Diseño Basado en Microprocesadores
Práctica 8
Timers del LPC4088

Índice

---

1. Introducción

Los timers son periféricos internos de los microcontroladores que pueden actuar como contadores o como temporizadores. Como contadores permiten contar eventos externos y como temporizadores posibilitan generar y medir intervalos de tiempo de forma precisa. El uso combinado de los timers y del sistema de interrupciones permite llevar a cabo este tipo de tareas con muy poca intervención de la CPU. Sin embargo, como aún no se ha estudiado el sistema de interrupciones, en esta práctica los timers se sondearán por software para detectar por los eventos de *match* de los timers y no se habilitarán las interrupciones correspondientes. El uso conjunto de timers e interrupciones se verá en una práctica posterior.

2. Características de los timers del LPC4088

El microcontrolador LPC4088 posee cuatro timers llamados timer 0, timer 1, timer 2 y timer 3. Cualquiera de estos timers puede usarse para generar o medir intervalos de tiempo así como para contar eventos externos. El funcionamiento de los cuatro timers es muy parecido. Básicamente, cada timer permite llevar a cabo el conteo de ciclos del reloj de periféricos (PCLK) o de pulsos aplicados externamente a determinadas patillas del encapsulado del microcontrolador. Cada ciclo de reloj o pulso externo incrementa el valor de un registro interno del timer llamado registro contador. Cada timer cuenta también con cuatro de registros de *match* o coincidencia. Cuando el número de ciclos o pulsos alcanza un valor igual al indicado por alguno de los registros de match, el timer puede llevar a cabo una serie de acciones, incluyendo la solicitud de una interrupción a la CPU. Cada timer cuenta también con dos entradas de captura que permiten capturar el valor del registro de contador del timer en sendos registros de captura cuando en ellas se produce una transición. Una acción de captura también puede solicitar una interrupción a la CPU. Los timers poseen asimismo salidas de match, que pueden poner a cero, poner a uno o invertir el estado de los pines asociados cuando se produce la coincidencia del registro contador con un registro de match. De forma resumida, las características fundamentales de cada timer son:

- Funcionamiento como temporizador o como contador.
- Registro contador de 32 bits.
- Registro de preescala de 32 bits.
- Hasta dos registros de captura de 32 bits por timer que pueden capturar al vuelo el valor del registro contador del timer cuando se produce una transición en una señal de entrada. Opcionalmente, un evento de captura puede generar una interrupción.
- Cuatro registros de *match* que pueden usarse para:
  - Generar una interrupción cuando se produce una coincidencia entre los valores del registro contador del timer y del registro de match.
  - Parar el registro contador del timer cuando se produzca una coincidencia, con generación opcional de una interrupción cuando se produce dicha coincidencia.
  - Poner a cero el registro contador del timer cuando se produzca una coincidencia, con generación opcional de una interrupción cuando se produce la coincidencia.

- Hasta cuatro salidas externas, una para cada registro de *match* que permiten:
  - Poner la salida a nivel bajo cada vez que se produzca una coincidencia.
  - Poner la salida a nivel alto cada vez que se produzca una coincidencia.
  - Cambiar el estado de la salida cada vez que se produzca una coincidencia.

En la figura 1 podemos ver el diagrama de bloques proporcionado por el fabricante, válido con pequeñas diferencias para cualquiera de los cuatro timer.

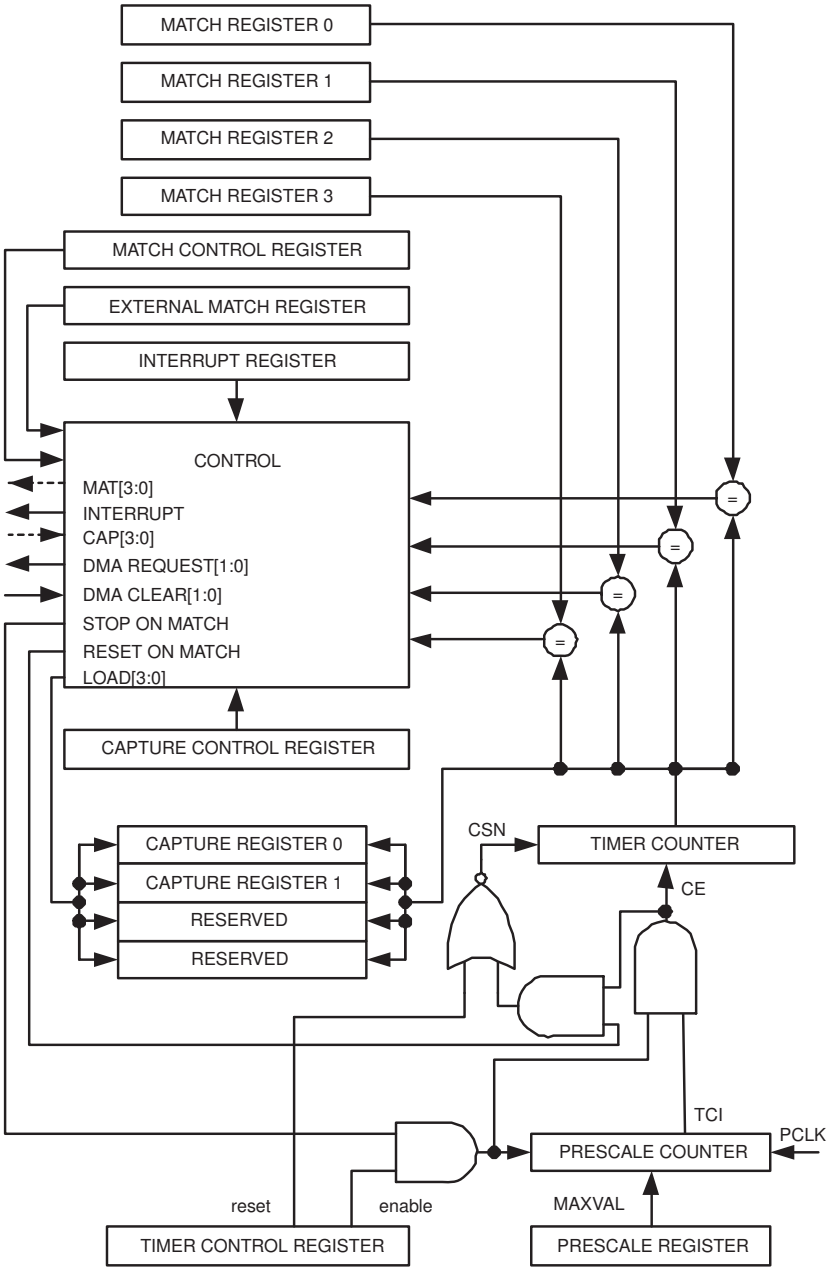

Figura 1: Diagrama de bloques de uno cualquiera de los timers del LPC4088.

3. Pines asociados a los timers

Cuadro 1: Pines asociados a los timers

Pin	**Tipo**	**Descripción**
T0_CAP1:0 T1_CAP1:0 T2_CAP1:0 T3_CAP1:0	Entradas	Entradas de captura. Los pines de captura pueden ser configurados para que una transición aplicada a los mismos cargue el valor del registro contador del timer en uno de los registros de captura y, opcionalmente, genere una interrupción. La función de captura está disponible en varios pines. Si se selecciona más de un pin para realizar la función de captura en un único canal del timer, se usará el pin con el número más bajo. Puede usarse un pin de captura para aplicar al timer una señal de reloj externa en lugar del reloj PLCK interno.
T0_MAT1:0 T1_MAT1:0 T2_MAT3:0 T3_MAT1:0	Salidas	Salidas externas de coincidencia (*match*). Cuando el valor de uno de los registros MR0, MR1, MR2 o MR3 es igual al registro contador del timer, TC, estas salidas pueden ponerse a nivel bajo, a nivel alto, cambiar de estado o no hacer nada. El registro *External Match Register*, EMR, controla la función de estas salidas. Estas funciones pueden ser seleccionadas para varios pines a la vez.

Es posible seleccionar varios pines para la mayoría de las funciones de captura (CAP) y match (MAT). Cuando se seleccionan varios pines para la función de salida MAT, todos los pines cambiarán de la misma forma. Cuando se selecciona más de un pin para la función de entrada CAP, el timer sólo usará el pin con el número de pin de puerto más bajo.

4. Descripción de los registros de los timers

Cuadro 2: Registros de los timers. Continúa en el cuadro 3

Nombre	**Descripción**	**Acceso**	**Reset**	**Direcciones**
IR	Registro de interrupción. Puede leerse para identificar la causa concreta de una petición de interrupción generada por el timer. Puede escribirse para borrar los bits de petición de interrupción.	R/W	0	TIMER0 – 0x4000 4000 TIMER1 – 0x4000 8000 TIMER2 – 0x4009 0000 TIMER3 – 0x4009 4000
TCR	Registro de control del timer. El registro TCR permite controlar el registro contador del timer, TC. El registro contador del timer puede ser habilitado o deshabilitado así como puesto a cero mediante el TCR.	R/W	0	TIMER0 – 0x4000 4004 TIMER1 – 0x4000 8004 TIMER2 – 0x4009 0004 TIMER3 – 0x4009 4004
TC	Registro contador del timer. El registro contador del timer se incrementa en una unidad cada PR+1 ciclos del reloj PCLK, siendo PR el registro de preescala. El registro TC se controla mediante el registro TCR.	R/W	0	TIMER0 – 0x4000 4008 TIMER1 – 0x4000 8008 TIMER2 – 0x4009 0008 TIMER3 – 0x4009 4008
PR	Registro de preescala. Cuando el valor del registro contador de preescala, PC, es igual al valor del registro de preescala, PR, el siguiente pulso del reloj PCLK incrementa el registro TC y pone a cero el registro PC.	R/W	0	TIMER0 – 0x4000 400C TIMER1 – 0x4000 800C TIMER2 – 0x4009 000C TIMER3 – 0x4009 400C
PC	Contador de preescala. El registro PC se incrementa con cada pulso del reloj PCLK. Cuando su valor coincide con el del registro PR, el siguiente pulso de reloj hace que el registro TC se incremente y el PC se pone a cero.	R/W	0	TIMER0 – 0x4000 4010 TIMER1 – 0x4000 8010 TIMER2 – 0x4009 0010 TIMER0 – 0x4009 4010
MCR	Registro de control de Match. El resitro MCR se usa para controlar el tipo de acción que el timer lleva a cabo cuando el registro contador del timer, TC, se hace igual a uno de los registro de match.	R/W	0	TIMER0 – 0x4000 4014 TIMER1 – 0x4000 8014 TIMER2 – 0x4009 0014 TIMER3 – 0x4009 4014
MR0	Registro de match 0. Mediante la configuración del registro de control de match, MCR, puede conseguirse que cuando el registro contador del timer, TC, tome un valor igual al registro MR0, el registro TC pare de contar, se inicialice a cero y/o se genere una interrupción.	R/W	0	TIMER0 – 0x4000 4018 TIMER1 – 0x4000 8018 TIMER2 – 0x4009 0018 TIMER3 – 0x4009 4018

Cuadro 3: Registros de los timers. Continuación del cuadro 2

Nombre	**Descripción**	**Acceso**	**Reset**	**Direcciones**
MR1	Registro de match 1. Análogo al registro MR0. Ver la descripción de MR0.	R/W	0	TIMER0 – 0x4000 401C TIMER1 – 0x4000 801C TIMER2 – 0x4009 001C TIMER3 – 0x4009 401C
MR2	Registro de match 2. Análogo al registro MR0. Ver la descripción de MR0.	R/W	0	TIMER0 – 0x4000 4020 TIMER1 – 0x4000 8020 TIMER2 – 0x4009 0020 TIMER3 – 0x4009 4020
MR3	Registro de match 3. Análogo al registro MR0. Ver la descripción de MR0.	R/W	0	TIMER0 – 0x4000 4024 TIMER1 – 0x4000 8024 TIMER2 – 0x4009 0024 TIMER3 – 0x4009 4024
CCR	Registro de control de captura. El registro CCR controla cual de los flancos aplicados a las entradas de captura produce la carga de los registros de captura y si se genera una interrupción o no cuando se produce dicha captura.	R/W	0	TIMER0 – 0x4000 4028 TIMER1 – 0x4000 8028 TIMER2 – 0x4009 0028 TIMER3 – 0x4009 4028
CR0	Registro de captura 0. El registro CR0 se carga con el valor del registro TC cuando se produce un evento en las entrada CAPn[0] del timer, donde n es el número del timer (0, 1, 2 ó 3).	RO	0	TIMER0 – 0x4000 402C TIMER1 – 0x4000 802C TIMER2 – 0x4009 002C TIMER3 – 0x4009 402C
CR1	Registro de captura 1. El registro CR1 se carga con el valor del registro TC cuando se produce un evento en las entrada CAPn[1] del timer, donde n es el número del timer (0, 1, 2 ó 3).	RO	0	TIMER0 – 0x4000 4030 TIMER1 – 0x4000 8030 TIMER2 – 0x4009 0030 TIMER3 – 0x4009 4030
EMR	Registro de match externo. El registro EMR controla el comportamiento de los pines externos de match MATn[0], MATn[1], MATn[2], MATn[3], donde n es el número del timer (0, 1, 2 ó 3).	R/W	0	TIMER0 – 0x4000 403C TIMER1 – 0x4000 803C TIMER2 – 0x4009 003C TIMER3 – 0x4009 403C
CTCR	Registro de control de conteo. El registro CTCR selecciona si el timer funciona como temporizador o como contador y, en modo contador, selecciona la señal y el (o los) flanco de conteo.	R/W	0	TIMER0 – 0x4000 4070 TIMER1 – 0x4000 8070 TIMER2 – 0x4009 0070 TIMER3 – 0x4009 4070

4.1. Interrupt Register IR

Cuadro 4: Bits de los registros IR

Bit	Símbolo	Descripción	Reset
0	MR0INT	Flag de interrupción del canal de match 0.	0
1	MR1INT	Flag de interrupción del canal de match 1.	0
2	MR2INT	Flag de interrupción del canal de match 2.	0
3	MR3INT	Flag de interrupción del canal de match 3.	0
4	CR0INT	Flag de interrupción del canal de captura 0.	0
5	CR1INT	Flag de interrupción del canal de captura 1.	0
31:6	-	Reservados. Valor leído indefinido. Sólo deben escribirse ceros.	NA

Todos los bits del registro IR de un timer pueden tanto leerse como escribirse. Cuando se realiza una escritura el registro IR de un timer el comportamiento del registro es algo peculiar:

- Los bits del registro IR correspondientes a bits que están a 0 en el dato que se escribe en el registro no cambian. O dicho al revés, los bits que están a 0 en el dato que se escribe en el registro no afectan al estado del correspondiente bit del registro.
- Los bits del registro IR correspondientes a bits que están a 1 en el dato que se escribe en el registro se ponen a 0, independientemente de su estado anterior. O dicho al revés, los bits que están a 1 en el dato que se escribe en el registro IR fuerzan a 0 el correspondiente bit del registro.

Por ejemplo, si un momento dado el valor del registro IR de un timer es 0x00000013 (0 $\cdots$ 00010011 en binario) y escribimos en el registro el dato 0x00000011 (0 $\cdots$ 00010001 en binario) el valor del registro IR después de realizar la escritura será 0x00000002 (0 $\cdots$ 00000010 en binario).

4.2. Timer Control Register TCR

Cuadro 5: Bits de los registros TCR

Bit	Símbolo	Descripción	Reset
0	CE	Cuando está a 1, los registros TC y PC están habilitados para contar. Cuando está a 0, los registros TC y PC no cuentan.	0
1	CRST	Cuando está a 1, los registros TC y PC se ponen a cero de forma síncrona con el siguiente flanco positivo de PCLK. Los contadores permanecen a cero hasta que este bit se pone a 0 de nuevo.	0
31:2	-	Reservados. Valor leído indefinido. Sólo deben escribirse ceros.	NA

Todos los bits del registro TCR de un timer pueden tanto leerse como escribirse.

4.3. Count Control Register CTCR

Cuadro 6: Bits de los registros CTCR

Bits	Símbolo	Valor	Descripción	Reset
1:0	CTMODE		Este campo determina qué señal y qué flanco(s) incrementan a los registros PC y TC.	00
		00	Modo timer: PC se incrementa con cada flanco de subida de PCLK. Cuando PC coincide con PR, PC se pone a 0 y TC se incrementa, ambas cosas en el siguiente flanco positivo de PCLK.	
		01	Modo contador: TC se incrementa con cada flanco de subida aplicado a la entrada CAP seleccionada por los bits 3 y 2.	
		10	Modo contador: TC se incrementa con cada flanco de bajada aplicado a la entrada CAP seleccionada por los bits 3 y 2.	
		11	Modo contador: TC se incrementa con los flancos de subida y de bajada aplicados a la entrada CAP seleccionada por los bits 3 y 2.	
3:2	CINSEL		Cuando los bits 1 y 0 se este registro no son 00, este campo determina qué señal CAP se usa.	00
		00	CAPx.0 para timer x.	
		01	CAPx.1 para timer x.	
		10	Reservado.	
		11	Reservado.	
31:4	-		Reservados. Valor leído indefinido. Sólo deben escribirse ceros.	NA

Todos los bits del registro CTCR de un timer pueden tanto leerse como escribirse.

4.4. Match control register MCR

Cuadro 7: Bits de los registros MCR

Bit	Símbolo	Valor	Descripción	Reset
0	MR0I	1	Petición de interrupción cuando MR0 coincide con TC.	0
		0	Petición de interrupción deshabilitada.	0
1	MR0R	1	Reset de TC y PC cuando MR0 coincide con TC.	0
		0	Reset deshabilitado.	0
2	MR0S	1	Stop de TC y PC cuando MR0 coincide con TC.	0
		0	Stop deshabilitado.	0
3	MR1I	1	Petición de interrupción cuando MR1 coincide con TC.	0
		0	Petición de interrupción deshabilitada.	0
4	MR1R	1	Reset de TC y PC cuando MR1 coincide con TC.	0
		0	Reset deshabilitado.	0
5	MR1S	1	Stop de TC y PC cuando MR1 coincide con TC.	0
		0	Stop deshabilitado.	0
6	MR2I	1	Petición de interrupción cuando MR2 coincide con TC.	0
		0	Petición de interrupción deshabilitada.	0
7	MR2R	1	Reset de TC y PC cuando MR2 coincide con TC.	0
		0	Reset deshabilitado.	0
8	MR2S	1	Stop de TC y PC cuando MR2 coincide con TC.	0
		0	Stop deshabilitado.	0
9	MR3I	1	Petición de interrupción cuando MR3 coincide con TC.	0
		0	Petición de interrupción deshabilitada.	0
10	MR3R	1	Reset de TC y PC cuando MR3 coincide con TC.	0
		0	Reset deshabilitado.	0
11	MR3S	1	Stop de TC y PC cuando MR3 coincide con TC.	0
		0	Stop deshabilitado.	0
31:12	-		Reservados. Valor leído indefinido. Sólo deben escribirse ceros.	NA

Todos los bits del registro MCR de un timer pueden tanto leerse como escribirse.

4.5. Capture Control Register CCR

Cuadro 8: Bits de los registros CCR

Bit	Símbolo	Valor	Descripción	Reset
0	CAP0RE	1	Captura cuando se aplica un flanco de subida a CAPx.0. El flanco causa que el registro CR0 se cargue con el valor en ese momento de TC.	0
		0	Captura con flanco de subida en CAPx.0 deshabilitada.	0
1	CAP0FE	1	Captura cuando se aplica un flanco de bajada a CAPx.0. El flanco causa que el registro CR0 se cargue con el valor en ese momento de TC.	0
		0	Captura con flanco de bajada en CAPx.0 deshabilitada.	0
2	CAP0I	1	Petición de interrupción cuando se produce el evento de captura por CAPx.0.	0
		0	Interrupción por flancos en CAPx.0 deshabilitada.	0
3	CAP1RE	1	Captura cuando se aplica un flanco de subida a CAPx.1. El flanco causa que el registro CR1 se cargue con el valor en ese momento de TC.	0
		0	Captura con flanco de subida en CAPx.1 deshabilitada.	0
4	CAP1FE	1	Captura cuando se aplica un flanco de bajada a CAPx.1. El flanco causa que el registro CR1 se cargue con el valor en ese momento de TC.	0
		0	Captura con flanco de bajada en CAPx.1 deshabilitada.	0
5	CAP1I	1	Petición de interrupción cuando se produce el evento de captura por CAPx.1.	0
		0	Interrupción por flancos en CAPx.1 deshabilitada.	0
31:6	-		Reservados. Valor leído indefinido. Sólo deben escribirse ceros.	NA

Todos los bits del registro CCR de un timer pueden tanto leerse como escribirse.

4.6. External Match Register EMR

Cuadro 9: Bits de los registros EMR

Bit	Símbolo	Descripción	Reset
0	EM0	External Match 0. Cuando se produce coincidencia entre TC y MR0 este bit puede cambiar de estado, ponerse a 0, ponerse 1 o no cambiar en función del estado de los bits 5:4 de este registro. El bit EM0 puede a su vez actuar sobre el pin externo MATx.0.	0
1	EM1	External Match 1. Cuando se produce coincidencia entre TC y MR1 este bit puede cambiar de estado, ponerse a 0, ponerse 1 o no cambiar en función del estado de los bits 7:6 de este registro. El bit EM1 puede a su vez actuar sobre el pin externo MATx.1.	0
2	EM2	External Match 2. Cuando se produce coincidencia entre TC y MR2 este bit puede cambiar de estado, ponerse a 0, ponerse 1 o no cambiar en función del estado de los bits 9:8 de este registro. El bit EM2 puede a su vez actuar sobre el pin externo MATx.2.	0
3	EM3	External Match 3. Cuando se produce coincidencia entre TC y MR3 este bit puede cambiar de estado, ponerse a 0, ponerse 1 o no cambiar en función del estado de los bits 11:10 de este registro. El bit EM3 puede a su vez actuar sobre el pin externo MATx.3.	0
5:4	EMC0	External Match Control 0. Determina el funcionamiento del External Match 0. Ver el cuadro 10.	00
7:6	EMC1	External Match Control 1. Determina el funcionamiento del External Match 1. Ver el cuadro 10.	00
9:8	EMC2	External Match Control 2. Determina el funcionamiento del External Match 2. Ver el cuadro 10.	00
11:10	EMC3	External Match Control 3. Determina el funcionamiento del External Match 3. Ver el cuadro 10.	00
31:12	-	Reservados. Valor leído indefinido. Sólo deben escribirse ceros.	NA

Todos los bits del registro EMR de un timer pueden tanto leerse como escribirse.

Cuadro 10: Combinaciones de los bits EMC de los registros EMR.

EMR[11:10], EMR[9:8], EMR[7:6] y EMR[5:4]	Función
00	No hacer nada.
01	Poner a cero el correspondiente bit EM y la salida MAT asociada.
10	Poner a uno el correspondiente bit EM y la salida MAT asociada.
11	Invertir el estado del correspondiente bit EM y de la salida MAT asociada.

5. Control del consumo de energía de los periféricos

La mayoría de los microcontroladores poseen modos de funcionamiento especiales que permiten reducir su consumo de energía. La reducción del consumo se consigue inhibiendo la señal de reloj que llega a los distintos subsistemas internos. En un dispositivo digital de tecnología CMOS, como un microcontrolador, un porcentaje muy grande de la corriente consumida de la fuente de alimentación se debe a las transiciones de estado y es proporcional a la frecuencia de la señal de reloj a la que trabaja. Si a una parte del microcontrolador deja de aplicarse la señal de reloj su consumo de energía se reduce a un valor muy pequeño.

El microcontrolador LPC4088 tiene tres modos principales de ahorro de consumo llamados *Sleep*, *Deep Sleep* y *Power-down*. En modo Sleep se inhibe la señal de reloj de la CPU, mientras que los demás subsistemas y periféricos continúan funcionando normalmente. En modo Deep Sleep tanto la CPU como la mayoría de los subsistemas y periféricos dejan de operar. Por último, en modo Power-down, prácticamente todos los bloques internos dejan de recibir señal de reloj. El microcontrolador puede salir de un modo de bajo consumo mediante señales de interrupción o mediante un reset.

Aparte de los modos de ahorro de consumo principales que se han comentado, también puede inhibirse por separado la señal de reloj aplicada a cada uno de los periféricos integrados en el microcontrolador. Esto permite dejar activos sólo los periféricos necesarios para una determinada aplicación y desactivar el resto. El control del consumo de los periféricos se lleva a cabo mediante un registro llamado PCONP (*Power CONtrol for Peripherals*), situado en el bloque *System and Control*. En el cuadro 11 podemos ver que cada uno de los bits del registro PCONP permite controlar el consumo de un periférico del microcontrolador. Si un bit del registro PCONP está a 1, el correspondiente periférico recibirá señal de reloj y funcionará normalmente. Por el contrario, si un bit del registro PCONP está a 0, se inhibirá la llegada de la señal de reloj al periférico asociado y esté permanecerá en modo de bajo consumo. Como vemos en el cuadro 11, hay periféricos que están operativos inmediatamente tras el reset, pues el estado inicial de su bit en el registro PCONP es 1. Sin embargo otros están en modo de bajo consumo después de un reset debido a que el valor de su bit en el registro PCONP es 0 inicialmente. Por ejemplo, de los cuatro timers, sólo los timers 0 y 1 están activos tras el reset, mientras que los timers 2 y 3 están por defecto en bajo consumo.

Cuando un periférico está en modo de bajo consumo debido a que su bit en el registro PCONP está a 0 no es posible utilizarlo ni acceder a ninguno de sus registros. En general, un intento de acceso a un registro de un periférico que está en bajo consumo produce una excepción. Por tanto, en caso de que queramos usar un periférico que por defecto tiene a 0 su bit en PCONP, lo primero que tendremos que hacer antes de intentar acceder a él será cambiar a 1 el estado de dicho bit. Para ello, primero leeremos el valor actual de PCONP, realizaremos una operación lógica OR bit a bit con una máscara que tendrá a 1 sólo el bit que queremos activar y escribiremos el resultado de nuevo en el registro. Esto asegurará que el bit adecuado se pone a 1 sin que los demás sufran ningún cambio. Si no lo hacemos así, otros periféricos distintos a aquel que queremos activar podrían cambiar su modo de consumo. Si, por ejemplo, en nuestro programa queremos usar el timer 2, tendremos que activarlo mediante una línea como esta:

```
LPC_SC->PCONP |= 1 << 22;
```

El símbolo `LPC_SC` es el puntero al bloque de registros del *System and Control* en el que se encuentra el registro PCONP (este puntero se define en el fichero `LPC407x_8x_177x_8x.h`).

Cuadro 11: Bits del registro PCONP

Bit	**Símbolo**	**Descripción**	**Reset**
0	PCLCD	Bit de control de consumo del controlador de LCD.	0
1	PCTIM0	Bit de control de consumo del timer 0.	1
2	PCTIM1	Bit de control de consumo del timer 1.	1
3	PCUART0	Bit de control de consumo de la UART 0.	1
4	PCUART1	Bit de control de consumo de la UART 1.	1
5	PCPWM0	Bit de control de consumo del generador PWM 0.	0
6	PCPWM1	Bit de control de consumo del generador PWM 1.	0
7	PCI2C0	Bit de control de consumo de la interfaz I2C 0.	1
8	PCUART4	Bit de control de consumo de la UART 4.	0
9	PCRTC	Bit de control de consumo del RTC y registrador de eventos.	1
10	PCSSP1	Bit de control de consumo del interfaz SSP 1.	0
11	PCEMC	Bit de control de consumo del controlador de memorias externas.	0
12	PCADC	Bit de control de consumo del convertidor A/D.	0
13	PCCAN1	Bit de control de consumo del interfaz CAN 1.	0
14	PCCAN2	Bit de control de consumo del interfaz CAN 2.	0
15	PCGPIO	Bit de control de consumo de los GPIO y sus interrupciones.	1
16	PCSPIFI	Bit de control de consumo del controlador de Flash SPIFI.	0
17	PCMCPWM	Bit de control de consumo del generador PWM para motores.	0
18	PCQEI	Bit de consumo del decodificador de encoder en cuadratura.	0
19	PCI2C1	Bit de control de consumo del interfaz I2C 1.	1
20	PCSSP2	Bit de control de consumo del interfaz SSP 2.	0
21	PCSSP0	Bit de control de consumo del interfaz SSP 0.	0
22	PCTIM2	Bit de control de consumo del timer 2.	0
23	PCTIM3	Bit de control de consumo del timer 3.	0
24	PCUART2	Bit de control de consumo de la UART 2.	0
25	PCUART3	Bit de control de consumo de la UART 3.	0
26	PCI2C2	Bit de control de consumo del interfaz I2C 2.	1
27	PCI2S	Bit de control de consumo del interfaz I2S.	0
28	PCSDC	Bit de control de consumo de la interfaz de tarjetas SD.	0
29	PCGPDMA	Bit de control de consumo del controlador de DMA.	0
30	PCENET	Bit de control de consumo del interfaz Ethernet.	0
31	PCUSB	Bit de control de consumo del interfaz USB.	0

6. Uso de los timers para generar retardos de tiempo

En muchas aplicaciones de los microcontroladores es necesario conseguir que entre los instantes en los que queremos que el programa lleve a cabo diferentes acciones transcurra un intervalo de tiempo dado. Por ejemplo, recordemos el primer programa que hicimos funcionar en el microcontrolador. Se trataba de un programa muy sencillo que se limitaba a hacer parpadear un LED cambiando alternativamente entre 0 y 1 el estado de un pin previamente configurado como salida. Para que el parpadeo fuese visible fue necesario insertar retardos de tiempo entre los instantes en los que la salida se ponía a 0 y a 1. En ese primer programa, los retardos de tiempo se realizaron haciendo que la CPU perdiese tiempo ejecutando bucles que, aunque estaban vacíos, obligaban a incrementar y verificar el índice de los mismos muchísimas veces (un millón de veces).

Sin embargo, esta técnica de realizar retardos no es muy aconsejable. La primera razón es que no es fácil ajustar de forma precisa el tiempo de retardo, ya que ello requeriría conocer cuánto tiempo tarda la CPU en ejecutar cada una de las iteraciones del bucle y esto depende del compilador usado y de los ajustes concretos del mismo. En el programa de parpadeo, el tiempo que el LED permanece encendido y apagado no es crítico mientras el parpadeo sea visible, así que el número de iteraciones de los bucles puede ajustarse por tanteo hasta que el parpadeo tenga una cadencia más o menos buena para nuestro gusto. Pero, en otras aplicaciones, el tiempo que debe trascurrir entre dos acciones en el programa tiene que tener una exactitud de milisegundos o microsegundos así que este método deja de ser válido. Otra razón por la que necesitamos disponer de otro método para realizar retardos de tiempo es permitir a la CPU aprovechar el tiempo de retardo para llevar a cabo otros procesos. Usando un bucle para realizar el retardo, la CPU se mantiene ocupada ejecutando el bucle y no puede usar este tiempo para nada útil.

En esta sección se verá una forma sencilla de utilizar un timer para generar un retardo de tiempo de una duración precisa. Realmente, la forma más conveniente de realizar este tipo de funciones es usar la posibilidad de los timers de generar interrupciones. Esto permite a la CPU realizar otras tareas mientras el timer lleva a cabo la temporización. Si embargo, puesto que aún no se ha estudiado el sistema de interrupciones del microcontrolador, en esta práctica no se aprovechará esta posibilidad. En su lugar, para detectar el momento en el que un timer señala la finalización del periodo de tiempo programado, se sondeará por software uno de sus flags de solicitud de interrupción. Esto impedirá a la CPU usar el tiempo de retardo para llevar a cabo otro trabajo, pero al menos conseguiremos que los retardos tengan una duración bastante exacta.

En la figura 2 se muestra un diagrama de bloques que podemos emplear para entender el principio de funcionamiento de un timer cuando se usa para generar un retardo de tiempo. Se muestra el caso del uso del timer 0 y su registro de match 0 pero la misma idea sería aplicable a cualquiera de los otros timers y registros de match. Hay que advertir que este diagrama tiene un carácter puramente didáctico y que algunos de los elementos que aparecen en él, tales como las tres puertas AND, sólo son un recurso para explicar el funcionamiento del timer y no responden a ninguna información proporcionada por NXP acerca de la circuitería interna de los timers.

Suponemos que el campo CTMODE del registro CTCR tiene el valor 0, que es su valor tras el reset. Esto implica que la señal que hace que el registro contador del timer se incremente se obtiene dividiendo la frecuencia de la señal de reloj de periféricos, PCLK. Como puede verse en la figura 2, la señal PCLK se aplica a la entrada de un divisor de frecuencia

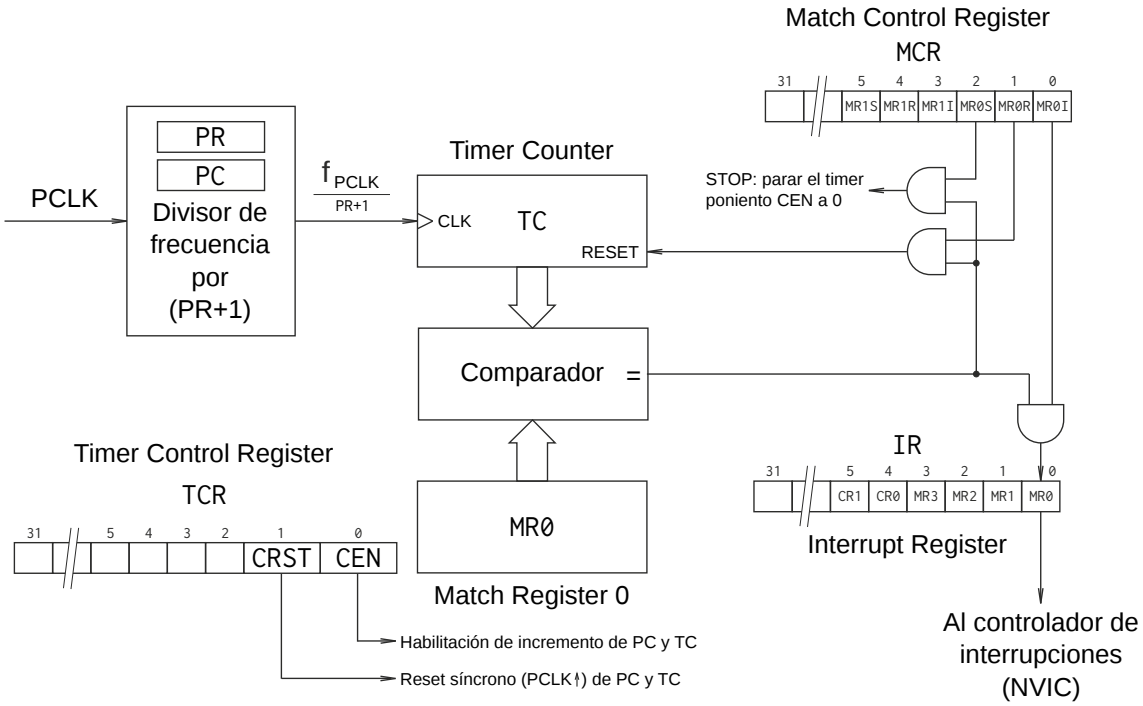

Figura 2: Diagrama de bloques del timer 0 usándose para generar un retardo de tiempo.

constituido por el Contador de Preescala, PC, y el Registro de Preescala, PR. Por defecto, la señal de reloj de periféricos PCLK tiene una frecuencia igual a la frecuencia de reloj de la CPU (CCLK) dividida entre 2. Como la frecuencia de reloj a la que hacemos funcionar la CPU es de 120 MHz, la frecuencia del reloj PCLK es de 60 MHz. El divisor de frecuencia divide adicionalmente la frecuencia del reloj PCLK por un factor (PR+1), siendo PR el valor del registro de preescala, PR. La señal de salida del divisor de frecuencia se aplica al registro contador del timer, TC. Este se incrementará a razón de $f_{PCLK}/(PR+1)$ veces por segundo, donde f_{PCLK} es la frecuencia del reloj PCLK (60 MHz en nuestro caso).

El valor del registro contador del timer se compara constantemente con los valores de los registros de match MR0, MR1, MR2 y MR3 del propio timer. En la figura 2 sólo se muestra el registro MR0. Cuando se produce una coincidencia de valores entre el registro contador del timer, TC, y el registro de match MR0, el siguiente pulso de reloj que llegue a TC puede desembocar en distintas acciones sobre el timer, dependiendo del valor del registro MCR. Si el bit MR0I (bit 0) del registro MCR está a 1, se producirá una petición de interrupción. Si el bit MR0R (bit 1) del registro MCR está a 1, el registro contador del timer se pondrá a cero inmediatamente. Si el bit MR0S (bit 2) del registro MCR está a 1, el registro contador del timer parará de contar. Si varios de los bits del registro MCR mencionados están a 1 se realizarán al mismo tiempo todas las acciones asociadas a ellos. Como se ha comentado previamente, el mecanismo de funcionamiento en relación con los otros tres registros de match es análogo. Cada registro de match tiene asociado un conjunto de tres bits en el registro MCR. El registro MR0 tiene los bits 0, 1 y 2 que hemos descrito antes, el MR1 tiene los bits 3, 4 y 5, el MR2 los bits 6, 7 y 8 el MR3 los bits 9, 10 y 11.

Hemos dicho que si se pone a 1 el bit MRxI (donde x es el número del registro de match asociado), cuando el valor del registro contador del timer, TC, y el valor del registro de match correspondiente coinciden, el siguiente pulso que sale del bloque de preescala hace que el timer solicite una interrupcion a la CPU. Realmente, lo primero que ocurre es que

el bit de petición de interrupción asociado al registro de match situado en el registro de interrupción del timer, IR, se pone a 1. En caso de que el sistema de interrupciones del microcontrolador se haya programado convenientemente, a su vez esto generará una señal de interrupción para la CPU (Esto se verá en una práctica posterior).

Pero, aunque no usemos la posibilidad de generar una interrupción, podemos *sondear* los flags de petición de interrupción del registro IR de un timer, es decir, podemos consultar su estado por software (en nuestro programa) para saber en cualquier momento cuál es el estado de cualquiera de ellos. Esto proporciona una forma básica de usar un timer para generar un retardo de tiempo bastante preciso sin necesidad de conocer todavía cómo manejar interrupciones.

Los pasos para generar un retardo de tiempo con un timer serían entonces:

1. Parar el timer poniendo a cero el bit CEN (bit 0) de su registro TCR. De esta forma nos aseguramos de que el registro contador, TC, del timer no aumenta mientras realizamos los siguientes pasos.
2. Poner a cero los registros PC y TC. Con esto nos aseguramos de que cuando pongamos en marcha el timer el contero comienza desde 0. Aunque el valor de estos registros tras el reset sea 0, no podemos estar seguros de que en el momento de querer usar el timer aún tengan este valor, ya que el timer ya podría haber sido usado previamente en otra parte del programa.
3. Programar el registro de preescala PR y uno de los cuatro registros de match de forma que

$$\frac{(PR+1)\cdot(MRx+1)}{f_{PCLK}} = \textit{tiempo de retardo requerido en segundos}$$

   pero evitando programar el valor 0 en MRx.
4. Programar el registro MCR activando el bit de interrupción asociado al registro de match seleccionado en el paso anterior.
5. En el registro de petición de interrupción del timer, IR, poner a cero el bit de petición de interrupción correspondiente al registro de match seleccionado en el paso 3. Para conseguirlo, escribir en el registro IR un dato que tenga a 1 el bit que se desea borrar. Por ejemplo, si deseamos borrar el bit 0 de IR, escribiremos en el registro el valor 1. Este paso es necesario para asegurarnos de que el bit está a 0 antes de poner en marcha el timer, ya que el evento que marcará el final del tiempo de retardo será el cambio de 0 a 1 del bit. Si el bit está ya 1 cuando pongamos en marcha el timer no será producirá ningún cambio en su estado al final del periodo de tiempo.
6. Poner en marcha el timer poniendo a 1 el bit CEN del registro TCR.
7. Esperar a que en el registro IR se active el bit de petición de interrupción asociado al registro de match seleccionado en el paso 3. Esto se hará mediante un bucle que se repetirá una y otra vez mientras el estado del bit sea 0. Cuando el bit cambie a 1, saldremos del bucle y el programa seguirá adelante.

El punto 3 indica que cualquier combinación de los registros PR y MRx que cumpla la igualdad es válida, siempre que MRx no sea 0. Supongamos que queremos generar un retardo de 4 milisegundos usando uno de los timers y que vamos a emplear su registro de match MR0. Una forma de elegir los valores de PR y MRx sería PR = 59999 y MR0 = 3.

El funcionamiento es el siguiente: un valor 59999 en el registro PR hará que la frecuencia del reloj de periféricos, que recordemos es habitualmente de 60 MHz, se divida entre 60000 antes de ser aplicada al registro contador TC. Por tanto, partiendo del valor 0 con el que ha sido cargado al principio, TC se incrementará a un ritmo de $(60 \times 10^6)/(60 \times 10^3) = 1000$ veces por segundo, es decir, se incrementará cada milisegundo. Cuando el valor de TC alcance el valor 3, el siguiente pulso de reloj que salga del bloque de preescala hará que el bit 0 del registro IR se ponga a 1. El registro TC habrá recorrido los valores 0, 1, 2 y 3, permaneciendo un milisegundo en cada valor, lo que dará un total de 4 milisegundos entre la puesta en marcha del timer en el paso 6 y la detección de la activación del bit 0 de IR en el paso 7. Otra combinación válida sería, por ejemplo, PR = 59 y MR0 = 3999. En este caso el valor del registro PR provoca que el registro TC se incremente una vez cada microsegundo y el valor del registro MR0 dará un total de 4000 μs = 4 ms entre la puesta en marcha del timer y la activación del bit 0 de IR. El timer no funciona bien si el valor del registro de match es 0, así que evitaremos darle este valor.

En realidad, el tiempo requerido para todo el proceso será superior al requerido porque los pasos de 1 a 6 consumen también un tiempo. No obstante, a la frecuencia de funcionamiento máxima de la CPU de nuestro microcontrolador (120 MHz) los pasos 1 a 6 no llegarán a consumir ni un microsegundo, de forma que si, comparativamente, el tiempo de retardo requerido es mucho más grande, se cometerá un error relativo muy pequeño en el retardo de tiempo conseguido respecto al buscado.

7. Manejo de errores

Durante la ejecución de un programa pueden producirse situaciones anómalas que requieren llevar a cabo una acción especial. Por una parte, pueden surgir circunstancias que, aunque sean indeseables o infrecuentes, forman parte de los escenarios legítimos que pueden darse durante la ejecución. Por ejemplo, supongamos que en un sistema que guarda datos en una tarjeta de memoria SD, ésta se extrae sin querer. La próxima vez que se ejecute la función del programa que graba los datos en la tarjeta la operación no podrá realizarse. A pesar de ser una circunstancia anómala el programa puede continuar si de alguna forma se avisa al usuario de que debe volver a insertar la tarjeta. Por tanto, es lógico diseñar la función que graba los datos en la tarjeta de manera que indique si la operación se ha podido completar o no. Una forma sería hacer que la función retorne un booleano que indique éxito o fracaso:

```
bool_t grabar_datos(uint8_t *ptr_datos, uint32_t numero_bytes);
```

Hemos supuesto que la función recibe como argumentos un puntero a los datos y el número de bytes que queremos guardar.

El inconveniente de diseñar así la función es que en caso de error no se puede conocer la causa concreta (tarjeta no presente, llena, protegida contra escritura, etc.). Entonces, una mejora sería retornar un entero cuyo valor indique qué tipo de error se ha producido. Típicamente, en este caso, si no hay error se retorna 0 y en caso contrario un valor distinto de 0, diferente para cada una de las posibles causas de error:

```
int32_t grabar_datos(uint8_t *ptr_datos, uint32_t numero_bytes);
```

Analizando el valor retornado, puede conocerse si ha habido algún problema y en ese caso tomar alguna medida para intentar remediarlo, por ejemplo, pedir al usuario que reinserte la

tarjeta. Una vez subsanado el problema puede reintentarse la operación de escritura:

```
do
{
    err = grabar_datos(datos, 10);
    if (err == TARJETA_NO_PRESENTE)
    {
        /* Pedir al usuario que inserte la tarjeta y continuar cuando lo haga.
         */
    }
    else if ... /* Otras causas de error. */

}
while (err != NO_ERROR);
```

Hay que notar que en C es lícito ignorar el valor retornado por una función, así que no hay garantías de que la información de error retornada por una función sea analizada por la función llamadora. Esto no es lo correcto, pero es fácil olvidarlo o confiar en que no habrá errores.

Otra forma de devolver información de error desde una función es emplear un argumento adicional de tipo puntero:

```
void grabar_datos(uint8_t *ptr_datos, uint32_t numero_datos, int32_t *error);
```

En este caso, un ejemplo de llamada sería

```
grabar_datos(datos, 10, &error);
```

Este método hace más difícil obviar la información de error, ya que no podemos llamar a la función sin pasar un puntero a una variable donde guardar el código de error, pero tampoco garantiza que su valor sea analizado, aunque lo correcto sea hacerlo siempre.

Este tipo de errores de los que, en principio, debería ser posible recuperarse suelen llamarse excepciones.

Por otra parte, independientemente del método que hayamos elegido para devolver información de error desde `grabar_datos`, supongamos que se la llama pasando un puntero nulo a través del argumento `ptr_datos`. Si la función `grabar_datos` detecta que `ptr_datos` es nulo no hay ninguna razón legítima por la que esto pueda suceder. La función nunca debería haber sido llamada de esta forma por ninguna circunstancia, ya que se está indicando que a la función que lea los datos a grabar a partir de una dirección de memoria que, por definición, no es válida. Si esto ocurre, seguro que se debe a que el programador ha cometido un error en alguna parte del código del programa que ha terminado causando esta llamada sin sentido. En este caso no tiene ninguna utilidad que la función retorne indicando error porque no hay nada que la función llamadora pueda hacer para recuperarse del problema. Intentar retornar y proseguir la ejecución del programa sólo es demorar la detección del problema. Si, como se ha dicho antes, el programador no se ha preocupado de analizar la información de error devuelta por la función confiando en que nada puede ir mal, el error puede pasar desapercibido y manifestarse mucho más tarde, dificultando su detección. Un error como este sólo puede subsanarse corrigiendo el código del programa y para ello es mejor que en cuanto se detecte aborte inmediatamente la ejecución del programa generando, si es posible, un mensaje que dé al programador pistas sobre lo que ha pasado.

7.1. Aserciones

El fichero de cabecera estándar `assert.h` incluye la macro `assert` que está pensada para detectar errores y abandonar la ejecución del programa. `assert` evalúa una condición y si resulta ser falsa imprime un mensaje de error en la pantalla y sale al sistema operativo.

Sin embargo, la función `assert` estándar no siempre es útil en un programa para un sistema embebido, ya que en muchos casos no hay pantalla donde imprimir ni un sistema operativo al que devolver el control. Por ello, lo más habitual es crear una función o macro para realizar aserciones adaptada a los medios de que se disponga para informar del error. Además, como no hay sistema operativo al que volver, en lugar de salir del programa lo habitual es dejar a éste encerrado en un bucle infinito.

En nuestro caso, usaremos la macro `ASSERT` que se define en el fichero `error.h` que formará parte de nuestros proyectos. Nuestro `ASSERT` admite dos parámetros

```
ASSERT(expr, mensaje)
```

`expr` es la expresión cuya veracidad queremos comprobar y `mensaje` es el mensaje de error que queremos generar en caso de que la expresión sea falsa. El texto del mensaje también ayuda a documentar el programa.

La macro `ASSERT` está definida así en `error.h`

```
#if HABILITAR_ASSERT != 0
#define ASSERT(expr, mensaje)\
if (expr) {} else { parar_con_error(__FILE__, __FUNCTION__, __LINE__, mensaje); }
#else
#define ASSERT(expr, mensaje)
#endif
```

La macro se expande de manera distinta según el símbolo `HABILITAR_ASSERT` se haya definido con un valor cero o distinto de cero. Esto se hace mediante un `define` en el mismo fichero `error.h`

```
#define HABILITAR_ASSERT 1
```

Si el valor de `HABILITAR_ASSERT` se define distinto de cero la macro `ASSERT` se expande a la línea

```
if (expr) {} else { parar_con_error(__FILE__, __FUNCTION__, __LINE__, mensaje); }
```

que evalúa la expresión y, si es falsa, llama a la función `parar_con_error` de `error.c`, la cual imprimirá en la pantalla LCD el mensaje de error junto con el nombre del fichero fuente, el nombre de la función y la línea donde ha fallado la aserción. La información sobre el nombre del fichero fuente, la función y el número de línea se obtiene mediante los símbolos `__FILE__`, `__FUNCTION__` y `__LINE__` que proporcionan los compiladores de C.

Puede ser que nos preguntemos por qué `ASSERT` no se ha definido de esta otra forma, más sencilla

```
if (!(expr)) { parar_con_error(__FILE__, __FUNCTION__, __LINE__, mensaje); }
```

La razón es evitar que en caso de que `ASSERT` se use detrás de un `if` al que le sigue un `else`, el `else` quede asociado al `if` que genera la expansión de `ASSERT` y no al `if` original.

Si el valor de `HABILITAR_ASSERT` se define como cero, la macro `ASSERT` se expande a una línea vacía, lo que implica desactivar todos los `ASSERT` que se hayan colocado en el programa. Esto puede hacerse una vez que hayamos comprobado concienzudamente el programa y estemos seguros de que no contiene ningún error lógico. La ventaja de hacerlo así es eliminar el tiempo de ejecución extra que requiere comprobar cada una de las condiciones, con lo que puede incrementarse la velocidad de ejecución del programa. De todas formas, se aconsejable dejar siempre activas las aserciones, ya que no puede descartarse que durante el funcionamiento del sistema embebido (incluso una vez entregado al cliente) surja un error que no se ha detectado en la fase de pruebas. Si las aserciones están activas será más fácil localizar y resolver el problema.

Supongamos que aplicamos la macro `ASSERT` a la función de grabación de datos en la tarjeta de memoria. Destacando sólo los elementos importantes, quedaría

```
#include "error.h"

...

int32_t grabar_datos(uint8_t *ptr_datos, uint32_t numero_bytes)
{
    ASSERT(ptr_datos != NULL, "Error: el puntero ptr_datos es nulo.");

    /* Resto de la función grabar_datos
     */

    ...

}
```

Vemos la forma de usar la macro `ASSERT`. El primer argumento es la condición que queremos verificar, en este caso que el puntero `ptr_datos` que nos han pasado no es un puntero nulo. Si la condición es cierta la ejecución prosigue normalmente. Sin embargo, si la condición no se cumple se imprimirá en la pantalla LCD el mensaje `Error: el puntero ptr_datos es nulo.`, junto con información que indicará la ubicación del `ASSERT` que ha fallado, y el programa se detendrá. Esto se hace mediante una llamada a la función `parar_con_error` definida en el fichero fuente `error.c` (así que es necesario que este fichero forme parte del proyecto).

En el fichero `error.h` también se define la macro `ERROR`. Esta macro no comprueba ninguna condición sino que para el programa incondicionalmente imprimiendo el mensaje de error suministrado e información sobre la ubicación del error.

La función `parar_con_error` puede adaptarse a las posibilidades de comunicación con el exterior del sistema embebido que estamos desarrollando. Por ejemplo, si el sistema no posee una pantalla donde imprimir mensajes de error pero sí un puerto de comunicación con el exterior, podemos enviar los mensajes de error a un PC que puede usarse para el diagnóstico. En última instancia, podemos situar un punto de ruptura en la función `parar_con_error` y usar el depurador para averiguar qué ha ocurrido.

8. Biblioteca de funciones para manejar los timers

De manera análoga a como hicimos con los GPIO en la práctica anterior, crearemos un conjunto básico de funciones en C para facilitar el uso de los timers desde nuestros programas. Escribiremos estás funciones en un fichero fuente llamado `timer_lpc40xx.c` que tendrá asociado un fichero de cabecera llamado `timer_lpc40xx.h`. Las funciones que compondrán las biblioteca será:

- `void timer_inicializar(LPC_TIM_TypeDef *timer_regs)`
- `void timer_retardo_ms(LPC_TIM_TypeDef *timer_regs, uint32_t retardo_en_ms)`
- `void timer_retardo_us(LPC_TIM_TypeDef *timer_regs, uint32_t retardo_en_us)`
- `void timer_iniciar_ciclos_ms(LPC_TIM_TypeDef *timer_regs, uint32_t periodo_en_ms)`
- `void timer_iniciar_ciclos_us(LPC_TIM_TypeDef *timer_regs, uint32_t periodo_en_us)`
- `void timer_esperar_fin_ciclo(LPC_TIM_TypeDef *timer_regs)`
- `void timer_iniciar_conteo_ms(LPC_TIM_TypeDef *timer_regs)`
- `void timer_iniciar_conteo_us(LPC_TIM_TypeDef *timer_regs)`
- `uint32_t timer_leer(LPC_TIM_TypeDef *timer_regs)`

A continuación se muestra el listado de las funciones `timer_inicializar` y `timer_retardo_ms`. El resto de funciones se desarrollarán como parte de los ejercicios de la práctica.

Para asegurarnos de que un timer está activo antes de usarlo, utilizaremos a la función `timer_inicializar` cuyo código se muestra a continuación.

El argumento `timer_regs` es un puntero al bloque de registros del timer que queremos inicializar. Podemos usar los punteros `LPC_TIMx` declarados en el fichero `LPC407x_8x_177x_8x.h` o los símbolos equivalentes `TIMERx` definidos en `timer_lpc40xx.h`.

La función debe comprobar que el argumento `timer_regs` es correcto, es decir, que señala al bloque de registros de uno de los cuatro timers. En caso contrario, la función debe abortar la ejecución del programa. Para comprobar `timer_regs`, usa la macro `ASSERT` definida en el fichero `error.h`.

Listado 1: Función para inicializar un timer

```
/*****************************************************************************//**
 * \brief       Usar un timer para generar un retardo del número de milisegundos
 *              indicado. La función no retorna hasta que transcurre este
 *              tiempo.
 *
 * \param[in]   timer_regs      puntero al bloque de registros del timer.
 * \param[in]   retado_en_ms    número de milisegundos de duración del retardo.
 */
```

```
void timer_inicializar(LPC_TIM_TypeDef *timer_regs)
{
    if (timer_regs == TIMER0)
    {
        LPC_SC->PCONP |= 1 << 1;
    }
    else if (timer_regs == TIMER1)
    {
        LPC_SC->PCONP |= 1 << 2;
    }
    else if (timer_regs == TIMER2)
    {
        LPC_SC->PCONP |= 1 << 22;
    }
    else if (timer_regs == TIMER3)
    {
        LPC_SC->PCONP |= 1 << 23;
    }
    else ERROR("timer_regs incorrecto");
}
```

En el listado 2 se muestra una función que usa las ideas del apartado **??** para generar un retardo de tiempo de tantos milisegundos como indica el argumento `retardo_en_ms`. Como se indicó anteriormente, debemos programar el registro de preescala y un registro de match con valores tales que $(PR + 1)(MRx + 1)/f_{PCLK}$ sea igual al número de segundos de retardo que queremos. Aquí se ha optado por programar el registro PR con un valor que hace que el registro contador TC se incremente a una frecuencia de 1 MHz, es decir cada μs, y el registro de match MR0 con el número de milisegundos que indica el argumento `retardo_en_ms` multiplicado por 1000. El valor para el registro PR se obtiene dividiendo la frecuencia del reloj de periféricos entre 59. En lugar de escribir directamente el valor 59 hemos aprovechado la variable global `PeripheralClock` que se define en uno de los ficheros de inicialización del microcontrolador. Esta variable nos indica la frecuencia del reloj PCLK (puede comprobarse que con $f_{PCLK} = 60$ MHz, la expresión (PeripheralClock/1000000 – 1) tiene un valor de $60 \times 10^6/1000000 - 1 = 59$ como se requiere). La ventaja de usar esta variable es que, en caso de que la frecuencia del reloj de periféricos cambie, no será necesario hacer ningún cambio en esta línea de programa. Puede pensarse que sería más lógico usar un valor de preescala que haga que TC se incremente cada milisegundo y cargar MR0 con el número de milisegundos de retardo requerido menos uno, pero en caso de que el argumento `retardo_en_ms` sea 1 esto produciría el valor 0 en TC que queremos evitar.

Como se ha mencionado en la sección 5, los timers 2 y 3 están en modo de bajo consumo tras el reset así que antes de usar uno de estos dos timers es necesario activarlo en el registro PCONP.

Listado 2: Función de retardo en milisegundos usando un timer

```
/*****************************************************************************//**
 * \brief       Usar un timer para generar un retardo del número de milisegundos
 *              indicado. La función no retorna hasta que transcurre este
 *              tiempo.
 *
 * \param[in]   timer_regs      puntero al bloque de registros del timer.
 * \param[in]   retado_en_ms    número de milisegundos de duración del retardo.
 */
void timer_retardo_ms(LPC_TIM_TypeDef *timer_regs,
```

```
                    uint32_t retardo_en_ms)
{
    ASSERT(timer_regs == TIMER0 ||
           timer_regs == TIMER1 ||
           timer_regs == TIMER2 ||
           timer_regs == TIMER3,
           "timer_regs incorrecto");

    if (retardo_en_ms == 0) return;

    timer_regs->TCR = 0;
    timer_regs->TC = 0;
    timer_regs->PC = 0;
    timer_regs->PR = PeripheralClock/1000000 - 1;
    timer_regs->MR0 = 1000*retardo_en_ms - 1;
    timer_regs->MCR |= 7;
    timer_regs->IR = 1;
    timer_regs->TCR = 1;
    while ((timer_regs->IR & 1) == 0) {}
}
```

9. Pantalla LCD

En los ejercicios propuestos en esta práctica y en otras posteriores tendremos que acceder a la pantalla LCD con la que cuenta el sistema de desarrollo. Se trata de una pantalla de tipo TFT de 4.3" con una resolución de 480 pixels en horizontal y 272 pixels en vertical. Normalmente, usaremos la pantalla en el modo de 16 bits por pixel, que permite visualizar 65536 colores distintos. En la figura 3 tenemos una referencia de los ejes y coordenadas de la pantalla.

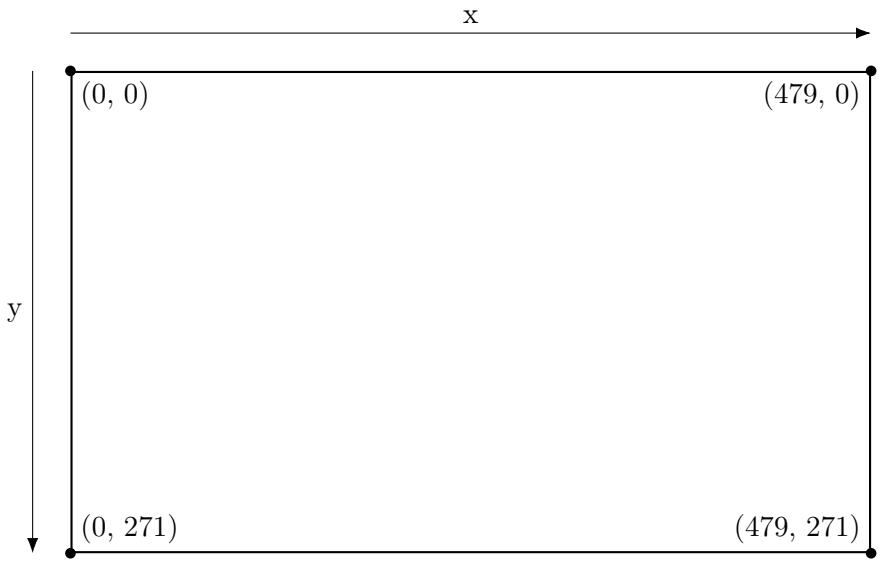

Figura 3: Coordenadas de la pantalla LCD.

9.1. Funciones para manejar la pantalla LCD

Para facilitar el manejo de la pantalla LCD aprovecharemos las funciones disponibles en el fichero fuente `glcd.c` que se suministrará como parte de los proyectos de los que partiremos para realizar los ejercicios. El fichero de cabecera correspondiente se llama `glcd.h`. A continuación, se describen algunas de las funciones incluidas en `glcd.c`.

- `void glcd_inicializar(void)`

  Inicializa la pantalla LCD. Esta función debe ser llamada antes de llevar a cabo cualquier otra operación sobre el LCD.

- `void glcd_borrar(uint16_t color)`

  Borra la pantalla LCD rellenándola con el color indicado por el argumento `color`.

- `int32_t glcd_printf(const char *fmt, ...)`

  Esta función es análoga a la función `printf` de la biblioteca estándar. El argumento `fmt` es un puntero a la cadena de formato. En caso de que la cadena de formato incluya especificadores de formato, tales como `%d`, `%f`, etc., detrás de la cadena de formato se incluirán las expresiones cuyos valores sustituirán a los especificadores en la salida impresa. Igual que `printf`, `glcd_printf` retorna el número de caracteres impresos y –1 en caso de error.

- `void glcd_color_texto(uint16_t color)`

  Fija el color del texto para las subsiguientes llamadas a `glcd_printf` al color indicado por el argumento `color`.

- `void glcd_fondo_texto(uint16_t color_fondo)`

  Fija el color del fondo del texto para las subsiguientes llamadas a `glcd_printf` al color indicado por el argumento `color_fondo`.

- `void glcd_xy_texto(int32_t x, int32_t y)`

  Indica las coordenadas (x, y) de pantalla a partir de las cuales se producirá la impresión del texto en la siguiente llamada a `glcd_printf`.

- `void glcd_font(uint32_t font)`

  Fija el tamaño de los caracteres para las subsiguientes llamadas a `glcd_printf`. Existen tres tamaños de caracteres que se indican en función de la celdilla en la que están inscritos: de 8×16 pixels, de 12×24 pixels y de 16×32 pixels. En el fichero `glcd.h` se definen tres constantes llamadas `FONT8X16`, `FONT12X24` y `FONT16X24` para seleccionar cada uno de estos fonts. Dentro de cada font, la anchura de los caracteres es siempre constante.

- ```
  int32_t glcd_xprintf(int32_t x,
                       int32_t y,
                       uint16_t color,
                       uint16_t color_fondo,
                       uint32_t font,
                       const char *fmt,
                       ...)
  ```
  ```

Realiza la misma función que `glcd_printf` pero permite seleccionar las coordenadas, los colores del texto y del fondo y el tamaño de los caracteres que se emplearán en la salida de texto sin necesidad de llamar previamente a las funciones `glcd_color_texto`, `glcd_fondo_texto` y `glcd_xy_texto` y `glcd_font`. Las coordenadas, colores y font seleccionados sólo tienen efecto en cada llamada a la función `glcd_xprintf`. La posición actual, colores y font seleccionados actualmente para `glcd_printf` no cambian.

- ```
  void glcd_punto(int32_t x,
                  int32_t y,
                  uint16_t color)
  ```

  Dibuja un punto (pixel) en las coordenadas de pantalla (`x`, `y`) del color indicado por `color`.

- ```
  void glcd_linea(int32_t x0,
                  int32_t y0,
                  int32_t x1,
                  int32_t y1,
                  uint16_t color)
  ```

  Dibuja un segmento de línea recta que tiene por extremos los puntos (`x0`, `y0`) y (`x1`, `y1`) usando el color indicado por `color`. Si uno o ambos extremos del segmento está fuera de la pantalla, se dibuja la porción del segmento contenida en ella.

- ```
  void glcd_rectangulo(int32_t x0,
                       int32_t y0,
                       int32_t x1,
                       int32_t y1,
                       uint16_t color)
  ```

  Dibuja el contorno de un rectángulo que tiene como una de sus diagonales aquella cuyos extremos están situados en las coordenadas (`x0`, `y0`) y (`x1`, `y1`). El color del rectángulo se indica con el argumento `color`. Si (`x0`, `y0`) y/o (`x1`, `y1`) están fuera de la pantalla, se dibujan las porciones de los lados del rectángulo contenidas dentro de ella.

- ```
  void glcd_rectangulo_relleno(int32_t x0,
                               int32_t y0,
                               int32_t x1,
                               int32_t y1,
                               uint16_t color)
  ```

  Dibuja un rectángulo con los vértices de una diagonal situados en las coordenadas (`x0`, `y0`) y (`x1`, `y1`) y usando el color indicado por `color` tanto para su contorno como para su relleno interior. Si (`x0`, `y0`) y/o (`x1`, `y1`) están fuera de la pantalla, se dibuja la porción del rectángulo que está contenida dentro de ella.

- ```
  void glcd_circunferencia(int32_t xc,
                           int32_t yc,
                           int32_t radio,
                           uint16_t color)
  ```

  Dibuja una circunferencia con el centro situado en las coordenadas de pantalla (`xc`, `yc`) y un radio en pixels dado por el argumento `radio`. El argumento `color` determina
  ```

el color de la circunferencia. Es posible especificar una circunferencia que cruce los límites de la pantalla. En este caso se dibuja la parte de la circunferencia que está contenida dentro de la misma.

- ```
  void glcd_circulo(int32_t xc,
                    int32_t yc,
                    int32_t radio,
                    uint16_t color)
  ```

  Dibuja un círculo con el centro situado en las coordenadas de pantalla (`xc`, `yc`) y un radio en pixels dado por el argumento `radio`. El argumento `color` determina tanto el color del contorno exterior como del relleno interior. Se dibuja la porción del círculo que está contenida dentro de los límites de la pantalla.

9.2. Constantes para designar colores

En `glcd.h` se definen constantes con los nombres de varios colores tanto en inglés (RED, GREEN, etc.) como es español (ROJO, VERDE, etc.). Puedes abrir el fichero `glcd.h` para ver todos los colores disponibles. Estos símbolos se pueden usar para indicar los colores que queremos que usen las funciones de manejo del LCD. Por ejemplo, para dibujar una línea de color amarillo entre los puntos (12, 25) y (312, 155) podemos escribir

```
#include "glcd.h"
...
glcd_linea(12, 15, 312, 155, AMARILLO);
```

Si queremos usar un color que no se encuentre entre los definidos en `glcd.h` podemos usar la macro RGB definida también allí. La macro RGB se usa especificando las componentes roja, verde y azul como valores entre 0 y 255. Por ejemplo

```
#include "glcd.h"
...
glcd_circulo(100, 100, 50, RGB(120, 100, 240));
```

También podemos definir nuestros propios colores para usarlos posteriormente.

```
#include "glcd.h"
...
#define MICOLOR RGB(120, 100, 240)
...
glcd_circulo(100, 100, 50, MICOLOR);
```

Aunque esto permite especificar 2^{24} colores, el color acaba convirtiéndose al espacio de color de 16 bits que usa el LCD. Con 16 bits pueden especificarse 65536 colores distintos. De los 16 bits, 5 se emplean para la componente roja, 6 bits para la componente verde y 5 bits para la componente azul (por eso se conoce como codificación de color 5:6:5). Por tanto, la macro RGB sólo aprovecha los 5 bits más significativos de la componente roja, los 6 bits más significativos de la componente verde y los 5 bits más significativos de la componente azul de los valores de 8 bits para rojo, verde y azul que se le pasan como parámetros.
```

10. Ejercicios

Como en la práctica anterior, puedes descargar desde el campus virtual de la asignatura un proyecto de μVision que servirá de punto de partida para la realización de los ejercicios. En este proyecto ya se ha añadido el fichero `timer_lpc40xx.c`, sólo que el cuerpo de la mayoría de sus funciones está vacío y deberá ser completado.

10.1. Ejercicio 1

Usa las funciones `timer_incializar` y `timer_retardo_ms` y completa la función `main` del programa para que en la pantalla LCD se muestre cronómetro que permita medir el transcurso del tiempo en horas, minutos y segundos. Para ello:

1. Declara variables llamadas `segundos`, `minutos` y `horas` inicializadas a 0.
2. Inicializa la pantalla LCD llamando a la función `glcd_inicializar`.
3. Inicializa el timer 0 llamando a la función `timer_inicializar`.
4. Imprime el cronómetro en la pantalla LCD. Usa para ello la función `glcd_xprintf` de `glcd.c`.
5. Llama a la función `timer_retardo_ms` para generar un retardo de un segundo mediante el timer 0.
6. Actualiza las variables `segundos`, `minutos` y `horas`. Es decir, primero incrementa la variable `segundos`. Si `segundos` toma el valor 60, ponla a 0 e incrementa `minutos`. Si `minutos` toma el valor 60, ponla a 0 e incrementa `horas`. Si `horas` toma el valor 24, ponla a 0. El cronómetro dará la vuelta cada 24 horas.
7. Vuelve al paso 4.

10.2. Ejercicio 2

En un primer momento, el cronómetro que hemos hecho en el ejercicio anterior puede parecer correcto, pero en realidad no está muy bien diseñado. La razón es que, dentro del bucle que comprende los pasos 4 a 7, los pasos 4, 6 y 7 añaden tiempo extra al segundo de retardo generado con la llamada a la función `timer_retardo_ms` en el paso 5. Además, los pasos de preparación del timer realizados en la función `timer_retardo_ms` añaden un pequeño tiempo adicional. Por tanto, cada iteración del bucle tarda realmente algo más de un segundo, por lo que el cronómetro se irá retrasando. Es cierto que el tiempo extra que se alarga cada segundo es muy pequeño en términos relativos pero vamos a intentar mejorar la situación. Para ello, empezaremos por completar las siguientes funciones del fichero `timer_lpc40xx.c`.

- ```
  void timer_iniciar_ciclos_ms(LPC_TIM_TypeDef *timer_regs,
                               uint32_t periodo_en_ms)
  ```

  Esta función programará el timer indicado por el argumento `timer_regs` para que realice continuamente ciclos con la duración en milisegundos indicada por el argumento `periodo_en_ms`.

  Al final de cada ciclo, el bit de solicitud de interrupción debe ponerse a 1.
  ```

Para conseguir el comportamiento cíclico, pon a 1 el bit MR0R (bit 1) del registro MCR. Esto hará que el registro TC se ponga a cero automáticamente cuando alcance un valor igual a MR0.

Para conseguir que el bit de solicitud de interrupción se ponga a 1 cada vez que el registro TC alcance al valor de MR0, activa también el bit MR0I (bit 0) del registro MCR.

Asegurarte de que el bit 0 del registro IR está a 0 antes de poner en marcha el timer por si acaso por alguna razón está ya 1. Recuerda que para poner a 0 bits del registro IR hay que escribir unos en los bits que se desean *borrar*.

La función no debe quedar esperando sino que, una vez programado el comportamiento cíclico deseado y puesto en marcha el timer, debe salir inmediatamente.

Usa la macro `ASSERT` para comprobar que el argumento `timer_regs` apunta al bloque de registros de uno de los timers y que el argumento `periodo_en_ms` es mayor que 0. Si hay algún error en los argumentos, aborta la ejecución del programa.

- `void timer_esperar_fin_ciclo(LPC_TIM_TypeDef *timer_regs)`

  Esta función debe esperar hasta que el bit de petición de interrupción MR0INT (bit 0 del registro IR) del timer indicado por `timer_regs` se ponga a 1. Cuando esto ocurra, pondrá dicho bit a0 y saldrá inmediatamente. Usa la macro `ASSERT` para comprobar que `timer_regs` es correcto y abortar la ejecución del programa si no lo es.

Una vez completadas estas funciones, replantea el programa del cronómetro siguiendo los siguientes pasos:

1. Declara variables llamadas `segundos`, `minutos` y `horas` inicializadas a 0.
2. Inicializa la pantalla LCD llamando a la función `glcd_inicializar`.
3. Inicializa el timer 0 llamando a la función `timer_inicializar`.
4. Llama a la función `timer_iniciar_ciclos_ms` para que el timer 0 empiece a realizar ciclos de 1 segundo (1000 ms) de duración.
5. Imprime el cronómetro en la pantalla LCD.
6. Llama a la función `timer_esperar_fin_ciclo`.
7. Actualiza las variables `segundos`, `minutos` y `horas`.
8. Vuelve al paso 5.

De esta forma, la indicación de tiempo transcurrido del cronómetro será mucho más exacta, ya que el tiempo que consumen los pasos 5, 6, 7 y 8 está incluido dentro del periodo de un segundo que tarda cada ciclo del timer.

10.3. Ejercicio 3

Completa las siguientes funciones del fichero `timer_lpc40xx.c`.

- `void timer_iniciar_conteo_ms(LPC_TIM_TypeDef *timer_regs)`

Programa el timer indicado por `timer_regs` para que, partiendo de 0, su registro TC se incremente cada milisegundo. No es necesario ajustar ningún registro de match. El registro MCR del timer debe ponerse a 0 para que el timer no se *resetee*, ni se pare ni solicite interrupciones. Usa la macro `ASSERT` para comprobar que `timer_regs` es correcto.

- `uint32_t timer_leer(LPC_TIM_TypeDef *timer_regs)`

  Retornar el valor actual del registro TC del timer. Usa la macro `ASSERT` para comprobar que `timer_regs` es correcto.

Usa las funciones de los timers en un programa que permita medir el tiempo de reacción del usuario. El comportamiento del programa debe ser el siguiente:

1. Inicializa la pantalla LCD mediante la función `glcd_inicializar`.
2. Borra la pantalla rellenándola con color negro usando la función `glcd_borrar`.
3. Imprime "Pulsa abajo para comenzar" en el LCD.
4. Espera a que el usuario pulse el joystick hacia abajo. Usa la función `leer_joystick` disponible en el fichero `joystick.c`.
5. Borra el LCD con color negro.
6. Imprime "¡Preparado!" en el LCD.
7. Genera un número pseudoaleatorio entre 1000 y 5000. Puedes usar la función `rand` de `stdlib.h`.
8. Genera un retardo de tiempo de una duración en milisegundos igual al número pseudoaleatorio generado en el paso anterior.
9. Borra la pantalla con color rojo. Cuando el usuario vea el cambio de color debe presionar la posición central del joystick tan pronto como le sea posible.
10. Imprime el tiempo en milisegundos que ha transcurrido entre el cambio de color y la pulsación.
11. Espera 5 segundos.
12. Vuelve al paso 2.