rm -f Module.markers Module.symvers modules.order
[tab][tab]rm -rf .tmp_versions Modules.symvers
```

En el Makefile, se implementa el comando de compilación del módulo del kernel, en la regla *all:*. Se implementa además regla para la instalación del módulo dentro de los módulos de kernel, y para limpiar la compilación.

Archivo de licencia

El proyecto Yocto, como participante de la iniciativa de *Open Source*, requiere que los paquetes a construir incluyan un archivo de licencia y copiado. Para este tutorial se utilizará una licencia GPLv2 (`http://tinyurl.com/zrcj2wt`). El archivo de licencia, denominado COPYING, se encuentra en el tecDigital, en la sección de documentos. Otro tipo de licencia podría ser utilizado, pero siempre debe existir el archivo correspondiente, así como conocer el md5 *checksum* del mismo.

Archivo de receta (.bb)

Ahora se creará un archivo de receta (.bb) específico para el módulo. Este archivo de receta será parte de un paquete que se incluirá en la imagen de Linux, creada por Yocto, para la raspberrypi 2. El contenido del archivo timer-mod.bb se muestra a continuación:

```
SUMMARY = "Linux module for BCM2835 timer"
LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://COPYING;md5=12f884d2ae1ff87c09e5b7ccc2c4ca7e"

inherit module

SRC_URI = "file://Makefile \
           file://timer.c \
           file://COPYING \
          "

S = "${WORKDIR}"

# The inherit of module.bbclass will automatically name module packages with
# "kernel-module-" prefix as required by the oe-core build environment.
```

En el archivo, se incluye un resumen del paquete, el tipo de licencia a utilizar y el *checksum* md5 del archivo de licencia. Se incluye además el macro SRC_URI, que determina cuáles archivos forman parte del codigo fuente. La notación *file://...* indica que los archivos Makefile, timer.c y COPYING se encuentran dentro de un directorio estándar llamado *files*, dentro del directorio del paquete a construir.

Estructura del paquete timer-mod

Para la construcción de un paquete propio dentro del Yocto, debe crearse un directorio con el nombre del paquete, dentro de alguno de los directorios pertenecientes a las capas incluidas (*layers*) del proyecto de construcción. Para facilitar la construcción, se creará un directorio llamado recipes-drivers, dentro del directorio meta-raspberrypi, mencionado en el tutorial 5.
Una vez dentro del directorio recipes-drivers, se creará un directorio llamado timer-mod. Dentro de timer-mod, se encontrará el archivo de receta timer-mod.bb, así como el directorio *files*. Así mismo, dentro del directorio *files* se encontrará el Makefile, código fuente del driver (timer.c) y archivo de licencia (COPYING).

Archivo de configuración local**

En el directorio de construcción de la imagen para Raspberrypi 2, se deberá modificar el archivo de configuración conf/local.conf. Ya que el módulo del driver se construirá contra el código

de fuente del kernel, es **fundamental** que la imagen **NO** haya sido construida con el macro INHERIT += rm_work". Si este fue el caso, deberá crearse un nuevo directorio de construcción exclusivamente para la compilación del paquete con el driver, esto ya que Yocto no permite la inclusión de un módulo de kernel dentro de una imagen que ya fue construida, de manera natural. Por esta razón se construirá solamente el paquete, y la copia al sistema de archivos se realizará manualmente. Si es la primera vez que construye la imagen, se puede agregar el macro CORE_IMAGE_EXTRA_INSTALL += "timer-mod", esta instrucción agregará automáticamente el módulo del timer a la imagen. Sin embargo, como ya se mencionó, para este tutorial se asume que la imagen ya ha sido creada previamente, por lo que se detallará el procedimiento para la copia manual del módulo al sistema de archivos de la imagen.

Construcción e instalación del driver

Ya sea con un directorio de construcción nuevo (configurado correctamente para la imagen de raspberrypi 2) o un directorio previo, para realizar la construcción del driver es necesario ejecutar el comando:

```
# bitbake timer-mod
```

Este comando realiza la construcción del módulo, pero este debe ser copiado manualmente al sistema de archivos. El archivo timer.ko recién construido deberá encontrarse, a partir del directorio de construcción, en el directorio:

```
tmp/work/raspberrypi2-poky-linux-gnueabi/timer-mod/1.0-r0/
```

Se recomienda que este archivo sea copiado en el sistema de archivos de la tarjeta Raspberry pi 2, en el directorio /lib/modules/extra.

Verificación del controlador

Para verificar el funcionamiento del driver, una vez dentro de la imagen de Linux de la Raspberry pi2, se ejecutarán los siguientes pasos.

Archivo de dispositivo

Primeramente, dentro de la Raspberry pi 2, se deberá crear el archivo de dispositivo correspondiente al driver diseñado para el temporizador. Esto es:

```
# mknod /dev/timer c 60 0
```

y se cambian sus permisos para ser escrito y leído correctamente:

```
# chmod 666 /dev/timer
```

Inicialización del driver

Seguido, se deberá inicializar el driver por medio del comando insmod:

```
# insmod /lib/modules/extra/timer.ko
```

La verificación de la inicialización puede verse con el comando:

```
# dmesg | tail
```

donde deberá asegurarse que el dispositivo responda con el comando 0x00010020, como es esperado. Un punto importante a verificar es que el sistema haya reservado la región de memoria correspondiente para el dispositivo. Esto puede verse con el comando:

```
# cat /proc/iomem
```

La salida de este comando genera el mapa actual de memoria y dispositivos de entrada/salida del sistema. El comando deberá mostrar en una de sus líneas:

```
# 3f00b000 - 3f00b423 : timer
```

que corresponde a la dirección base y rango de direcciones reservados para el módulo temporizador.

Adicionalmente, puede verificarse que el dispositivo haya sido inicializado leyendo del archivo de dispositivo, por medio de:

```
# cat /dev/timer
```

cuya salida, en la primera ejecución debería ser 00000000. Dado que el conteo no ha sido iniciado.

Inicio de conteo

Como se mencionó previamente, el inicio de conteo se realiza escribiendo un '1' al archivo de dispositivo. Esto se puede realizar con:

```
# echo -n 1 > /dev/timer
```

El comando # dmesg puede usarse además para verificar que la escritura se haya realizado correctamente.

Lectura de conteo

Una vez iniciado el conteo, puede leerse el valor del contador. Esto se puede realizarse, de nuevo, con:

```
# cat /dev/timer
```

Pueden realizarse varias lecturas, para comprobar que el conteo se está realizando efectivamente.

Fin el conteo

Para finalizar el conteo, debe escribirse un '0' al archivo de dispositivo. Esto es:

```
# echo -n 0 > /dev/timer
```

Para verificar que se haya detenido el conteo, pueden realizarse varias lecturas, en las que el valor de conteo debe permanecer constante. El valor no será 0, ya que el conteo no puede ser *resetado*.

4. Evaluación

Para la evaluación de este tutorial, se deberá implementar una biblioteca dinámica, para la tarjeta raspberrypi, que permite la generación de números seudo-aleatorios, así como la medición de tiempos de ejecución, utilizando el driver diseñado anteriormente. Deberá implementarse un archivo de prueba que verifique el funcionamiento de la biblioteca. Debe seguir la estructura estandar (lib, src, etc), así como utilizar algún gestor de compilación (autotools, cmake, etc).

Entrega

Para la entrega, se deberá subir un archivamiento al tecDigital con nombre Nombre_Apellido_-tar.gz, que incluya todos los códigos fuentes, así como los pasos para la compilación.