

Colección de Problemas

Problemas de Computadores

2º curso de Grado en Informática

Departamento de Ingeniería Informática y Matemáticas

Escuela Técnica Superior de Ingeniería

Universitat Rovira i Virgili

santiago.romani@urv.cat

Copyright © Santiago Romaní (*Universitat Rovira i Virgili*), Tarragona, año 2024

Índice

Problema 1: Reloj de tiempo real	4
Problema 2: Detector de inclinación	6
Problema 3: Generador de vibración.....	9
Problema 4: Piano polifónico.....	11
Problema 5: Generador de sonido	14
Problema 6: Micrófono	16
Problema 7: Servomotor	18
Problema 8: Display LCD	21
Problema 9 (1 ^a Conv. 2011-12): Paddles	25
Problema 10 (2 ^a Conv. 2011-12): Espirómetro.....	27
Problema 11 (1 ^a Conv. 2012-13): Lector de códigos de barras	29
Problema 12 (1 ^a Conv. 2013-14): Sensor de distancia.....	31
Problema 13 (2 ^a Conv. 2013-14): Tensiómetro	34
Problema 14 (1 ^a Conv. 2014-15): Teclado numérico.....	37
Problema 15 (2 ^a Conv. 2014-15): Display de LEDs	40
Problema 16 (1 ^a Conv. 2015-16): Propeller display	43
Problema 17 (1 ^a Conv. 2015-16): Velocímetro para bicicletas.....	47
Problema 18 (2 ^a Conv. 2015-16): Luz LED regulada por PWM.....	50
Problema 19 (1 ^a Conv. 2016-17): Anemómetro electrónico.....	54
Problema 20 (1 ^a Conv. 2016-17): Motor de tracción	57
Problema 21 (2 ^a Conv. 2016-17): Emisor IR	61
Problema 22 (1 ^a Conv. 2017-18): Ascensor con memoria.....	65
Problema 23 (1 ^a Conv. 2017-18): Matriz 8x8 LEDs.....	69
Problema 24 (1 ^a Conv. 2017-18): Boca artificial.....	73
Problema 25 (2 ^a Conv. 2017-18): Disparador de cámara réflex	77
Problema 26 (1 ^a Conv. 2018-19): Lectura Morse	81
Problema 27 (1 ^a Conv. 2018-19): Display numérico 4 dígitos	85
Problema 28 (1 ^a Conv. 2018-19): Lector de hojas de marcas	88
Problema 29 (2 ^a Conv. 2018-19): Envío de datos por RS-232-E	92
Problema 30 (1 ^a Conv. 2019-20): Frecuencia cardíaca.....	96
Problema 31 (1 ^a Conv. 2019-20): Joystick analógico	99
Problema 32 (2 ^a Conv. 2019-20): Caudalímetros	103
Problema 33 (1 ^a Conv. 2020-21): Tiempo de tránsito del pulso.....	106
Problema 34 (2 ^a Conv. 2020-21): Combinaciones de color.....	110
Problema 35 (1 ^a Conv. 2021-22): Metrónomo digital	115
Problema 36 (1 ^a Conv. 2021-22): Control de telescopio	118
Problema 37 (2 ^a Conv. 2021-22): Escritura Morse.....	122
Problema 38 (1 ^a Conv. 2022-23): Teléfono vintage	126
Problema 39 (1 ^a Conv. 2022-23): Lanzamiento con electroimanes.....	130

Problema 40 (2 ^a Conv. 2022-23): Escáner de líneas de texto	134
Problema 41 (1 ^a Conv. 2023-24): Semáforo con botón emergencia.....	138
Problema 42 (1 ^a Conv. 2023-24): Seguimiento de caminos	142
Problema 43 (2 ^a Conv. 2023-24): Velocidad de toque	146
Propuesta Implementación Problema 3.....	150
Propuesta Implementación Problema 13.....	152
Propuesta Implementación Problema 14.....	154
Propuesta Implementación Problema 20.....	156
Propuesta Implementación Problema 22.....	158
Propuesta Implementación Problema 30.....	161
Propuesta Implementación Problema 36.....	163
Propuesta Implementación Problema 38.....	165
Solución Problema 3	167
Solución Problema 14	168
Solución Problema 20	169

Problema 1: Reloj de tiempo real

Se propone trabajar con el reloj en tiempo real con el que está equipada la NDS para mostrar la fecha/hora actual por pantalla y activar una alarma cuando se llegue a un tiempo especificado.

Los valores de tiempo se definen con un vector de 6 números (bytes), con el siguiente contenido:

Posición	Campo	Rangos
0	Año	número del 0 al 99 (de 2000 a 2099)
1	Mes	número del 1 al 12 (de enero a diciembre)
2	Día	número del 1 al 31 (según el mes)
3	Hora	número del 0 al 23 (en modo 24 horas)
4	Minuto	número del 0 al 59
5	Segundo	número del 0 al 59

Se dispone de las siguientes rutinas, ya implementadas:

Rutina	Descripción
inicializaciones()	Realiza inicializaciones del <i>hardware</i>
tareas_independientes()	Tareas que no dependen de la alarma (ej. captación del movimiento del usuario)
swiWaitForVBlank()	Espera retroceso vertical
mostrar_tiempo(char *tiempo)	Escribe en pantalla el tiempo que se pasa por parámetro
detectar_alarma(char *t, char *a)	si el tiempo t coinciden con el tiempo de alarma a, se activa un proceso de alarma, la ejecución del cual es (aprox.) de 50 milisegundos

Además, hay que realizar la captura del tiempo real por interrupciones. Como el reloj en tiempo real NO genera interrupciones, habrá que utilizar las interrupciones del *timer 0*.

Se dispone de una rutina ya implementada, de nombre `inicializar_timer0()`, que programa la interrupción IRQ_TIMER0 con una frecuencia ligeramente superior a 1 Hz (para evitar perder segundos).

También disponemos de las siguientes rutinas para comunicarnos con el reloj de tiempo real:

Rutina	Descripción														
iniciar_RTC()	Activa el <i>chip select</i> del reloj en tiempo real														
enviar_RTC(byte comando)	Envía un comando al reloj en tiempo real; concretamente hay que enviar el valor 0x26														
byte recibir_RTC()	Recibe un byte de datos del reloj en tiempo real; después de enviar el comando 0x26 se recibirán 7 bytes consecutivos con la siguiente información: <table style="margin-left: 20px;"> <tr><td>Year Register:</td><td>BCD 00h..99h</td></tr> <tr><td>Month Register:</td><td>BCD 01h..12h</td></tr> <tr><td>Day Register:</td><td>BCD 01h..31h</td></tr> <tr><td>Day of Week Register:</td><td>00h..06h</td></tr> <tr><td>Hour Register:</td><td>BCD 00h..23h</td></tr> <tr><td>Minute Register:</td><td>BCD 00h..59h</td></tr> <tr><td>Second Register:</td><td>BCD 00h..59h</td></tr> </table>	Year Register:	BCD 00h..99h	Month Register:	BCD 01h..12h	Day Register:	BCD 01h..31h	Day of Week Register:	00h..06h	Hour Register:	BCD 00h..23h	Minute Register:	BCD 00h..59h	Second Register:	BCD 00h..59h
Year Register:	BCD 00h..99h														
Month Register:	BCD 01h..12h														
Day Register:	BCD 01h..31h														
Day of Week Register:	00h..06h														
Hour Register:	BCD 00h..23h														
Minute Register:	BCD 00h..59h														
Second Register:	BCD 00h..59h														
parar_RTC()	Desactiva el <i>chip select</i> del reloj en tiempo real														

El protocolo de comunicación es el siguiente:

- iniciar RTC
- enviar comando
- recibir, transformar y almacenar los bytes necesarios
- parar RTC

El total de tiempo para realizar esta comunicación supera los 500 microsegundos, por lo tanto, no se aconseja realizarla dentro de una RSI.

Todo el protocolo de comunicación se encapsulará dentro de una rutina de nombre `capturar_tiempo(char *tiempo)`, la cual guardará la información del tiempo real dentro del vector que se pasa por parámetro (por referencia).

En el contexto del problema, la codificación BCD (Binary Coded Decimal) son números decimales de 2 dígitos codificados dentro de un único byte, en el cual se guardan las unidades en los 4 bits de menor peso y las decenas en los 4 bits de mayor peso. Por ejemplo, el número en BCD 0x39 (00111001) representa 3 decenas (0011) y 9 unidades (1001), o sea, el valor decimal 39.

Se pide:

Programa principal en C, RSI del *timer 0* y rutina `capturar_tiempo()` en ensamblador.

Problema 2: Detector de inclinación

Se propone trabajar con el sensor de movimiento MK6, que se puede conectar a la NDS como tarjeta de expansión GBA. El programa a realizar deberá calibrar el sensor (solo al inicio de su ejecución) y después deberá representar gráficamente la inclinación de la NDS, ininterrumpidamente.



Se dispone de las siguientes rutinas, ya implementadas:

<i>Rutina</i>	<i>Descripción</i>
inicializaciones ()	Realiza inicializaciones del <i>hardware</i>
tareas_independientes ()	Tareas que no dependen del cálculo de la inclinación (ej. medir una distancia con un sensor láser)
swiWaitForVBlank ()	Espera retroceso vertical
dibujar_inclinacion(short inc_x, short inc_y)	Dibuja en pantalla una burbuja en un nivel circular según los grados de inclinación que se pasan por parámetro
calibrar_inclinacion(short *calib)	Devuelve por referencia los valores para la calibración del cálculo de la inclinación

El contenido de un vector de calibración se estructura en las siguientes posiciones de 16 bits cada una:

<i>Posición</i>	<i>Campo</i>	<i>Descripción</i>
0	xoff	Offset de la aceleración X
1	yoff	Offset de la aceleración Y
2	zoff	Offset de la aceleración Z
3	xsens	Sensibilidad de la aceleración X

4	ysens	Sensibilidad de la aceleración Y
5	zsens	Sensibilidad de la aceleración Z

Además, hay que realizar la captura de la inclinación por interrupciones. Como el sensor de movimiento no genera interrupciones, habrá que utilizar las interrupciones del *timer 0*.

Se dispone de una rutina ya implementada de nombre `inicializar_timer0()` que programa la interrupción `IRQ_TIMER0` a una frecuencia aproximada de 10 Hz.

También disponemos de las siguientes rutinas para comunicarnos con el sensor de movimiento:

Rutina	Descripción
<code>iniciar_MK6()</code>	Activa el <i>chip select</i> del sensor de movimiento
<code>enviar_MK6(byte comando)</code>	Envía un comando al sensor de movimiento; concretamente, tendremos que enviar 2 comandos: Read_X: 0x00 Read_Y: 0x02
<code>short recibir_MK6()</code>	Recibe un short (16 bits) del sensor de movimiento, con el valor de aceleración del eje indicado con el comando anterior
<code>finalizar_MK6()</code>	Desactiva el <i>chip select</i> del sensor de movimiento

El protocolo de comunicación es el siguiente:

- iniciar MK6
- enviar comando `Read_X`
- recibir, transformar y almacenar inclinación X
- enviar comando `Read_Y`
- recibir, transformar y almacenar inclinación Y
- finalizar MK6

El total de tiempo para realizar esta comunicación no supera los 100 microsegundos, es decir, se puede incluir dentro de una RSI. Por lo tanto, el protocolo de comunicación se realizará dentro de la RSI del *timer 0*, que guardará los valores de inclinación dentro de dos variables globales `inclin_X` e `inclin_Y`.

Además, hay que transformar el valor de aceleración de cada eje al valor de inclinación, con la siguiente fórmula (ejemplo para el eje X):

$$\text{inclin_X} = \text{inclin_X} + (\text{acel_X} - \text{xoff}) * \text{xsens};$$

donde `inclin_X` es el valor de inclinación actual y `acel_X` es el valor de aceleración que devuelve el sensor de movimiento, y `xoff`, `xsens` son los valores de calibración del eje X; la fórmula para el eje Y es análoga.

Se pide implementar una función específica para realizar este cálculo, de nombre `convertir_aceleracion(short *incl, short acel, short off, short sens)` donde `incl` (inclinación) se pasa por referencia y el resto de los parámetros se pasan por valor.

Se pide:

Programa principal en C, RSI del `timer0` y rutina `convertir_aceleracion()` en ensamblador.

Problema 3: Generador de vibración

Se propone interactuar con un generador de vibración *Rumble Pak* integrado en un cartucho de expansión GBA.



Se dispone de las siguientes rutinas, ya implementadas:

<i>Rutina</i>	<i>Descripción</i>
<code>inicializaciones ()</code>	Realiza inicializaciones del <i>hardware</i>
<code>scanKeys ()</code>	Captura la pulsación actual de las teclas
<code>int keysDown ()</code>	Devuelve el estado de las últimas teclas pulsadas
<code>retardo(int dsec)</code>	Espera el paso de tantas décimas de segundo como indique el parámetro
<code>swiWaitForVBlank ()</code>	Espera retroceso vertical
<code>printf(const char * format, ...)</code>	Imprime por pantalla un mensaje de texto con formato

Hay que implementar una rutina denominada `generar_vibracion(short freq)`, cuya función será iniciar la vibración a la frecuencia especificada por parámetro, en Hercios. Si la frecuencia es cero, se parará la vibración.

El *hardware* para generar la vibración consiste en un registro de nombre `REG_RUMBLE`. Cada vez que se cambia el valor del bit 1 de este registro se produce un movimiento del dispositivo vibrador. Por lo tanto, para generar vibración a una determinada frecuencia hay que cambiar dicho bit a la frecuencia requerida.

El funcionamiento del programa principal tiene que ser el siguiente:

- inicializaciones
- bucle principal (infinito)
 - capturar teclas
 - si tecla X, iniciar vibración a 5 Hz
 - si tecla Y, iniciar vibración a 20 Hz
 - si tecla A, iniciar vibración a 50 Hz
 - si vibración iniciada,
 - esperar 5 décimas de segundo y parar vibración
 - sincronización de pantalla
 - escribir frecuencia de la última vibración activada

Para generar la vibración a la frecuencia indicada utilizaremos el *timer 0*, con los registros:

0x04000100	TIMER0_DATA	Valor del contador / carga del divisor de frecuencia
0x04000102	TIMER0_CR	Registro de control del <i>timer 0</i>

donde **TIMER0_DATA** se utiliza para cargar el divisor de frecuencia y **TIMER0_CR** se utiliza para iniciar y parar la generación de interrupciones periódicas, donde este segundo registro deberá contener los siguientes valores específicos para iniciar dichas interrupciones:

Característica	Bits	Valor	Descripción
Prescaler Selection	1..0	11	frecuencia de entrada aprox. 32.728 Hz
Count-up Timing	2	0	No
Timer IRQ Enable	6	1	Sí
Timer Start/Stop	7	1	Start

Para detener la generación de interrupciones periódicas bastará con escribir un cero en el registro de control. Para calcular el divisor de frecuencia hay que aplicar la siguiente fórmula:

$$\text{Div_Frecuencia} = -(\text{Freq_Estable} / \text{Freq_Salida})$$

Para realizar la división se llamará a una función de la BIOS con la instrucción de lenguaje máquina `swi 9`, pasando el numerador en R0 y el denominador en R1; la función devuelve el cociente (con signo) en R0, el resto en R1 y el valor absoluto del cociente en R3.

Se pide:

Programa principal en C, RSI del *timer 0* y rutina `generar_vibracion()` en ensamblador.

Problema 4: Piano polifónico

Se propone trabajar con un teclado de piano “NDS Easy Piano option pack”, que se puede conectar a la NDS como tarjeta de expansión GBA.



El programa a realizar debe consultar periódicamente el registro de 16 bits específico del piano, que se encuentra en la posición 0x09FFFFE, el cual proporciona un bit para cada una de las 13 teclas del piano:

<i>Bit</i>	<i>Campo</i>	<i>Nota</i>	<i>Código de nota</i>
0	PIANO_C	Do	0
1	PIANO_CS	Do#	1
2	PIANO_D	Re	2
3	PIANO_DS	Re#	3
4	PIANO_E	Mi	4
5	PIANO_F	Fa	5
6	PIANO_FS	Fa#	6
7	PIANO_G	Sol	7
8	PIANO_GS	Sol#	8
9	PIANO_A	La	9
10	PIANO_AS	La#	10
13	PIANO_B	Si	11
14	PIANO_C2	Do (segunda escala)	12

Se dispone de las siguientes rutinas, ya implementadas:

Rutina	Descripción
inicializaciones()	Realiza inicializaciones del <i>hardware</i>
tareas_independientes()	Tareas que no dependen del control de teclas (ej. visualizar una partitura)
swiWaitForVBlank()	Espera retroceso vertical
generar_nota(char código, short volumen)	Inicia la reproducción de una nota de piano según el código de nota y el volumen inicial
modular_volumen(char código, short volumen)	Cambia el volumen de una nota

El código de cada nota será el número de bit mostrado en la primera tabla (primera columna). Los valores de volumen oscilarán entre 16 y 0, donde 16 será el volumen máximo, 1 será el mínimo y 0 indicará que la nota debe estar en silencio.

Para realizar la detección de las pulsaciones o liberaciones de las teclas del piano se propone utilizar la rutina de servicio de interrupción RSI del retroceso vertical (60 Hz), puesto que el piano no genera interrupciones.

Hay que tener en cuenta que el generador de notas de la NDS dispone de 16 canales independientes, de modo que podemos utilizar 13 de ellos, uno para cada nota. De este modo se podrán tocar hasta 13 notas simultáneamente (polifonía).

La activación de los canales de sonido, con sus respectivas frecuencias y volúmenes, se gestionará con las dos rutinas proporcionadas `generar_nota()` y `modular_volumen()`. Estas rutinas tardan menos de 5 microsegundos en ejecutarse.

Las tareas que tiene que controlar el programa son la siguientes:

- detección de las teclas del piano pulsadas / liberadas
- detección del tiempo de pulsación de cada tecla del piano

En la primera tarea, cuando se detecta el inicio de la pulsación de una tecla del piano hay que activar la generación de la nota correspondiente al volumen máximo, y cuando se detecta la liberación de una tecla previamente pulsada hay que parar la generación de la nota correspondiente fijando su volumen a 0.

En la segunda tarea, a medida que transcurre el tiempo en que se mantiene una tecla de piano pulsada hay que reducir su volumen progresivamente, a razón de un nivel cada dos décimas de segundo, es decir, que cada nota se extinguirá después de 32 décimas de segundo, a no ser que soltemos la tecla antes de este límite temporal.

Para cada nota se debe utilizar una estructura de información con los siguientes campos:

```
typedef struct {
    short bit_masc;      // máscara del bit de la nota
    short pressed;        // =1 indica que está pulsada
    short cont_ret;       // contador de retrocesos de la pulsación
    short volumen;        // volumen actual
} info_nota;
```

Se debe implementar una función específica para realizar el cálculo del volumen de cada nota, `actualizar_volumen(info_nota *nota, char codigo)`, la cual se invocará a cada retroceso vertical para las teclas pulsadas activas.

Se pide:

Programa principal en C, RSI del retroceso vertical y rutina `actualizar_volumen()` en ensamblador.

Problema 5: Generador de sonido

Se propone interactuar con el *hardware* de generación de sonido de la NDS. Concretamente, se trata de implementar unas funciones para activar notas musicales con una determinada frecuencia y duración.

Las rutinas a implementar son las siguientes:

Rutina	Descripción
activar_nota(char canal, short freq, short vol)	Activa la reproducción de una nota por un canal de sonido, a una determinada frecuencia (en Hz) y a un determinado volumen (127..0) durante un tiempo indefinido
RSI_timer0()	Rutina de servicio de interrupciones del <i>timer</i> 0: se encargará de controlar la duración de la nota actual y de activar la nota siguiente

Para accionar la nota en cada canal hay que acceder al registro de control (0x040004X0) y al registro de *timer* (0x040004X4) del canal especificado, donde X (mayúscula) indica el número de canal como dígito hexadecimal (de 0 a F). Los campos de dichos registros significan lo siguiente:

SOUND_X_CNT – SOUND Channel X Control Register (32 bits)

Bits	Campo	Descripción
6..0	Volume	Nivel de volumen, de 0 a 127 (0 es silencio)
28..27	Repeat Mode	01: bucle infinito, 10: una vez
31	Start / Stop	0: Parar, 1: iniciar

SOUND_X_TMR – SOUND Channel X Timer Register (16 bits)

Bits	Campo	Descripción
15..0	Timer value	Divisor de frecuencia de entrada para que la frecuencia de salida sea igual a freq: Timer value = - (33513982/2) / freq

Se dispone de las siguientes rutinas, ya implementadas:

Rutina	Descripción
inicializaciones()	Realiza inicializaciones del <i>hardware</i>

tareas_independientes()	Tareas independientes de la generación de sonido (ej. gestión de un juego)
swiWaitForVBlank()	Espera retroceso vertical
printf(const char * format, ...)	Imprime por pantalla un mensaje de texto con formato

Para cada nota se debe utilizar una estructura de información con los siguientes campos:

```
typedef struct {
    short freq; // frecuencia de la nota (Hz)
    short time; // tiempo de la nota (en centésimas de segundo)
    short vol; // volumen de la nota (0..127)
} info_note;
```

Todas las notas a tocar se encuentran en un vector con un número de posiciones igual a una constante MAX_NOTAS:

```
info_note musica[MAX_NOTAS];
```

El funcionamiento del programa principal tiene que ser el siguiente:

- inicializaciones
- leer primera nota
- activar primera nota
- bucle principal (infinito)
 - tareas independientes
 - sincronización de pantalla
 - escribir por pantalla el índice de la nota actual (solo cuando haya un cambio de nota)

Mientras tanto, la RSI del *timer 0* se activará a 100 Hz y se encargará de controlar el tiempo de cada nota y de cargar la siguiente. Cuando llegue a la última nota, volverá a empezar desde el principio.

Para realizar la división para el cálculo del divisor de frecuencia del controlador de sonido se llamará a una función de la BIOS con la instrucción de lenguaje máquina swi 9, pasando el numerador en R0 y el denominador en R1: la función devuelve el cociente (con signo) en R0, el resto en R1 y el valor absoluto del cociente en R3.

Se pide:

Programa principal en C y la RSI del *timer 0* y la rutina `activar_nota()` en ensamblador. Se utilizará el canal de sonido 0.

Problema 6: Micrófono

Se propone capturar audio con el micrófono con el que está equipada la plataforma NDS. El programa a realizar debe capturar sonido continuamente y mostrar por pantalla una visualización gráfica instantánea de la intensidad de cada rango de frecuencias.

Se dispone de las siguientes rutinas, ya implementadas:

Rutina	Descripción
inicializaciones ()	Realiza inicializaciones del <i>hardware</i>
tareas_independientes ()	Tareas que no dependen de la captación del audio (ej. reconocimiento de texto en el audio capturado anteriormente)
swiWaitForVBlank ()	Espera retroceso vertical
mostrar_frecuencias (char *audio)	Dibuja en una pantalla de la NDS la distribución de intensidad de diferentes rangos de frecuencias

Además, hay que realizar la captura del audio utilizando las interrupciones del *timer* 0. Se dispone de una rutina ya implementada de nombre `inicializar_timer0()` que programa la interrupción IRQ_TIMER0 con una frecuencia de 11 KHz.

También disponemos de las siguientes rutinas para comunicarnos con el micrófono:

Rutina	Descripción
iniciar_MIC ()	Activa el <i>hardware</i> del micrófono
byte recibir_MIC ()	Recibe un byte de datos del micrófono, con el nivel de audio actual digitalizado sobre el rango 0..255
parar_MIC ()	Desactiva el <i>hardware</i> del micrófono

El protocolo de comunicación consta de los siguientes comandos:

- iniciar el micrófono, una vez al principio del programa
- recibir y almacenar los bytes necesarios
- parar el micrófono, al final del programa

Cada comando de comunicación tarda menos de 40 microsegundos en ejecutarse.

La problemática principal del programa consiste en gestionar los “trozos” de audio que se tienen que pasar a la función `mostrar_frecuencias()`. Esta función recibe por parámetro un *buffer* de 1.100 bytes con el sonido muestreado a 11 KHz, es decir, todo el sonido capturado por el micrófono en una décima de segundo.

El cálculo del gráfico de frecuencias es relativamente lento, ya que puede tardar entre 20 y 80 milisegundos. Mientras se está realizando este cálculo, el *buffer* no se puede modificar. Por lo tanto, los datos capturados por el micrófono durante ese tiempo deben ser almacenados en otro *buffer*.

En consecuencia, para gestionar la información de audio se usarán dos *buffers*, más dos variables de soporte:

```
char buffer_mic[2][1100];
short mic_buffer_actual;
short mic_index;
short mic_buffer_disponible;
```

Los *buffers* se pueden referenciar como `buffer_mic[0]` y `buffer_mic[1]`. La variable `mic_buffer_actual` indicará sobre qué *buffer* (0 o 1) se está guardando la información que capture el micrófono. La variable `mic_index` se utilizará para saber en qué posición del *buffer* actual se tiene que almacenar la captura del micrófono. La variable `mic_buffer_disponible` indicará si ya existe un *buffer* de información de audio disponible para ser visualizada por la función `mostrar_frecuencias()`.

En definitiva, habrá que controlar el micrófono para ir capturando datos sobre un *buffer* mientras la función de `mostrar_frecuencias()` trabajará sobre los datos del otro *buffer*.

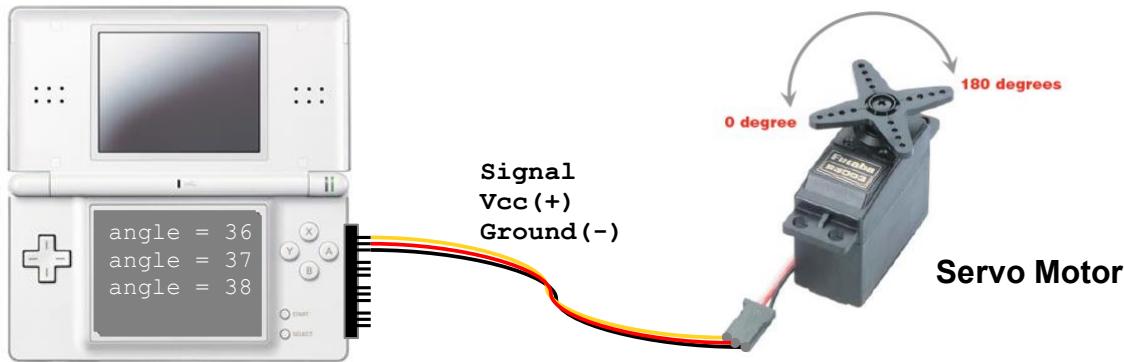
Habrá que crear una rutina de nombre `cambiar_buffers()`, que cambiará el buffer actual de captura.

Se pide:

Programa principal en C, RSI del *timer 0* y rutina `cambiar_buffers()` en ensamblador.

Problema 7: Servomotor

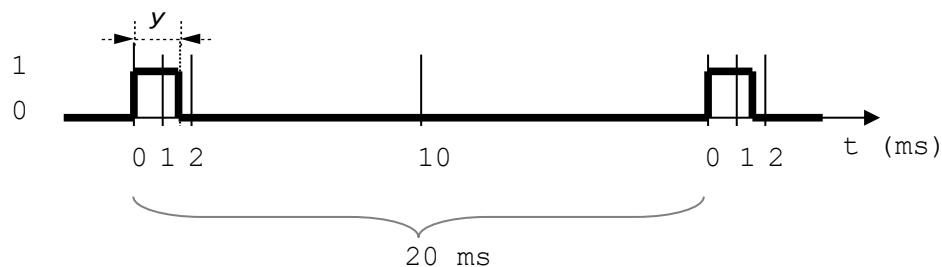
Se propone controlar el ángulo de giro de un servomotor tipo SG90 con la NDS. El programa a realizar permitirá al usuario seleccionar un valor del ángulo entre 0° y 180° :



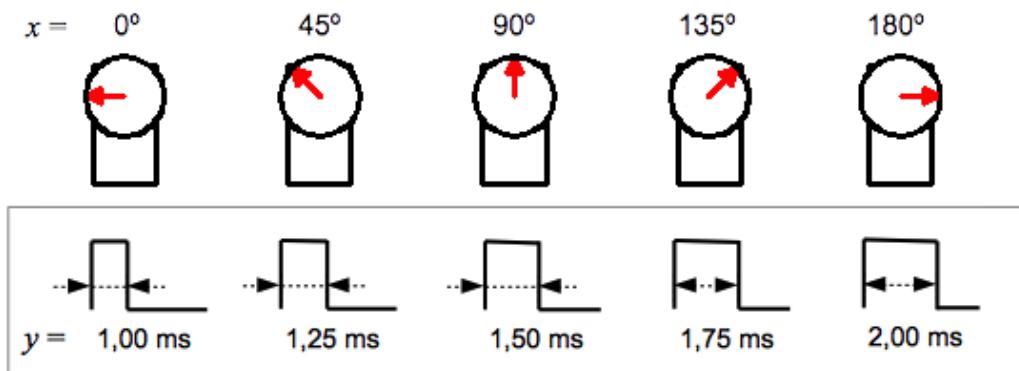
En el esquema se han representado los pines de un adaptador de Entrada/Salida que permite a la NDS controlar hasta 4 servomotores simultáneamente. El controlador de E/S del adaptador se gestiona a través de un registro de E/S denominado REG_SERVO, de 8 bits, de los cuales solo se utilizan los 4 bits de menor peso.

Cada uno de estos 4 bits permite controlar el cable de señal (amarillo) de uno de los servomotores. Los otros dos cables de cada servomotor se conectan a pines de corriente (rojo) y masa (negro) correspondientes. El cable de señal del servomotor a controlar está conectado al bit 3 del registro.

Para regular el ángulo del servomotor, el cable de señal debe emitir un pulso de control periódico de 20 milisegundos, donde el tiempo en que el pulso está a 1 (y) determinará el ángulo de giro (x):



Concretamente, el tiempo debe variar entre 1 ms para 0° y 2 ms para 180° , obteniendo todas las orientaciones posibles según la variación proporcional de dicho tiempo.



En los esquemas anteriores se ha mostrado el estado del servomotor para 5 orientaciones concretas ($0^\circ, 45^\circ, 90^\circ, 135^\circ, 180^\circ$), aunque se puede conseguir prácticamente cualquier valor (entero) de grados. La siguiente fórmula permite calcular el número de milisegundos del tiempo a 1 (y) a partir del valor del ángulo en grados sexagesimales (x):

$$y = 1 + \frac{x}{180} \text{ (ms)}, \quad x \in [0..180] (\text{°})$$

Como en lenguaje máquina no podemos trabajar con valores reales (coma flotante), podemos adaptar la fórmula anterior para expresar el tiempo en microsegundos, lo cual nos permitirá realizar los cálculos necesarios utilizando solo variables enteras.

El programa a implementar deberá permitir al usuario modificar el ángulo actual de giro en todo momento, utilizando los botones de las flechas derecha e izquierda para aumentar y disminuir el valor del ángulo en unidades, y los botones alternativos de derecha e izquierda (situados en la parte trasera de la consola) para aumentar y disminuir el valor del ángulo en decenas, todo ello sin superar los límites del rango permitido, obviamente. Además, cada vez que se cambie el valor actual del ángulo, éste se deberá mostrar por la pantalla inferior de la NDS.

Se dispone de las siguientes rutinas, ya implementadas:

Rutina	Descripción
inicializaciones ()	Inicializa el <i>hardware</i> (pantalla, interrupciones, etc.)
tareas_independientes ()	Tareas que no dependen del acceso al servomotor conectado al bit 3 (ej. controlar otros servomotores según otras fuentes de control, como un sensor de distancia); tiempo ejecución < 100 ms
scanKeys ()	Captura el estado actual de los botones de la NDS

<code>int keysDown()</code>	Devuelve un patrón de bits con los botones activos
<code>swiWaitForVBlank()</code>	Espera hasta el próximo retroceso vertical
<code>printf(char *format, ...)</code>	Escribe un mensaje en la pantalla inferior de la NDS

Para generar la forma del pulso correspondiente al ángulo requerido, se pide utilizar la RSI del *timer* 0, que se configurará (por la rutina de inicialización) para que se invoque cada vez que se active la correspondiente interrupción. El tiempo en que tardará a generarse la interrupción deberá alternar entre el tiempo para el estado del pulso a 1 y el tiempo restante del ciclo para el estado del pulso a 0.

Se propone usar las siguientes variables globales:

```
unsigned char x;           // valor actual del ángulo
unsigned char pulse_state; // estado actual del pulso
unsigned short y_mic;      // número de microsegundos a 1
```

La variable `pulse_state` permitirá saber el estado actual del pulso, de modo que la RSI del *timer* pueda cambiar alternativamente el estado del bit 3 del registro REG_SERVO. La variable `y_mic` almacenará el tiempo en que el pulso debe estar a 1, en microsegundos.

Para activar el *timer* 0 con un determinado periodo, se pide realizar una rutina específica:

```
void fijar_divfrectim0(unsigned short micros);
```

la cual recibe por parámetro el número de microsegundos del período del *timer*. Por lo tanto, esta rutina debe calcular el divisor de frecuencia correspondiente para que, después del tiempo especificado, se active la interrupción correspondiente. Como frecuencia de entrada, se sugiere utilizar la F/64.

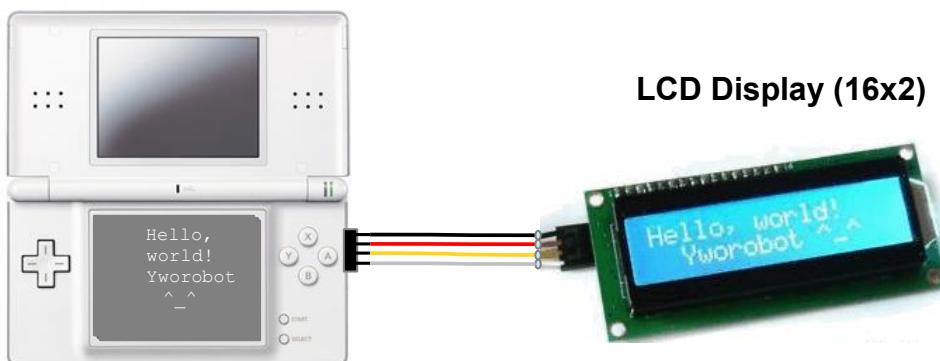
Para realizar las divisiones desde lenguaje ensamblador se puede utilizar la rutina de la BIOS `swi 9` (entrada: R0 = numerador, R1 = divisor; salida: R0 = cociente, R1 = resto, R3 = cociente sin signo), que puede tardar entre 5 y 20 microsegundos.

Se pide:

Programa principal en C, RSI del *timer* 0 y rutina `fijar_divfrectim0()` en ensamblador.

Problema 8: Display LCD

Se propone controlar con la NDS un display LCD de 2 filas x 16 columnas tipo HD44780U. El programa a realizar recibirá mensajes por wifi y los representará en el display:



En el esquema anterior se han representado los pines de un adaptador de Entrada/Salida que permite a la NDS enviar comandos al LCD mediante dos cables de datos (más dos cables de alimentación). En realidad, el display recibirá los datos serializados, pero de este proceso ya se encargará el controlador de E/S del adaptador.

Desde el punto de vista del programa a realizar, el controlador dispone de un registro de E/S denominado REG_DISPLAY, de 16 bits, de los cuales solo se utilizan los 10 bits de menor peso. La siguiente tabla muestra los posibles comandos que se pueden enviar con estos 10 bits, junto con sus respectivos parámetros:

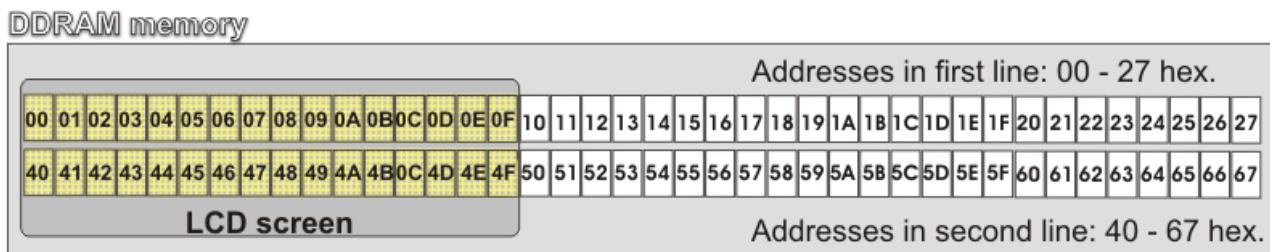
Instruction	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Clear display	0	0	0	0	0	0	0	0	0	1
Cursor home	0	0	0	0	0	0	0	0	1	x
Entry mode set	0	0	0	0	0	0	0	1	I/D	S
Display on/off control	0	0	0	0	0	0	1	D	C	B
Cursor/display shift	0	0	0	0	0	1	S/C	R/L	x	x
Function set	0	0	0	0	1	DL	N	x	BR1	BR0
CGRAM address set	0	0	0	1	CGRAM address					
DDRAM address set	0	0	1	DDRAM address						
Address counter read	0	1	BF=0	AC contents						
DDRAM or CGRAM write	1	0	Write data							
DDRAM or CGRAM read	1	1	Read data							

x = don't care

De todos estos comandos, para resolver este problema de examen nos interesan tres:

- Cursor/display shift: $\text{REG_DISPLAY} = 000001 (\text{S/C}) (\text{R/L}) \text{ xx}$
 - S/C: 1 → screen, 0 → cursor
 - R/L: 1 → right, 0 → left
- DDRAM Address set: $\text{REG_DISPLAY} = 001 (\text{DDRAM address})$
 - DDRAM address (7 bits)
- DDRAM or CGRAM write: $\text{REG_DISPLAY} = 10 (\text{Write data})$
 - Write data (8 bits)

Internamente, el display dispone de una memoria RAM de 80 posiciones de 1 byte cada una, denominada DDRAM (*Display Data RAM*), distribuidas en 2 filas de 40 columnas, aunque en la pantalla LCD (*screen*) solo se visualizan 2 filas por 16 columnas:



En el gráfico anterior, para cada posición de la DDRAM se muestra su dirección de memoria, en hexadecimal. Sin embargo, hay que tener en cuenta que en cada posición se guardará el código del carácter a visualizar, típicamente con su codificación ASCII.

Como la pantalla LCD solo dispone de 16 columnas, para poder visualizar todo el contenido de la memoria será necesario desplazar la posición inicial de la pantalla hacia la derecha o hacia la izquierda, con el comando “Cursor/display shift”.

El comando “DDRAM address set” permite fijar la dirección de la posición de memoria que se requiere leer o escribir. El comando “DDRAM or CGRAM write” permite escribir un dato (un byte) en la posición de memoria previamente fijada con el comando anterior. Cuando se fija una dirección de memoria, se puede escribir un conjunto de caracteres consecutivamente, sin tener que especificar la dirección para cada carácter, ya que el propio display se encargará de aumentar automáticamente la posición actual de escritura en memoria.

El programa a implementar deberá recibir mensajes de hasta 32 caracteres por la wifi y transferirlos a la memoria DDRAM, mediante los comandos adecuados. Cada nuevo mensaje que se reciba se escribirá en la segunda linea de la DDRAM, mientras que el contenido

anterior de la segunda linea se deberá copiar a la primera linea, realizando un efecto de *scroll* vertical. Los mensajes recibidos también se deberán mostrar por la pantalla inferior de la NDS.

Además, la visualización de la pantalla del display deber ir desplazándose para que se pueda leer todo el contenido de los mensajes, cada cierto tiempo, realizando un efecto de *scroll* horizontal. Se propone el siguiente algoritmo:

- Visualización de las 16 primeras columnas; espera de 3 segundos
- Desplazamiento gradual de 16 columnas a la derecha, durante 2 segundos
- Visualización de las 16 columnas siguientes; espera de 3 segundos
- Desplazamiento gradual de 16 columnas a la izquierda, durante 2 segundos

Estos pasos se deben repetir indefinidamente, es decir, cuando se acaba el último paso se vuelve a empezar por el primero. El algoritmo está simplificado, en el sentido de que no tiene en cuenta la longitud concreta de los mensajes cuando realiza el desplazamiento. Tampoco será necesario reiniciar el algoritmo en el momento que se inserte un nuevo mensaje.

Se dispone de las siguientes rutinas, ya implementadas:

<i>Rutina</i>	<i>Descripción</i>
initializaciones ()	Inicializa el <i>hardware</i> (pantalla, interrupciones, etc.)
tareas_independientes ()	Tareas que no dependen del acceso al display LCD (ej. realizar una animación en la pantalla superior); tiempo ejecución < 200 ms
int wifiReceiveText(char *)	Rutina de recepción de un mensaje por la interfaz wifi de la NDS; si hay un nuevo mensaje (desde la última llamada), copiará los bytes de dicho mensaje sobre el string que se pase por referencia (min. 33 posiciones) y devolverá 1; si no hay nuevo mensaje, devolverá 0
sincro_display ()	Espera a que el display esté preparado para recibir un nuevo comando; como máximo, tardará 41 μ s
strcpy(char *dest, const char *source)	Copia el string que se pasa por segundo parámetro sobre el string que se pasa por primer parámetro
swiWaitForVBlank()	Espera hasta el próximo retroceso vertical
printf(char *format, ...)	Escribe un mensaje en la pantalla inferior

Para realizar el desplazamiento horizontal del display, se pide utilizar la RSI del *timer 0*, que se configurará (por la rutina de inicialización) para que se invoque 8 veces por segundo.

Para transferir los caracteres sobre las dos líneas de la DDRAM, se pide realizar la siguiente rutina específica:

```
void insertar_strings(char str1[], char str2[]);
```

la cual recibe por parámetro y por referencia dos vectores de caracteres, de 32 posiciones cada uno, como mínimo, desde los cuales se copiarán los códigos ASCII de cada carácter sobre las 32 primeras columnas de la DDRAM.

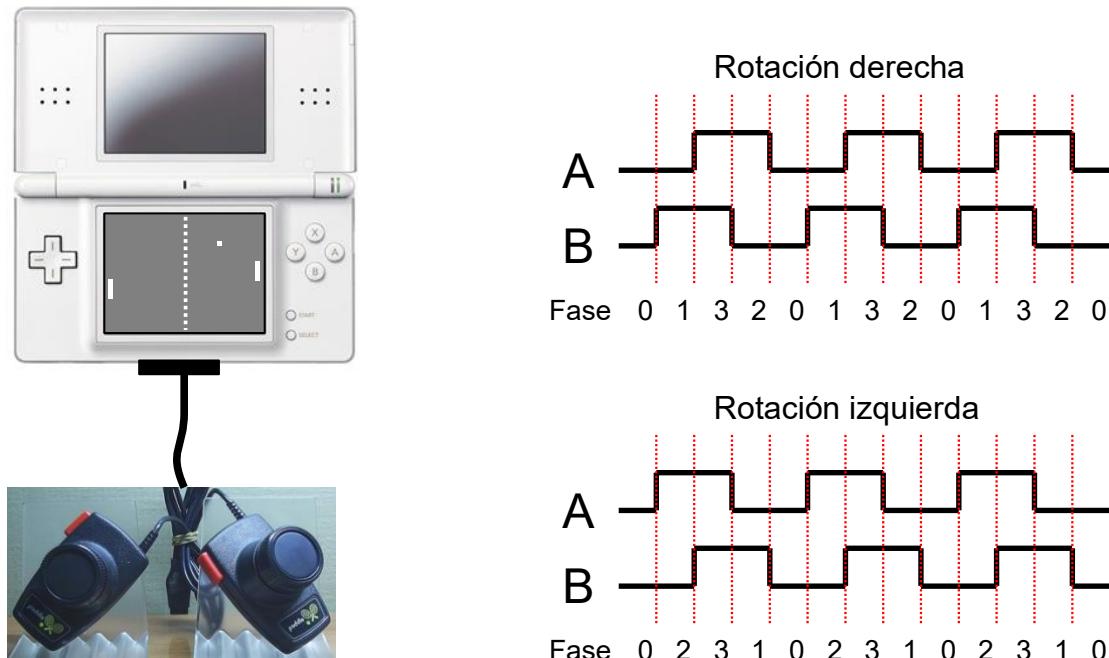
Para sincronizarse con el display, se debe invocar a la rutina `sincro_display()` antes de enviar cualquier comando al display. Esta rutina no retornará hasta que el display no esté preparado, pero se nos asegura que, como máximo, tardará 41 microsegundos en retornar.

Se pide:

Programa principal y variables globales en C, RSI del *timer 0* y rutina `insertar_string()` en ensamblador.

Problema 9 (1^a Conv. 2011-12): Paddles

Se propone trabajar con un par de controles rotatorios de tipo *paddle*, que se puede conectar a la NDS como tarjeta de expansión GBA:



Estos controles permiten indicar velocidad y sentido de la rotación, usando simplemente 2 bits, de nombres A y B, que definen cuatro fases de la rotación, 0, 1, 2 y 3, según la codificación binaria del estado de los dos bits, donde A es el bit de más peso (AB = 00 → fase 0, AB = 01 → fase 1, etc.)

Cada vez que se cambia de fase indica el movimiento de una posición de la pala. Para detectar en qué sentido se mueve (incremento / decremento) hay que verificar si las fases siguen la secuencia (0, 1, 3, 2) o bien la secuencia inversa (0, 2, 3, 1).

El programa a realizar debe consultar periódicamente el registro de 16 bits específico que se encuentra en la posición 0x0A000000, el cual proporciona 3 bits por cada *paddle*:

Bit	Campo	Descripción
0	PADDLE1_C	Botón del <i>paddle</i> 1
1	PADDLE1_B	Bit B del <i>paddle</i> 1
2	PADDLE1_A	Bit A del <i>paddle</i> 1
3	PADDLE2_C	Botón del <i>paddle</i> 2
4	PADDLE2_B	Bit B del <i>paddle</i> 2
5	PADDLE2_A	Bit A del <i>paddle</i> 2

Se dispone de las siguientes rutinas, ya implementadas:

Rutina	Descripción
inicializaciones()	Realiza inicializaciones del <i>hardware</i>
tareas_independientes()	Tareas que no dependen del movimiento de los <i>paddles</i> (ej. controlar el movimiento de la pelota y el juego en general)
swiWaitForVBlank()	Espera retroceso vertical
dibujar_raqueta(short posX, short posY)	Dibuja una raqueta de juego según la posición indicada.

El programa principal consistirá en un juego de tenis: a parte de invocar a las tareas independientes, se encargará de dibujar dos raquetas con la posición X fijada por dos constantes (`posX1`, `posX2`) y la posición Y definida por dos variables globales (`PosY1`, `PosY2`).

Para realizar la detección del movimiento de los *paddles* se propone utilizar las interrupciones del *timer 0*, ya que el controlador de los *paddles* no genera interrupciones. Se dispone de una rutina ya implementada, de nombre `inicializar_timer0()`, que programa la interrupción `IRQ_TIMER0` a una frecuencia aproximada de 300 Hz.

La RSI del *timer 0* tiene que detectar si hay un cambio en la fase de cada *paddle*. En caso afirmativo tiene que llamar a una rutina de nombre `detectar_sentido()`, que recibirá dos parámetros, la fase anterior y la fase actual. A partir de los dos valores de fase, la rutina devolverá 1 si el sentido es hacia la derecha, -1 si el sentido es hacia la izquierda, o 0 si los valores de fase no se corresponden a ninguna secuencia.

```
short detectar_sentido(char f_ant, char f_act);
```

Según el resultado de esta rutina, la RSI del *timer 0* incrementará, decrementará o no hará nada sobre la variable global `PosY` correspondiente.

Para detectar el sentido se recomienda utilizar dos vectores de bytes que indiquen, para cada número de fase anterior (índice del vector) cual es la fase actual siguiente para cada uno de los sentidos (contenido del vector):

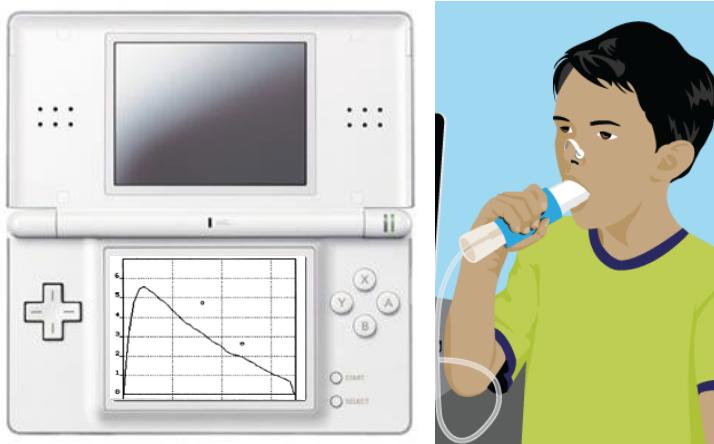
```
char s_derecha[] = {1, 3, 0, 2};
char s_izquierda[] = {2, 0, 3, 1};
```

Se pide:

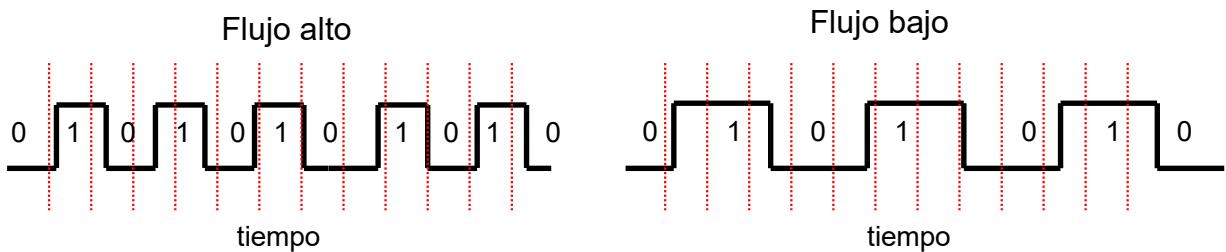
Programa principal en C, RSI del *timer 0* y rutina `detectar_sentido()` en ensamblador.

Problema 10 (2^a Conv. 2011-12): Espirómetro

Se propone construir un espirómetro con una NDS. Un espirómetro es un aparato para medir la capacidad pulmonar. Consta de un tubo equipado con una hélice, la cual gira como consecuencia del aire que insufla el usuario:



La hélice genera un tren de impulsos (0/1/0/1/0/...) que indicará el nivel del flujo del aire; cuanto más alto sea el flujo, más impulsos (cambios de 0 a 1) por unidad de tiempo se obtendrán. Ejemplos:



El tren de impulsos se recibirá por el bit 0 del registro de E/S 0x0A000180 (reg. de 8 bits). El programa utilizará las interrupciones del *timer* 0 para consultar el valor de este bit periódicamente. Se dispone de una rutina ya implementada, de nombre `inicializar_timer0()`, que programa la interrupción IRQ_TIMER0 a una frecuencia aproximada de 2 KHz.

El programa esperará la pulsación de la tecla START para empezar a contar impulsos. Se supone que el flujo máximo de una persona normal puede generar entre 200 y 400 impulsos por segundo. Se puede suponer que nunca se llegará hasta los 600 impulsos por segundo.

Después de pulsar la tecla START, el programa representará gráficamente el nivel del flujo de aire de cada instante durante 10 segundos. Concretamente, hay que pintar un punto en la gráfica cada 5 centésimas de segundo. Esto supone un total de 200 puntos para toda la gráfica.

El valor del tiempo nos proporciona la coordenada X y el nivel de flujo nos proporciona la coordenada Y. Concretamente, hay que contar el número de impulsos para cada intervalo de tiempo, es decir, cada 5 centésimas. El número máximo de impulsos por intervalo es de 600/20, es decir, 30.

Se dispone de las siguientes rutinas, ya implementadas:

Rutina	Descripción
inicializaciones()	Inicializa pantalla e interrupciones
scanKeys()	Captura las teclas
int keysDown()	Devuelve el estado de las teclas pulsadas
swiWaitForVBlank()	Espera retroceso vertical
dibujar_ejes()	Dibuja los ejes para representar el gráfico
add_pixel(int px, int py)	Añade un píxel al gráfico, según las coordenadas de pantalla px (20-220) y py (0-180)
actualizar_grafico()	Actualiza el dibujo del gráfico

La rutina `dibujar_ejes()` solo se tiene que llamar una vez, antes de empezar la captura de los niveles de flujo. La rutina `add_pixel()` tarda menos de 100 microsegundos en ejecutarse. La rutina `actualizar_grafico()`, sin embargo, puede tardar hasta 5 milisegundos en ejecutarse. Esta rutina se asegura de activar todos los píxeles entre el último píxel añadido y el penúltimo, de modo que todo el gráfico sea una línea continua.

Se debe realizar una rutina que se encargará de convertir los valores de tiempo y flujo en coordenadas de pantalla, para que se puedan activar los píxeles correspondientes:

```
void convertir_punto(int ppx, int ppy);
```

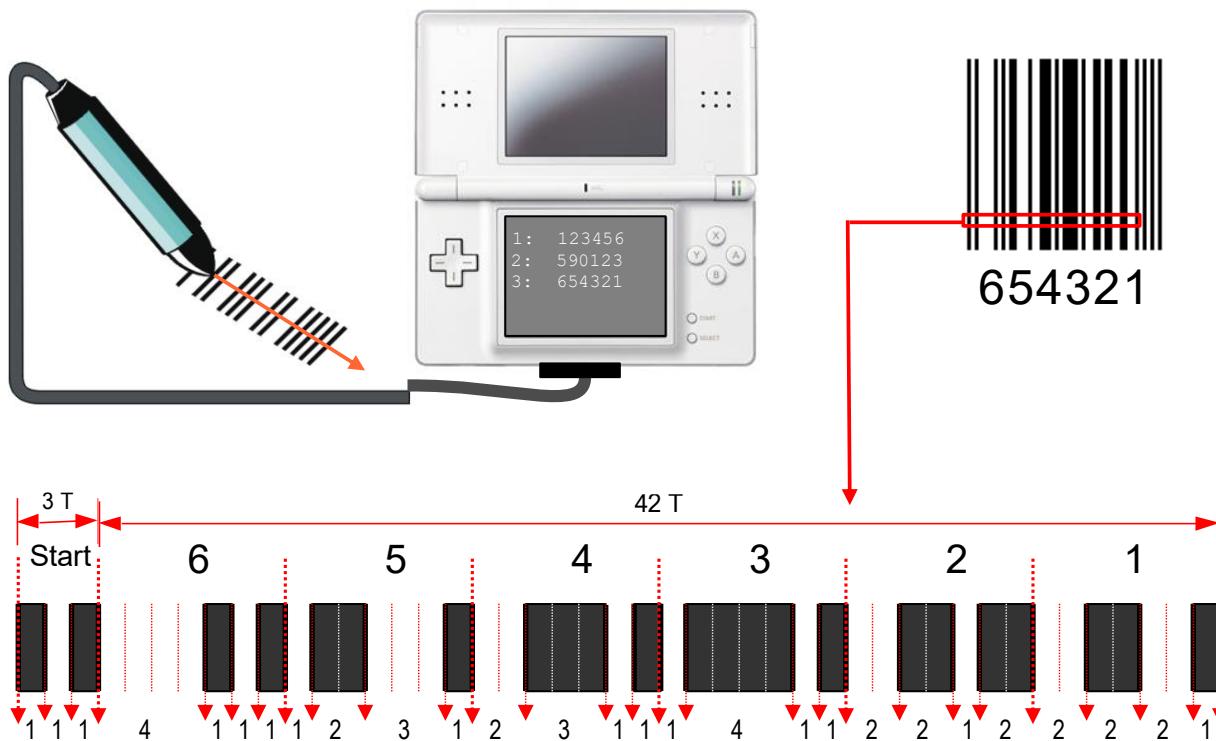
A la llamada de la rutina, los valores de entrada (tiempo, flujo) se deben almacenar en R0 y R1. Al retorno de la llamada, los mismos registros contendrán las coordenadas en píxeles (px, py). Hay que tener en cuenta que el valor de tiempo se debe desplazar 20 píxeles a la derecha para ajustarse a la gráfica, mientras que el valor de flujo se debe multiplicar por 6 y restar de 180.

Se pide:

Programa principal en C, RSI del *timer 0* y rutina `convertir_punto()` en ensamblador.

Problema 11 (1^a Conv. 2012-13): Lector de códigos de barras

Se propone trabajar con un lector de códigos de barras de tipo “lápiz”:



Este dispositivo periférico no presenta ningún registro, simplemente, generará una interrupción específica (IRQ_CART) cada vez que su haz de luz detecte un cambio de intensidad, es decir, cada paso de claro a oscuro o de oscuro a claro. Estos cambios se producirán cuando el usuario pase el lápiz por encima del código de barras, a una velocidad más o menos constante. En el gráfico del ejemplo, cada interrupción está indicada con una flecha hacia abajo.

Todos los códigos empiezan por una marca de inicio o *Start*, que son dos barras negras separadas por un espacio en blanco. Cada uno de estos tres elementos (2 barras + 1 espacio) presenta una anchura de referencia que llamaremos **anchura unitaria**.

A continuación aparecen las barras y espacios que codifican los dígitos del número del código de barras. Cada dígito se codifica con dos espacios y dos barras, donde cada elemento puede presentar una anchura de 1, 2, 3 o 4 veces la anchura unitaria. La suma de las anchuras de los cuatro elementos de cada dígito siempre será igual a 7 veces la anchura unitaria. Para simplificar, vamos a suponer que siempre se leerán números de 6 dígitos.

La anchura de cada barra o espacio se convertirá en un tiempo (absoluto) según la velocidad de movimiento del lápiz. El programa a realizar tendrá que obtener los tiempos absolutos correspondientes a las anchuras de las barras y espacios. Denominaremos T al tiempo absoluto que corresponderá a la anchura unitaria. Dividiendo los tiempos absolutos por T , obtendremos unos tiempos relativos para cada barra y cada espacio, que siempre serán valores entre 1 y 4, independientemente de la velocidad del lápiz.

Por ejemplo, la secuencia del gráfico anterior tendrá que proporcionar los siguientes tiempos relativos:

{1, 1, 1, 4, 1, 1, 1, 2, 3, 1, 2, 3, 1, 1, 1, 4, 1, 1, 2, 2, 1, 2, 2, 2, 2, 1}

Se dispone de las siguientes rutinas, ya implementadas:

Rutina	Descripción
inicializaciones ()	Inicializa pantalla e interrupciones
tareas_independientes ()	Tareas que no dependen de la lectura del código de barras actual (ej. enviar por Wifi el último código)
swiWaitForVBlank ()	Espera el retroceso vertical
cpuStartTiming (int timer)	Inicia un cronómetro de precisión usando el <i>timer</i> que se pasa por parámetro (0-2) y el siguiente <i>timer</i>
cpuGetTiming ()	Devuelve el conteo de tics del cronómetro desde que se inició (entero de 32 bits)
decodificar_codigo (char tiempos [])	Decodifica un código de barras de 6 dígitos a partir de los tiempos relativos y lo escribe en pantalla

Para obtener el tiempo absoluto de las barras y los espacios hay que utilizar la rutina `cpuStartTiming()` en una interrupción y la rutina `cpuGetTiming()` en la siguiente interrupción, la cual retornará el número de tics transcurridos entre las dos interrupciones. Estas rutinas tardan menos de 5 microsegundos.

Los tics se incrementan a la frecuencia base de la NDS ($\approx 33,5$ MHz). Si admitimos velocidades del lápiz entre 5 cm/s y 100 cm/s, los tiempos absolutos de la anchura unitaria oscilarán entre los 220.120 y los 11.060 tics. Los tiempos absolutos de toda la secuencia (incluida la marca de inicio) se almacenarán en un vector *y*, cuando se hayan obtenido todos, se tendrá que realizar su división por el tiempo unitario *T*, utilizando la rutina BIOS `swi 9` (entrada: R0 = numerador, R1 = divisor; salida: R0 = cociente, R1 = resto, R3 = cociente absoluto). Cada división puede tardar entre 5 y 20 μ s. Todas las divisiones se tienen que realizar en una rutina que almacenará los tiempos relativos en otro vector (acceso por referencia):

```
void normalizar_tiempos(int t_abs[], char t_rel[]);
```

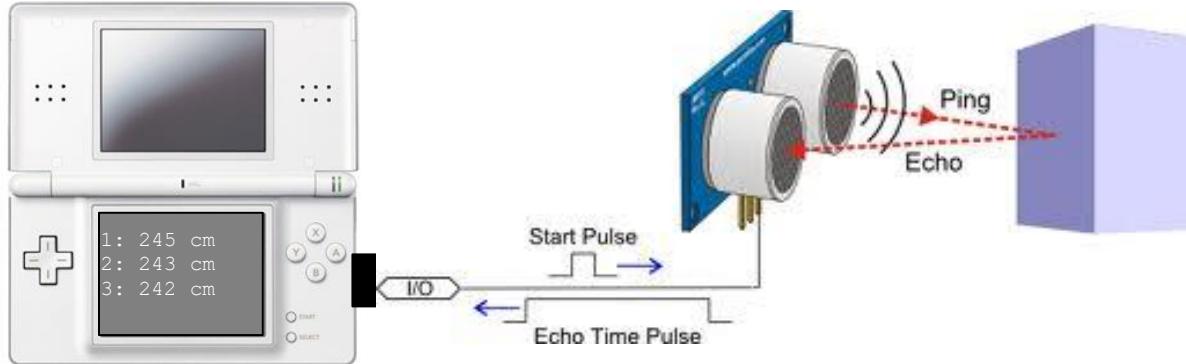
Después se tendrán que decodificar los tiempos relativos obtenidos y escribir los dígitos correspondientes, invocando a la rutina `decodificar_codigo()`.

Se pide:

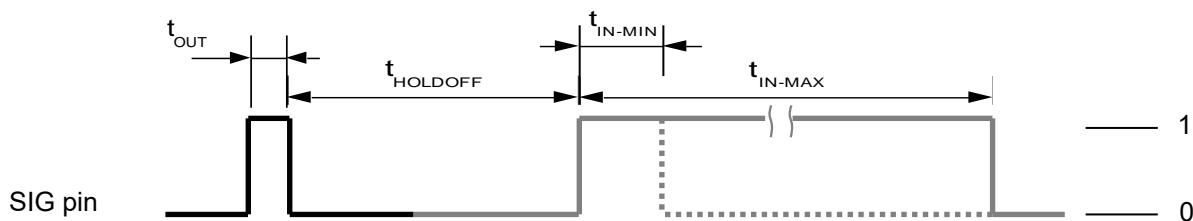
Programa principal en C, RSI del lector y rutina `normalizar_tiempos()` en ensamblador.

Problema 12 (1^a Conv. 2013-14): Sensor de distancia

Se propone controlar con la NDS un sensor de distancia por ultrasonidos PING)))TM, de la empresa *Parallax*^(R):



Cuando el computador envía un pulso eléctrico de inicio (*Start Pulse*, t_{OUT}) por el único cable de datos disponible (*SIG pin*), el dispositivo PING))) emite una pequeña ráfaga de impulsos ultrasónicos, los cuales rebotan en los objetos cercanos en forma de ecos. Después de emitir la ráfaga ($t_{HOLDOFF}$), el dispositivo activa *SIG pin* (1) hasta que detecta el primer eco, momento en el cual desactiva *SIG pin* (0). A continuación se muestra un cronograma esquemático del proceso de emisión-recepción de la ráfaga de ultrasonidos:



—	Host	<i>Start Pulse</i>	t_{OUT}	2 μ s (min), 5 μ s (típico)
—	PING)))	<i>Echo Holdoff</i>	$t_{HOLDOFF}$	750 μ s
		<i>Echo Time Pulse Minimum</i>	t_{IN-MIN}	115 μ s
		<i>Echo Time Pulse Maximum</i>	t_{IN-MAX}	18.5 ms
		<i>Delay before next measurement</i>		200 μ s (min)

Midiendo el tiempo del pulso que genera el dispositivo (*Echo Time Pulse*, t_{IN}) es posible saber la distancia del objeto más cercano. Suponiendo una velocidad del sonido de 340 m/s, el tiempo mínimo (t_{IN-MIN}) corresponderá a 2 cm, el tiempo máximo (t_{IN-MAX}) corresponderá a 315 cm, y un tiempo intermedio corresponderá a una distancia proporcional entre estos dos

valores. Si el primer objeto estuviera a menos de 2 cm o a más de 315 cm, el dispositivo generaría el pulso de tiempo mínimo o el de tiempo máximo, respectivamente.

El dispositivo se conectará a la NDS mediante el puerto de cartuchos de juegos GBA ROM. La señal *SIG pin* se podrá leer y escribir a través del bit 0 del registro 0x040001A2. Además, cada vez que este bit pase de 1 a 0, se activará la interrupción 13 (IRQ_CART), ya sea cuando el bit lo modifica la propia NDS (pulso de inicio) o cuando el bit lo modifica el dispositivo (pulso de tiempo de eco).

El programa de control, además de realizar ciertas tareas independientes, debe generar el pulso de inicio, detectar el tiempo de eco, calcular la distancia al objeto más cercano en función de dicho tiempo y escribir por pantalla dicha distancia, todo ello continua e indefinidamente. Delante de cada distancia se indicará su número de medida (ver pantalla de la NDS en el primer gráfico).

Se dispone de las siguientes rutinas, ya implementadas:

<i>Rutina</i>	<i>Descripción</i>
<code>inicializaciones ()</code>	Inicializa el <i>Hardware</i> (pantalla, interrupciones, etc.)
<code>tareas_independientes ()</code>	Tareas que no dependen del cálculo de distancias (ej. control del movimiento de un robot); tiempo de ejecución < 1 s
<code>swiWaitForVBlank ()</code>	Espera hasta el próximo retroceso vertical
<code>cpuStartTiming (int timer)</code>	Inicia un cronómetro de precisión usando el <i>timer</i> que se pasa por parámetro (0, 1 o 2) y el siguiente <i>timer</i>
<code>cpuGetTiming ()</code>	Devuelve el contaje de tics del cronómetro desde que se inició (entero de 32 bits)
<code>printf (char *format, ...)</code>	Escribe por pantalla la información especificada
<code>startPulse ()</code>	Genera un pulso de inicio de 5 microsegundos

Para poder contar tiempo con precisión hay que utilizar las rutinas `cpuStartTiming ()` y `cpuGetTiming ()`, las cuales permiten controlar dos *timers* encadenados que contarán tics a la frecuencia base de la NDS (\approx 33,5 MHz). Estas dos rutinas tardan menos de 5 microsegundos en ejecutarse.

Para poder realizar el programa de control sin tener que usar la calculadora, a continuación se muestra la equivalencia entre los tiempos de referencia del cronograma y el número de tics correspondiente:

t_{OUT} :	5 μs	\approx	168 tics;
$t_{HOLDOFF}$:	750 μs	\approx	25.135 tics;
t_{IN-MIN} :	115 μs	\approx	3.854 tics;
t_{IN-MAX} :	18,5 ms	\approx	620.009 tics;
retardo mínimo entre mediciones:	200 μs	\approx	6.703 tics;

El cálculo de la distancia se tendrá que implementar dentro de una rutina escrita en lenguaje ensamblador, que recibirá por parámetro el número de tics correspondientes al tiempo del pulso de eco (t_{IN}) y devolverá la distancia correspondiente, en centímetros:

```
int calcular_distancia(int t_in);
```

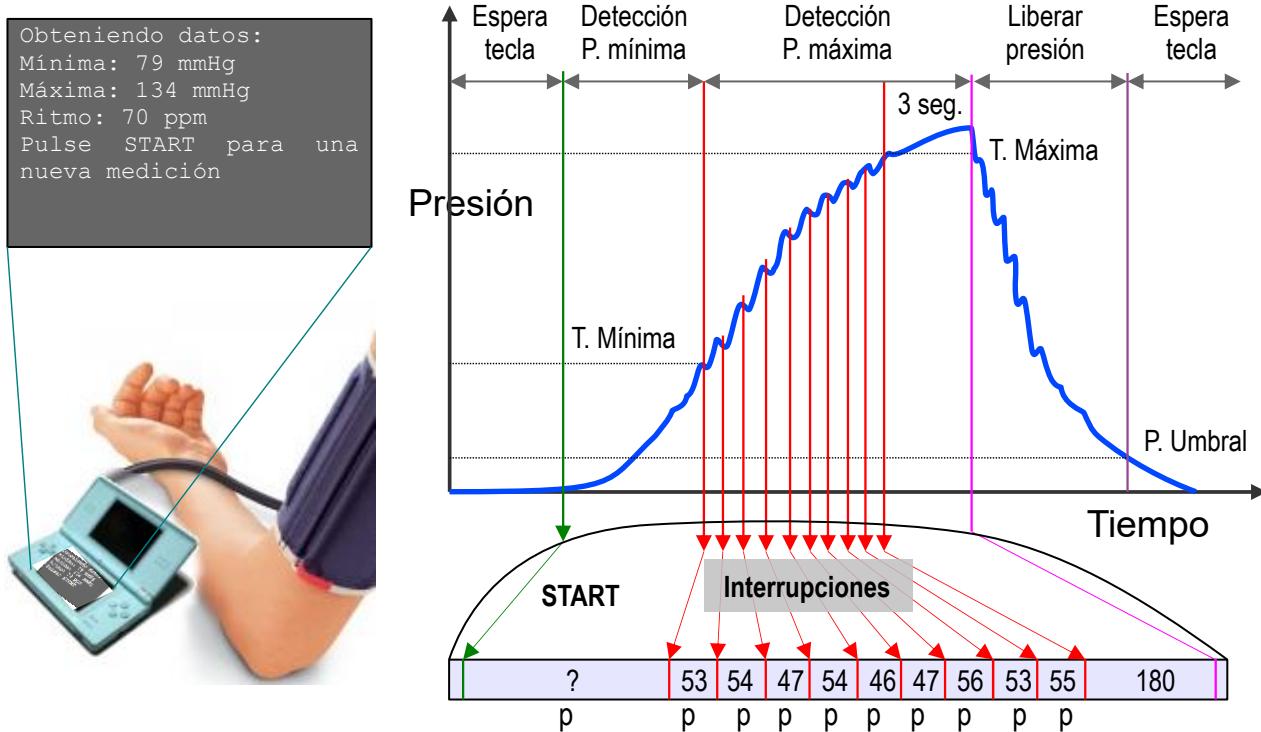
Para realizar las divisiones que sean necesarias se tendrá que utilizar la rutina BIOS swi 9 (entrada: R0 = numerador, R1 = divisor; salida: R0 = cociente, R1 = resto, R3 = cociente absoluto). Cada división puede tardar entre 5 y 20 microsegundos.

Se pide:

Programa principal en C, RSI del dispositivo y rutina `calcular_distancia()` en ensamblador.

Problema 13 (2^a Conv. 2013-14): Tensiómetro

Se propone controlar con la NDS un dispositivo para medir la tensión arterial de una persona:



El funcionamiento del sistema será el siguiente:

1. Inicializar el sistema.
2. Presentar mensaje de inicio: "Pulse START para iniciar la medición".
3. Detectar pulsación de la tecla 'START'.
4. Borrar pantalla y presentar mensaje: "Obteniendo datos:".
5. Iniciar el ciclo de aumento de presión: cerrar la válvula del dispositivo y activar un motor que insufla aire dentro del brazalete neumático.
6. Cuando haya suficiente presión, el dispositivo empezará a detectar latidos del corazón; la tensión mínima será la presión del brazalete al detectar el primer latido.
7. Se continuará aumentando la presión hasta que no se detecten más latidos; además, durante esta fase del proceso se debe calcular el tiempo entre cada dos latidos, almacenándolo en un vector de períodos ($p_0, p_1, p_2, \dots, p_{n-1}$, $n < 50$) para después poder calcular el ritmo cardíaco.
8. Cuando el tiempo desde que se detectó el último latido sea superior a 3 segundos se considerará que se ha superado la presión máxima; la tensión máxima será la presión del brazalete al detectar el último latido.
9. Presentar por pantalla los resultados: tensión mínima, tensión máxima y ritmo cardíaco.
10. Iniciar ciclo de disminución de presión: abrir la válvula del dispositivo, parar el motor y esperar a que la presión del brazalete llegue a un valor umbral

mínimo (10 mmHg).

11. Presentar mensaje de continuación: “Pulse START para una nueva medición” y repetir todo el proceso desde el paso 3.

Este dispositivo periférico presenta dos registros de Entrada/Salida de 16 bits:

- TENS_CTRL: 1 → cierra válvula y activa motor para insuflar aire,
 0 → abre válvula para liberar el aire y para motor.
- TENS_DATA: valor actual de la presión del brazalete, expresada en
 mmHg (milímetros de mercurio).

En el ciclo de aumentar la presión, cada vez que el dispositivo detecta un latido genera una interrupción específica (IRQ_CART).

Se dispone de las siguientes rutinas, ya implementadas:

<i>Rutina</i>	<i>Descripción</i>
inicializaciones ()	Inicializa pantalla e interrupciones
scanKeys ()	Captura el estado actual de los botones de la NDS
int keysDown ()	Devuelve un patrón de bits con los botones activos
swiWaitForVBlank ()	Espera hasta el próximo retroceso vertical
clear ()	Borra todo el contenido de la pantalla
printf(char *format, ...)	Escribe por pantalla un mensaje con un formato específico

Para obtener el tiempo entre los latidos del corazón hay que utilizar la rutina swiWaitForVBlank() y una variable global de tiempo que se incremente a cada retroceso vertical. Esto proporcionará un valor de período entre dos latidos, si en cada interrupción del tensiómetro se vuelve a poner a cero el tiempo actual. Como puede haber pequeñas variaciones de tiempo entre los latidos detectados, se pide que se almacenen dichos períodos en un vector para posteriormente obtener un valor promedio de todos los períodos.

Dicho valor promedio estará expresado en retrocesos verticales. Por lo tanto, hay que realizar las conversiones oportunas para obtener el ritmo cardíaco del usuario en pulsaciones por minuto (frecuencia).

Suponiendo que dicho ritmo nunca será inferior a 20 ppm ni superior a 200 ppm, los valores límite del período promedio serán 180 y 18 retrocesos verticales, respectivamente.

Todos los cálculos del período promedio y del ritmo cardíaco se encapsularán en la siguiente rutina:

```
unsigned char calcular_ritmo(unsigned char periodos[],  
                           unsigned char n_elem);
```

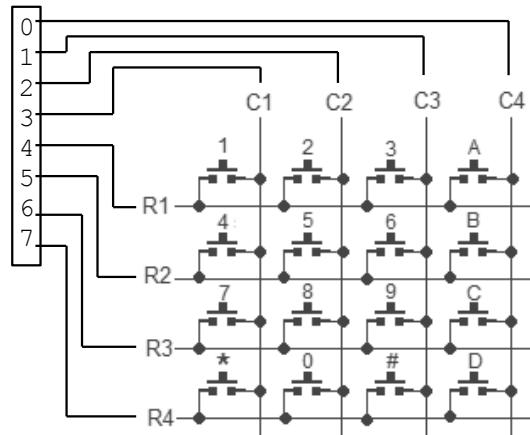
donde el primer parámetro corresponderá al vector de todos los períodos obtenidos (paso por referencia) y el segundo parámetro será el número de elementos registrados en el vector. El resultado que devolverá la rutina será el ritmo cardíaco expresado en pulsaciones por minuto. Para realizar divisiones desde lenguaje ensamblador hay que usar la rutina BIOS swi 9 (entrada: R0 = numerador, R1 = divisor; salida: R0 = cociente, R1 = resto, R3 = cociente absoluto).

Se pide:

Programa principal en C, RSI y rutina `calcular_ritmo()` en ensamblador.

Problema 14 (1^a Conv. 2014-15): Teclado numérico

Se propone controlar un teclado numérico de 16 teclas con la NDS. El teclado tiene una disposición matricial de los interruptores (teclas), en 4 filas por 4 columnas:



El dispositivo se conectará a la NDS mediante el puerto de cartuchos de juegos GBA ROM, y se controlará con un único registro de Entrada/Salida de 16 bits en la dirección simbólica REG_TECL, aunque solo los 8 bits de menos peso estarán conectados a la matriz de contactos, según indica el esquema de la figura anterior (bit 0 → C4, bit 1 → C3, etc.).

El registro es de lectura/escritura. La forma de leer las teclas será por barrido de filas y detección de columnas. Esto significa que, para cada fila, hay que realizar los siguientes pasos:

- escribir el registro REG_TECL, fijando todos los bits de filas (b7..b4) a 1 excepto el bit de la fila a analizar, que debe estar a 0,
- al cabo de cierto tiempo (del orden de centésimas de segundo), leer el registro REG_TECL, el cual presentará en cada bit de las columnas (b3..b0) un 0 si el interruptor correspondiente está pulsado, o un 1 si no lo está.

Después de analizar la última fila se debe volver a empezar por la primera, efectuando un barrido de todo el teclado a una frecuencia de 10 Hercios.

El programa de control debe consultar el estado de las teclas periódicamente y procesar cada pulsación para implementar una calculadora digital, escribiendo por pantalla los datos, operaciones y resultados generados.

Se dispone de las siguientes rutinas, ya implementadas:

Rutina	Descripción
inicializaciones ()	Inicializa el <i>Hardware</i> (pantalla, interrupciones, <i>timer</i>)
swiWaitForVBlank ()	Espera hasta el próximo retroceso vertical
processKey (char key)	Procesa la tecla que se le pasa por parámetro y realiza la función de calculadora, mostrando la información por pantalla; tiempo de ejecución < 0,01 s

Para poder capturar la pulsación de las teclas de forma concurrente con el programa principal, se pide utilizar la RSI del *timer* 0, que se programará (por la rutina de inicializaciones) para realizar 40 interrupciones por segundo.

En esta RSI hay que controlar el barrido de las 4 filas del teclado y guardar en una variable global, de nombre `currentKey`, un código numérico correspondiente a la tecla pulsada. Dicho código empezará por 0 para la tecla superior-izquierda (tecla “1”) y se incrementará de izquierda a derecha y de arriba a abajo del teclado. Si no hay ninguna tecla pulsada, la variable contendrá un -1. Si hay varias teclas pulsadas al mismo tiempo, sólo se almacenará el código de la tecla que tenga el número más grande (prioridad alta).

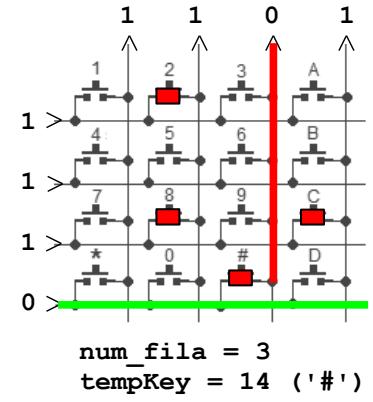
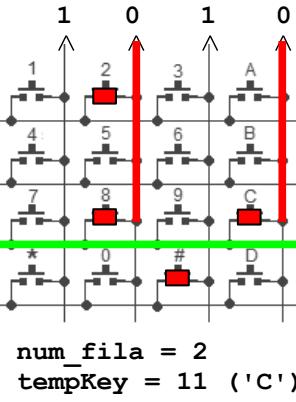
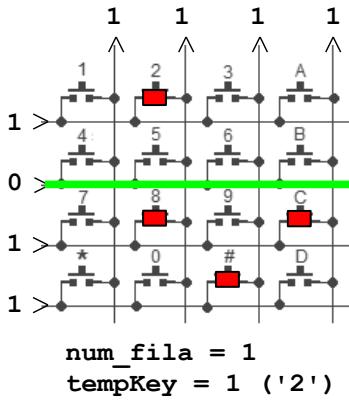
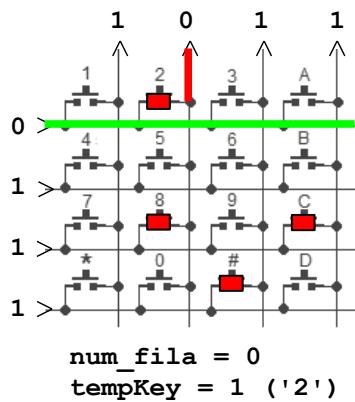
La RSI deberá llamar a una rutina auxiliar que se encargará de convertir los bits de columnas del registro REG_TECL, pasados por parámetro junto con el número de fila actual, en el código numérico de la tecla más prioritaria (código mayor) que se esté pulsando en dicha fila, o -1 si no existe pulsación en dicha fila:

```
char descodificar_tecla(char regteclval, char num_fila);
```

Por su parte, el programa principal debe sincronizarse con la RSI a través de la variable global `currentKey`, por el método de encuesta desde interrupción periódica, es decir, que se deben poder ejecutar las tareas independientes mientras se procesan las teclas. Además, para evitar que una misma pulsación se interprete como varias, será necesario detectar un cambio en `currentKey` antes de procesar el nuevo código de tecla con `processKey()`.

Por último, hay que tener en cuenta que la variable `currentKey` solo se actualizará al final del barrido de todo el teclado, puesto que hay que contrastar la prioridad de las teclas de las distintas filas, para lo cual necesitaremos otra variable global de nombre `tempKey` para mantener el código de tecla mayor de todo el barrido, además de la variable global `num_fila` que indicará la fila actual de procesamiento.

A continuación se muestra un ejemplo de barrido del teclado suponiendo que se han pulsado cuatro teclas a la vez (“2”, “8”, “C” y “#”). Aunque es un caso improbable, sirve para mostrar los valores que se obtendrán en cada fila y cómo se debe actualizar la variable `tempKey`:

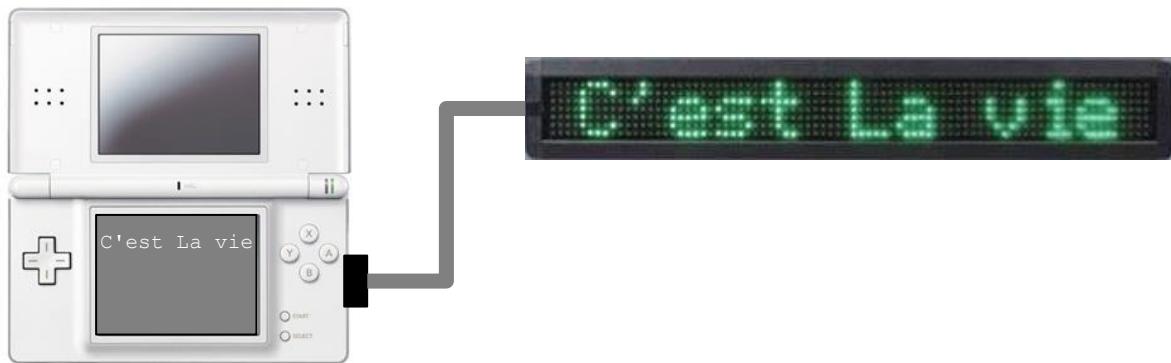


Se pide:

Programa principal en C, RSI del *timer* 0 y rutina `descodificar_tecla()` en ensamblador.

Problema 15 (2^a Conv. 2014-15): Display de LEDs

Se propone controlar un display de LEDs con la NDS. El display tiene 7 puntos de altura y 100 puntos de anchura, y los puntos a mostrar se insertan por la columna de la derecha, con un efecto de desplazamiento de todo el contenido hacia la izquierda, columna a columna (pixel a pixel):



El dispositivo se conectará a la NDS mediante el puerto de cartuchos de juegos GBA ROM, y se controlará con un único registro de Entrada/Salida de 16 bits en la dirección simbólica REG_DISP, aunque sólo los 8 bits de menor peso tendrán una funcionalidad específica:

- DATA (bits 6..0): deben contener el valor (0 → apagado, 1 → encendido) de los 7 puntos de una columna a introducir por la derecha, donde el bit 0 corresponde al punto superior y el resto de bits a los sucesivos puntos inferiores,
- STROBE (bit 7): se debe poner a 1 y después a 0 (mín. 2 ms entre cambios) para desplazar el contenido del display una columna a la izquierda, e insertar el estado de los puntos de la columna de más a la derecha según los bits de DATA.

La inserción de las columnas de puntos se debe realizar a una frecuencia de 18 Hercios, lo cual equivaldrá a una velocidad de 3 caracteres por segundo, dado que cada carácter está definido por 7 filas y 6 columnas de puntos (píxeles).

El programa principal debe realizar algunas tareas independientes, además de controlar el carácter a visualizar en todo momento, que se obtendrá de un *string* predeterminado (fijado dentro del programa), almacenado en un vector de códigos ASCII acabados con un carácter centinela '\0'.

Este *string*, que deberá tener al menos 17 caracteres, se tiene que visualizar indefinidamente, de modo que, cuando se llegue al carácter centinela, se empezará de nuevo por el primer carácter. El *string* también se debe mostrar por una pantalla de la NDS de una única vez, sin realizar ningún tipo de desplazamiento horizontal.

Se dispone de las siguientes rutinas, ya implementadas:

<i>Rutina</i>	<i>Descripción</i>
inicializaciones ()	Inicializa el <i>Hardware</i> (pantalla, interrupciones, <i>timer</i>)
tareas_independientes ()	Realiza tareas independientes a la visualización del <i>string</i> (ej. obtener la temperatura ambiente)
swiWaitForVBlank ()	Espera hasta el próximo retroceso vertical
printf(char *format, ...)	Escribe un mensaje por la pantalla inferior de la NDS

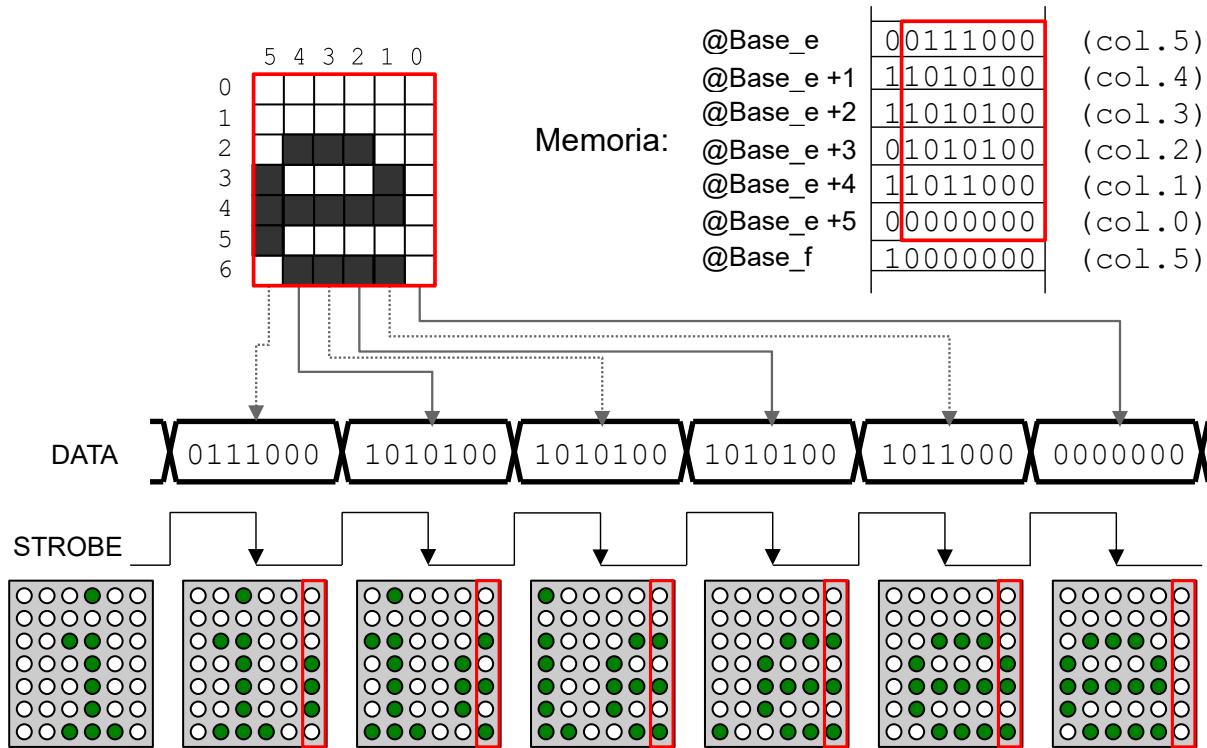
Para poder transferir las columnas de puntos de cada carácter de forma concurrente con el programa principal, se debe utilizar la RSI del *timer* 0, que se programará (por la rutina de inicializaciones) para realizar 36 interrupciones por segundo.

Esta RSI accederá al carácter actual a través de una variable global `currentChar`, y gestionará la columna actual a visualizar mediante otra variable global `num_col`. Además, tiene que llamar a una rutina auxiliar que hay que implementar, la cual retornará el estado de los puntos de una columna de un carácter, a partir del código ASCII del carácter y del número de columna actual que se pasarán por parámetro:

```
char obtener_puntos(char caracter, char num_columna);
```

Además de enviar el estado de los puntos, la RSI debe generar la señal de *strobe*, es decir, poner el bit 7 a 1 y, después de un cierto tiempo, poner el bit 7 a 0, para indicar al display que debe introducir una nueva columna.

A continuación se muestra la secuencia de introducción de la letra 'e' (detrás de una 'i'), así como el contenido en memoria que determina el estado de los puntos y el desplazamiento de las 6 columnas de puntos por la derecha (empezando por la columna de más a la izquierda):



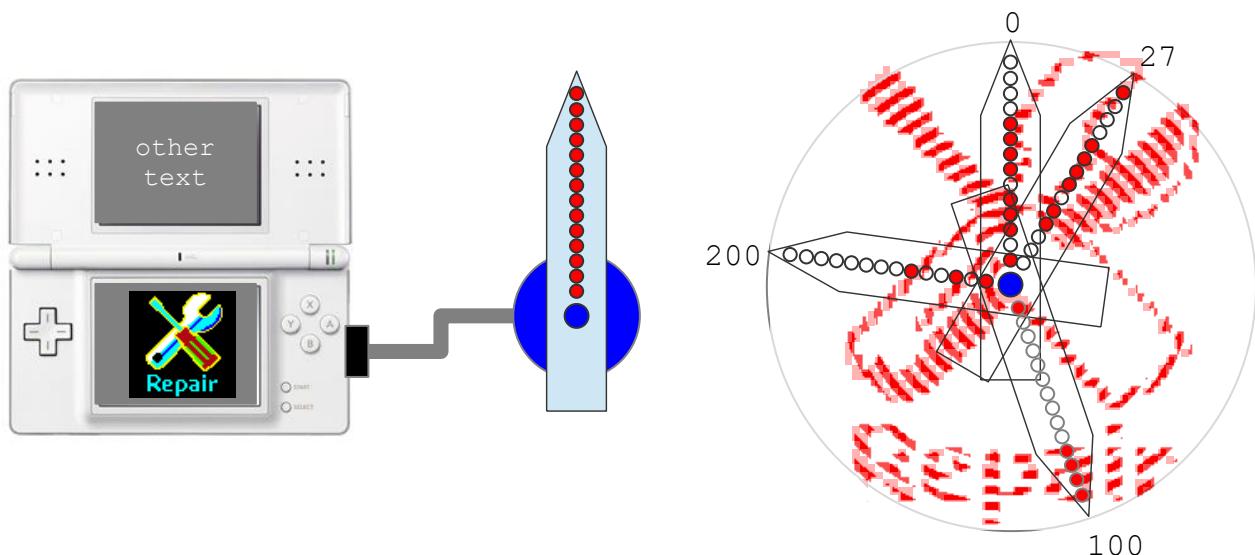
Hay que observar que cada carácter se almacenará a partir de una posición de memoria específica y ocupará 6 bytes consecutivos correspondientes al estado de sus 6 columnas, ocupando dicho estado los 7 bits de menor peso de cada byte, mientras que el bit de más peso del byte puede contener cualquier valor. Los datos de todos los caracteres se almacenarán consecutivamente a partir de una dirección base de memoria, cuyo nombre simbólico será `base_ASCII`. El primer carácter almacenado corresponderá al código ASCII 32 (espacio en blanco).

Se pide:

Programa principal en C, RSI del *timer 0* y rutina `obtener_puntos()` en ensamblador.

Problema 16 (1^a Conv. 2015-16): Propeller display

Se propone controlar un display de rotación con la NDS. Este dispositivo es un circuito impreso con una línea de LEDs, montado sobre un motor que lo hace girar a una velocidad más o menos constante. El programa de control debe encender y apagar los LEDs según el ángulo de giro de la línea, de modo que se visualicen los puntos de una imagen correspondientes a dicho ángulo. Si el motor gira lo suficientemente rápido (>25 rev./s), se podrá observar la imagen “en el aire”, gracias a la persistencia de la luz en la retina del ojo humano. A continuación se muestra un esquema de la estructura del dispositivo y su funcionamiento:



En el esquema anterior se ha representado un circuito impreso con 14 LEDs, y la activación de dichos LEDs en cuatro ángulos diferentes, así como una simulación del efecto del display “en el aire” para la imagen de ejemplo, la cual se muestra en la pantalla inferior de la NDS.

El sistema real a controlar dispone de 32 LEDs. Además, la circunferencia se divide en 256 fracciones, de modo que el rango del valor del ángulo será de 0 a 255. En la figura anterior se muestra la posición de la línea en los ángulos 0, 27, 100 y 200.

El dispositivo dispone de dos registros de Entrada/Salida:

- RDISP_STATUS: registro de 16 bits, del cual solo se utilizará el bit 0, que el dispositivo activará durante un pulso de $156\ \mu s$ cada vez que el motor pase por el ángulo 0,
- RDISP_DATA: registro de 32 bits, que el programa debe fijar para indicar el estado de cada uno de los 32 LEDs (0: apagado, 1: encendido), donde el LED más exterior corresponde al bit de más peso.

El programa de control debe recibir imágenes “rectangulares” (64x64 píxeles) desde el interfaz wifi de la NDS, convertir cada imagen recibida a su correspondiente imagen “circular”, y transferir continuamente el estado de los LEDs al display de rotación, según el

color de los *áxeles* de la imagen circular (*áxel* = *angular pixel*) y el ángulo de la línea de LEDs en cada instante, además de ajustar a cero el ángulo actual según el bit 0 del registro de estado del dispositivo.

Se dispone de las siguientes rutinas, ya implementadas:

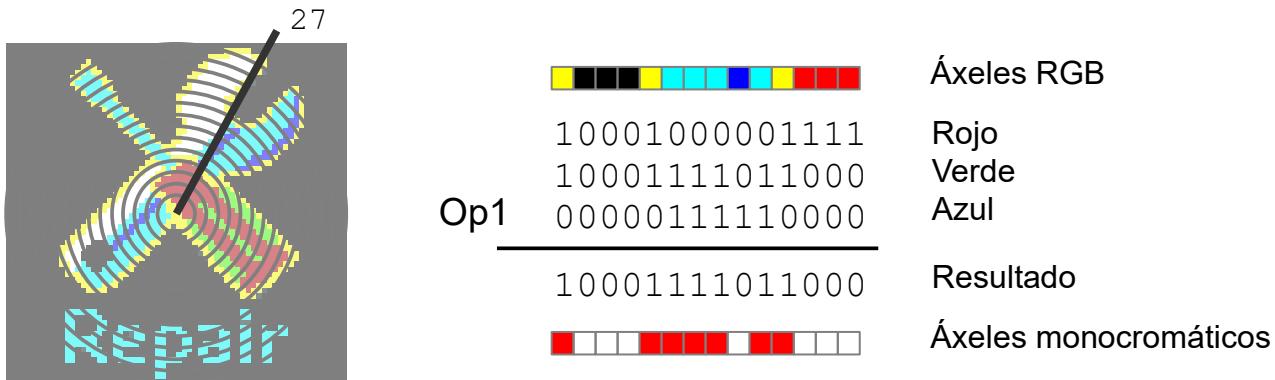
Rutina	Descripción
inicializaciones ()	Inicializa el <i>hardware</i> (pantalla, wifi, interrupciones, <i>timers</i> , etc.)
int wifiReceiveImage (unsigned char rimg[])	Rutina de recepción de una imagen rectangular por el interfaz wifi de la NDS; si hay una nueva imagen (desde la última llamada), copiará los píxeles de dicha imagen en el vector que se pasa por referencia y devolverá 1 (t. ejecución \approx 150 ms); si no hay nueva imagen, devolverá 0 (t. ejecución \approx 5 μ s)
convertirImagen (unsigned char rimg[], unsigned int cimg[])	Rutina de conversión de una imagen rectangular a la correspondiente imagen circular, determinando el color de los 32 <i>áxeles</i> de los 256 ángulos (tarda unos 30 ms)
swiWaitForVBlank ()	Espera hasta el próximo retroceso vertical
mostrarImagen (unsigned char rimg[])	Muestra la imagen rectangular que se pasa por parámetro en la pantalla inferior de la NDS

Para poder representar la imagen circular actual de forma concurrente con la recepción de nuevas imágenes, se pide utilizar la RSI del *timer* 0, que se programará (por la rutina de inicializaciones) para realizar 6.400 (25*256) interrupciones por segundo.

En esta RSI hay que controlar el barrido de los 256 radios (ángulos) de la circunferencia, leyendo los colores de los 32 *áxeles* de cada radio. Hay que tener en cuenta que una imagen circular, obtenida con la rutina `convertirImagen()`, almacenará el color de cada radio como tres *words* consecutivos, que contendrán el estado (binario) de los tres canales básicos de color (rojo, verde, azul, por este orden), donde el bit de más peso de cada *word* corresponderá al LED más exterior. La información de todos los 256 radios se almacenará consecutivamente en la memoria reservada para la imagen circular, a partir del radio cero.

La RSI deberá llamar a una rutina auxiliar que se encargará de convertir los tres bits de color de cada *áxel* en un bit para cada LED, ya que el dispositivo a controlar solo dispone de LEDs monocromáticos. Además de realizar la conversión, la RSI también debe transferir los bits resultantes al dispositivo.

La siguiente figura muestra un ejemplo de conversión del radio 27, aunque aquí solo se representa el color de 14 áxeles (para simplificar):



La conversión de los bits de color a bits monocromáticos se debe realizar dentro de la siguiente rutina:

```
void transferir_radio(unsigned int cim[], int num_radio);
```

Esta rutina recibe la dirección de memoria inicial de la imagen circular, así como el número de radio (ángulo) que se tiene que procesar y enviar al display de rotación. Además, esta rutina debe ser capaz de realizar dos tipos de procesado, según el estado del botón 'SELECT' de la NDS (bit 2 del registro REG_KEYINPUT):

- SELECT = 1 (soltado): cada bit monocromático valdrá 1 si el bit correspondiente del canal verde está a 1 y uno de los bits de los canales rojo y azul también está a 1, pero no los dos a la vez,
- SELECT = 0 (pulsado): cada bit monocromático valdrá 1 si alguno de los bits correspondientes de los tres canales de color vale 1.

El cálculo de los 32 bits monocromáticos se debe realizar aplicando las operaciones lógicas que correspondan (and, or, xor, not).

Es importante también realizar la puesta a cero del ángulo actual cada vez que se detecte el pulso de ángulo cero emitido por el display de rotación, puesto que la velocidad de rotación del motor puede tener pequeñas variaciones momentáneas de velocidad ($\pm 3\%$).

Además, hay que tener en cuenta que por la wifi podremos recibir hasta 25 imágenes rectangulares por segundo, aunque pueden ser menos, incluso puede que se envíe una única imagen para toda la sesión. En cualquier caso, por el display de rotación se deberá mostrar continuamente la última imagen circular recibida (imagen actual), mientras que, concurrentemente, puede que se reciban nuevas imágenes por la wifi.

Para el almacenamiento de imágenes se propone usar las siguientes estructuras de datos:

```
unsigned char rect_img[64*64];  
unsigned int circ_img[2][256*3];
```

El espacio para 2 imágenes circulares en `circ_img[2][256*3]` permitirá aplicar la técnica del doble buffer, de modo que la recepción y conversión de las nuevas imágenes rectangulares enviadas por wifi no interfiera con la visualización de la imagen circular actual.

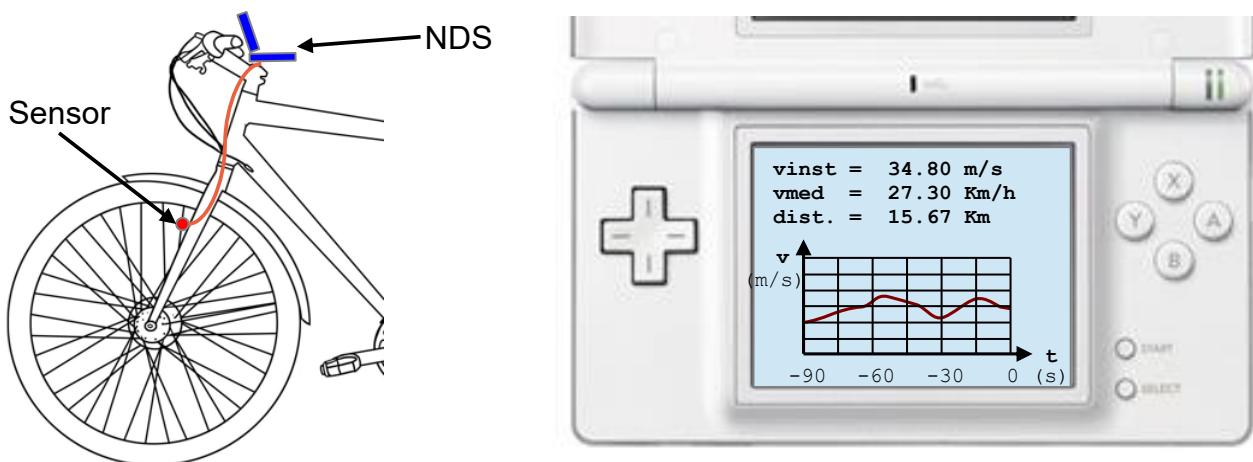
Por último, se puede considerar que, inicialmente, el contenido de los *buffers* de imagen está a cero, de modo que la visualización en el display de rotación de dicho contenido no activará ningún LED, aunque el motor girará a partir del primer instante que se encienda todo el sistema (NDS y dispositivo).

Se pide:

Programa principal y variables globales en C, RSI del *timer 0* y rutina `transferir_radio()` en ensamblador.

Problema 17 (1^a Conv. 2015-16): Velocímetro para bicicletas

Se propone realizar un programa para calcular la velocidad instantánea, la velocidad media y la distancia recorrida por una bicicleta, a partir de la señal de un sensor instalado en la horquilla de la rueda delantera. A continuación se muestra un esquema del sistema y la visualización de los datos en la pantalla inferior de la NDS:



Mediante un cable, el sensor envía un pulso cada vez que un rayo de la rueda pasa por delante suyo. El cable está conectado a la NDS, de forma que el pulso genera una petición de interrupción por la línea IRQ_CART, la cual activará una rutina de servicio de interrupción específica (RSI_sensor). Con esta RSI se podrá estimar la distancia recorrida por la rueda en todo momento.

Por otro lado, será necesario controlar el tiempo con el fin de poder calcular velocidades. Para este propósito se propone utilizar la RSI del *timer 0*, que generará interrupciones a una frecuencia de 2 Hz. La elección de esta frecuencia es porque se requiere que los cálculos de velocidades y distancia se actualicen cada medio segundo. Sin embargo, debido al retardo que pueden introducir las tareas independientes, la actualización en pantalla de los resultados se podría demorar hasta un segundo; este comportamiento se considerará normal.

Se dispone de las siguientes rutinas, ya implementadas:

Rutina	Descripción
swi 9	Rutina de la BIOS para división entera; parámetros (R0: dividendo, R1: divisor); resultado (R0: cociente, R1: resto, R3: cociente absoluto)
inicializaciones()	Inicializa el <i>hardware</i> (pantalla, interrupciones, <i>timers</i> , etc.) y las variables globales

tareas_independientes()	Tareas que no dependen de los cálculos de velocidad y distancia (ej. monitorización del ritmo cardíaco del ciclista en la pantalla superior); t. ejecución entre 0,1 y 1 s
scanKeys()	Captura la pulsación actual de las teclas
int keysDown()	Devuelve el estado de las últimas teclas pulsadas
swiWaitForVBlank()	Espera hasta el próximo retroceso vertical
representarInfo(int vinst, int vmed, int dist, short data[], int index)	Muestra la información por pantalla inferior según el valor de los parámetros: velocidad instantánea (en cm/s), velocidad media (en dam/hora), distancia total (en cm), vector de velocidades instantáneas (en cm/s), índice posición actual del vector

Para poder realizar todos los cálculos, se proponen las siguientes variables globales:

```
unsigned short Perimetro;           // perímetro de la rueda (en cm)
unsigned char Nrayos;               // número de rayos de la rueda
unsigned short Drayos;              // número de rayos por segundo
unsigned short Vinst;                // velocidad instantánea (en cm/s)
unsigned int Vmed;                  // velocidad media (en dam/hora)
unsigned int Tdist;                 // distancia total (en cm)
unsigned int Ttiempo;               // tiempo total (en semisegundos)
unsigned char ind;                  // indice posición actual buffer Vinst
unsigned short buffVinst[180];
```

Los valores de `Perimetro` y `Nrayos` dependerán de las características de la rueda instalada; la función de inicialización cargará el valor correspondiente en estas variables, por ejemplo, `Perimetro = 207 cm`, `Nrayos = 32 rayos`.

El valor de `Drayos` deberá contar el número de rayos que se detectan por unidad de tiempo. Este valor permitirá calcular la velocidad instantánea, que se tiene que almacenar en `Vinst`, en centímetros por segundo.

Para calcular la velocidad media, almacenada en `Vmed` (en decámetros por hora), será necesario registrar la distancia total recorrida desde que se inició el programa, así como el tiempo total. La distancia se almacenará en `Tdist` (en cm), mientras que el tiempo se almacenará en `Ttiempo` (en semisegundos). Además, si se pulsa la tecla 'START' en cualquier momento, todos los contadores se deberán poner a cero.

Por último, también se pide que cada semisegundo se almacene la velocidad instantánea en un vector de 180 posiciones, cada vez en una posición diferente (`ind`), de forma circular, es decir, cuando se llegue a la última posición se debe empezar por la primera otra vez. Este vector, de nombre `buffVinst[]`, permitirá representar la evolución de la velocidad

instantánea en los últimos 90 segundos. El contenido de este vector también se deberá poner a cero al pulsar la tecla 'START'.

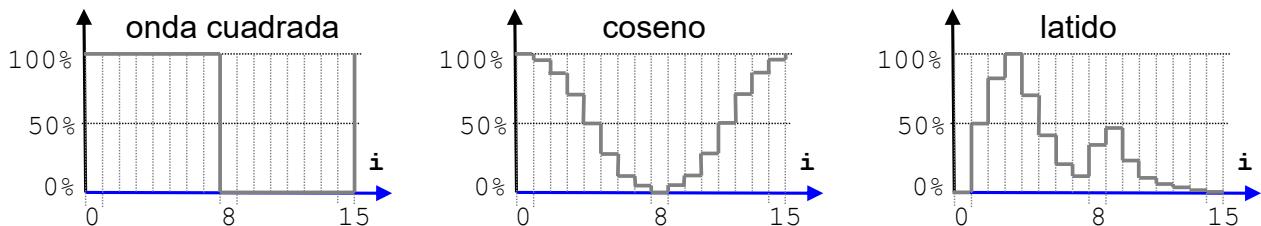
Toda esta información se debe pasar a la función `representarInfo()`, con las unidades indicadas para las variables globales. Internamente, esta función transformará dichas unidades a las habituales (m/s, Km/h, Km). Hay que observar que las variables globales propuestas utilizan fracciones de dichas unidades (cm/s, dam/h, cm) para no tener que operar con decimales, ya que el procesador ARM solo trabaja con valores enteros.

Se pide:

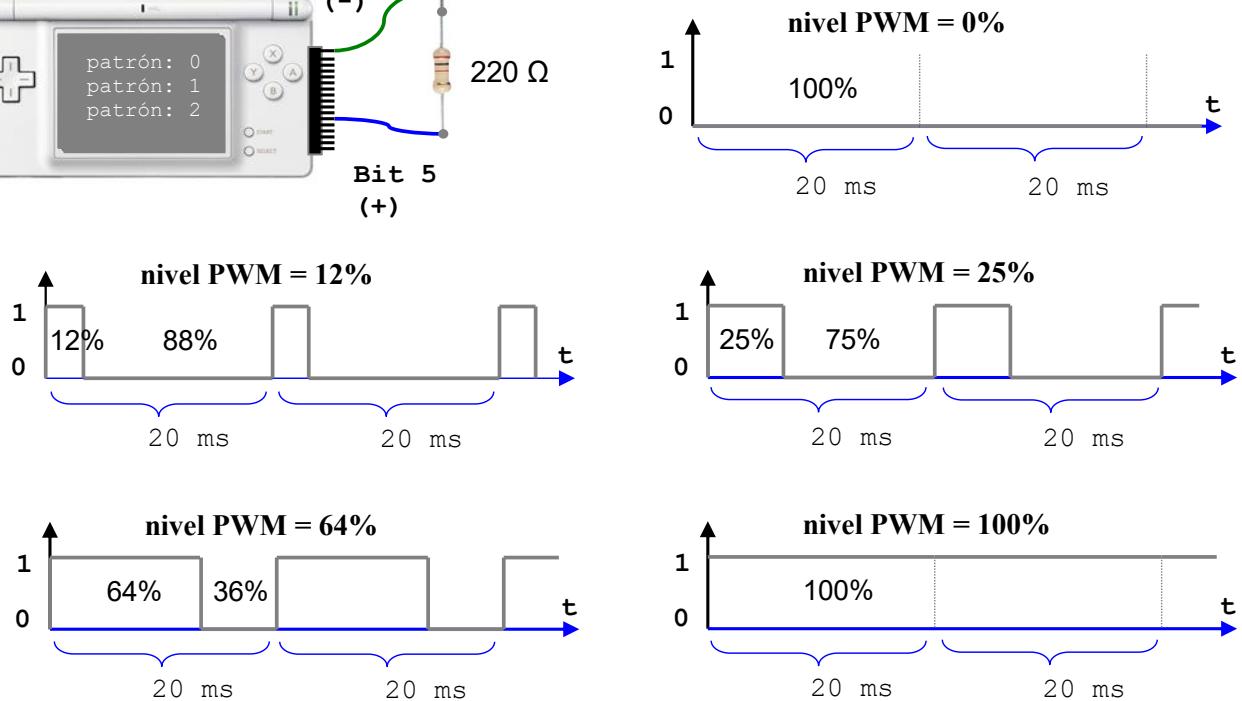
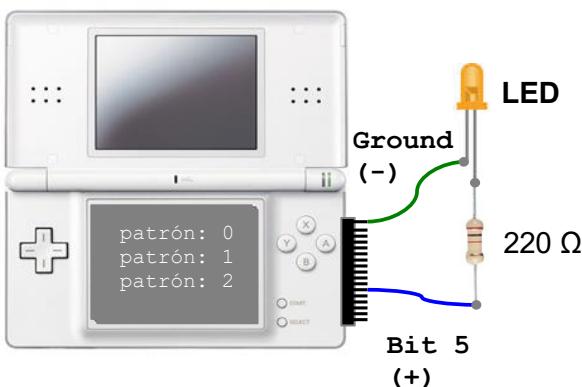
Programa principal y variables globales en C, RSI del *timer 0* y RSI del sensor en ensamblador.

Problema 18 (2^a Conv. 2015-16): Luz LED regulada por PWM

Se propone controlar la intensidad (o brillo) de una luz LED con la NDS, para que emita, repetidamente, una determinada secuencia de niveles del brillo. El programa a realizar permitirá emitir diferentes secuencias, que llamaremos *patrones*. A continuación se muestran tres patrones diferentes, con 16 valores de brillo por cada patrón (la coordenada horizontal indica el índice de cada valor):



Los niveles de brillo se codificarán como el tanto por ciento de la intensidad máxima que emite el LED. A continuación se muestra un esquema de la estructura del sistema, junto con unos cronogramas de ejemplo del control de brillo (nivel PWM):



Conectados a la NDS del esquema, se han representado los pins de salida de un dispositivo que permite generar 16 voltajes digitales, es decir, 0 o 5 voltios en cada pin, más un pin extra

de masa (*Ground*). El voltaje de los pins se controlará con el valor de los bits de un registro de salida de 16 bits, etiquetado como REG_DOUT, donde el valor del bit 0 indicará el voltaje el pin inferior, y el resto de bits se asignarán consecutivamente a los siguientes pines (hacia arriba). El pin superior es para conectar la masa de los circuitos.

El ánodo del LED (+) se ha conectado a una resistencia de 220 ohmios, y esta resistencia se ha conectado al pin del bit 5, mientras que el cátodo del LED (-) se ha conectado a masa. Sin embargo, con esta configuración solo podemos apagar el LED (bit 5 = 0) o encenderlo a su máxima intensidad (bit 5 = 1). Es decir, no es posible indicar directamente un porcentaje del brillo del LED usando únicamente el estado del bit 5 del registro de salida.

Para solucionar este problema se propone aplicar la técnica de modulación por ancho de pulso (PWM = *Pulse Width Modulation*), que consiste en encender y apagar el LED repetidamente, aplicando el valor 1 durante un porcentaje determinado del período del pulso, y el valor 0 durante el resto del período.

Los gráficos anteriores muestran diversas formas del pulso, con el ancho del valor 1 modulado según el nivel PWM (% de brillo) requerido. Si dicha forma del pulso se repite a una frecuencia alta, por ejemplo, 50 Hz, el ojo humano no percibe los cambios entre estados de encendido/apagado, sino que percibe diferentes intensidades de luz según el porcentaje del tiempo en que se está emitiendo luz.

El programa a implementar, además de controlar el ancho de pulso correspondiente al nivel de brillo actual del patrón seleccionado, tendrá que actualizar dicho nivel de brillo periódica y cíclicamente, es decir, debe pasar al siguiente valor del patrón cada cierto tiempo y, cuando llegue al último valor, volver a empezar por el primero. También se requiere que el usuario pueda seleccionar el patrón de entre un conjunto predefinido de patrones, pulsando el botón A de la NDS.

Se dispone de las siguientes rutinas, ya implementadas:

Rutina	Descripción
inicializaciones ()	Inicializa el <i>hardware</i> (pantalla, interrupciones, <i>timers</i> , etc.)
tareas_independientes ()	Tareas que no interfieren con la gestión del bit 5 del registro de salida, aunque pueden estar modificando algunos de los otros bits del mismo registro (ej. activar un altavoz externo); t. ejecución < 10 ms
scanKeys ()	Captura el estado actual de los botones de la NDS
int keysDown ()	Devuelve un patrón de bits con los botones activos

swiWaitForVBlank()	Espera hasta el próximo retroceso vertical
printf(char *format, ...)	Escribe un mensaje en la pantalla inferior de la NDS

Para generar la forma del pulso correspondiente al nivel PWM requerido, se pide utilizar la RSI del *timer* 0, que se programará (por la rutina de inicializaciones) para realizar 5.000 interrupciones por segundo (5 KHz). A esta frecuencia, se podrá controlar el % del periodo del pulso (20 ms) para cambiar los estados 1 y 0 del bit 5 del registro de salida, ya que se producirá una interrupción a cada centésima de dicho periodo (cada 200 µs).

Para cambiar el valor actual de brillo de cada patrón, se pide utilizar la RSI del *timer* 1, que se programará (por la rutina de inicializaciones) para realizar 16,6667 interrupciones por segundo (periodo de 60 ms). A esta frecuencia, cada valor de brillo estará visible durante 3 ciclos de PWM (aprox.). Para simplificar el diseño, no se exige que las dos RSIs estén sincronizadas.

Para realizar la gestión de los patrones se propone usar las siguientes estructuras de datos:

```
#define NUM_PATTERNS = 5      // número de patrones de brillo
#define NUM_VALUES = 16        // número valores de brillo por patrón

unsigned char patterns[NUM_PATTERNS][NUM_VALUES] = {
    //patrón cuadrado
    {100,100,100,100,100,100,100,0,0,0,0,0,0,0,0,0,0},
    //patrón triangular
    {100,88,75,63,50,38,25,13,0,13,25,38,50,63,75,88},   //patrón
rampa
    {0,6,13,19,25,31,37,44,50,56,63,69,75,81,88,94},
    //patrón coseno
    {100,96,85,69,50,31,15,4,0,4,15,31,50,69,85,96},
    //patrón latido
    {0,50,80,100,70,40,21,15,37,48,25,13,8,4,2,1}
};

unsigned char curr_pattern = 0; // índice del patrón actual
unsigned char curr_value = 0;   // índice del valor actual del pat.
unsigned char brightness = 0;  // nivel de brillo actual
```

En este ejemplo se han definido cinco patrones de 16 valores cada uno (no hace falta copiar los valores de ejemplo en la solución del examen).

La variable `curr_pattern` almacenará el índice del patrón seleccionado actualmente. Cada vez que se pulsa el botón **A**, esta variable debe actualizarse y mostrar su nuevo valor por la pantalla inferior de la NDS.

La variable `curr_value` almacenará el índice del nivel de brillo actual del patrón. Cada vez que cambie (por la RSI del *timer 1*), se copiará el contenido de la posición de la matriz `patterns[][]`, correspondiente a los índices de patrón y nivel actual, dentro de la variable `brightness`, que es la que consultará la RSI del *timer 0* para gestionar la modulación del brillo del LED por ancho de pulso.

Los símbolos `NUM_PATTERNS` y `NUM_VALUES` también estarán disponibles para el código fuente en ensamblador, definidos con las siguientes líneas:

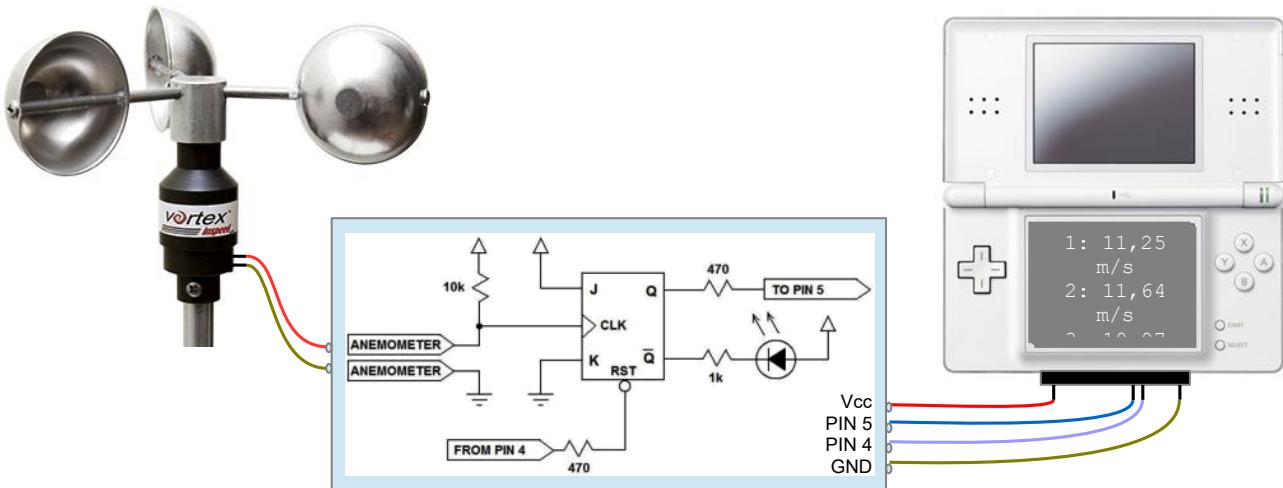
<code>NUM_PATTERNS = 5</code>	<code>@; número de patrones de brillo</code>
<code>NUM_VALUES = 16</code>	<code>@; número valores de brillo por patrón</code>

Se pide:

Programa principal y variables globales en C, RSI del *timer 0* y RSI del *timer 1* en ensamblador.

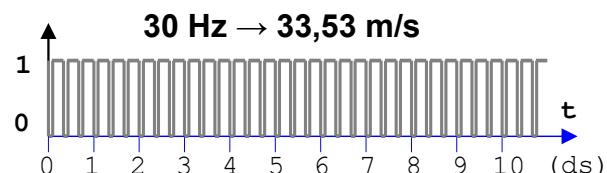
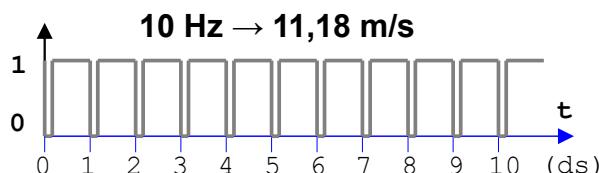
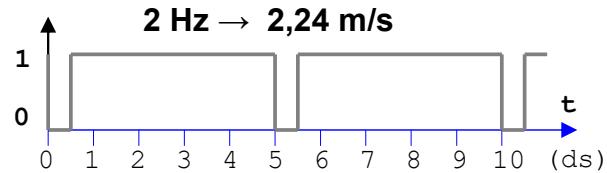
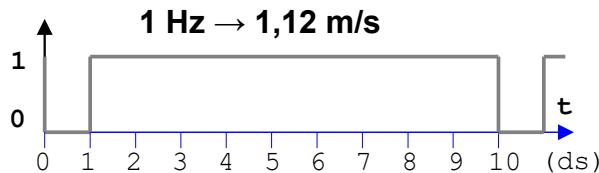
Problema 19 (1^a Conv. 2016-17): Anemómetro electrónico

Se propone conectar un anemómetro electrónico *Vortex wind sensor* (Inspeed.com) a la NDS, con el fin de medir la velocidad del viento. El anemómetro genera un contacto eléctrico entre sus dos terminales cada vez que el rotor, impulsado por el viento, da una vuelta completa. Como el contacto dura un tiempo variable (en función de la velocidad angular), se ha añadido un circuito electrónico basado en un biestable J-K para convertir dichos contactos en pulsos:



En el esquema anterior se han representado, además de los cables de los contactos del anemómetro y de la alimentación del circuito electrónico (Vcc y GND), los pines 4 y 5 de un adaptador de Entrada/Salida que permite a la NDS leer y escribir el estado de 32 pines, conectados a un registro de E/S de 32 bits, al cual se accede a través de la dirección absoluta de memoria `0xA000000`. El pin 5 permite leer el estado interno del J-K (salida Q), mientras que el pin 4 permite realizar un reset del J-K (~RST, señal negada).

Según el fabricante del anemómetro, la forma del cabezal hace que su frecuencia de rotación se incremente en 1 Hz por cada 2,5 mph (millas por hora) que acumule la velocidad del viento. Asumiendo que 2,5 mph equivalen a 1,1176 m/s (\approx 112 cm/s), los siguientes gráficos muestran diversos ejemplos de la forma de onda que obtendremos en la entrada CLK del biestable para 4 frecuencias de ejemplo, junto con sus velocidades del viento equivalentes:



Como se puede observar, el ancho del pulso invertido (1-0-1) se reduce a medida que la frecuencia aumenta (aprox. $\tau/10$). Suponiendo que la frecuencia máxima de captura será de 50 Hz ($55,88 \text{ m/s} \approx 200 \text{ Km/h}$), el tiempo de pulso podrá ser de hasta 2 ms (milisegundos) como mínimo. Por este motivo, el biestable J-K tiene la entrada J conectada a 1, la entrada K conectada a 0 y la señal del pulso conectado a la entrada CLK (flanco ascendente), lo cual permitirá memorizar un 1 cada vez que el pulso pase de 0 a 1. De este modo, la NDS dispondrá de suficiente tiempo (τ mínimo = 20 ms) para detectar el 1 en el pin 5, y volver a poner el biestable a 0 con la señal de reset invertido, es decir, poniendo el pin 4 a 0. Además, la señal de reset se debe mantener a 0 durante 10 ms, y luego se debe volver a poner a 1 para permitir capturar el siguiente pulso.

Como la interfaz de Entrada/Salida utilizada no genera interrupciones, se deberá utilizar la RSI del *timer 0*, para la cual se propone utilizar una frecuencia de activación de 100 Hz.

El programa a implementar deberá ir realizando una serie de tareas independientes mientras efectúa la gestión del biestable J-K para medir la velocidad actual (instantánea) del viento. El valor de la velocidad se debe ir escribiendo por la pantalla inferior de la NDS, cada 3 segundos (aprox.), expresado en m/s con 2 decimales; cada medición debe ir precedida por un número correlativo, que empezará en 1 (ver pantalla del esquema).

Se dispone de las siguientes rutinas, ya implementadas:

Rutina	Descripción
swi_9	Rutina de la BIOS para división entera; parámetros (R0: dividendo, R1: divisor); resultado (R0: cociente, R1: resto, R3: cociente absoluto)
inicializaciones()	Inicializa el <i>hardware</i> (pantalla, etc.)
tareas_independientes()	Tareas que no dependen del anemómetro (ej. captación de otros parámetros meteorológicos, como temperatura,

	humedad relativa, presión atmosférica, etc.); tiempo de ejecución < 1 s
swiWaitForVBlank()	Espera hasta el próximo retroceso vertical
printf(char *format, ...)	Escribe un mensaje en la pantalla inferior de la NDS

Para programar la frecuencia de activación de la RSI del *timer* 0, se pide realizar la siguiente rutina específica:

```
void inicializar_timer0_01(unsigned int freq);
```

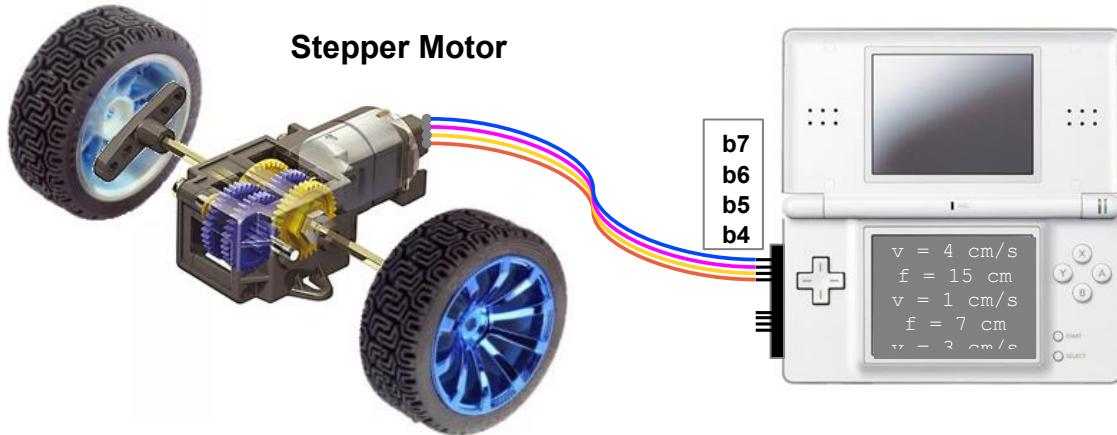
la cual recibe por parámetro la frecuencia de salida del *timer* 0. Esta rutina debe calcular el divisor de frecuencia asociado a la frecuencia de salida requerida, utilizando la segunda frecuencia de entrada más alta disponible (523.656,96875 Hz). Hay que recordar que las direcciones de memoria de los registros de datos y de control del *timer* 0 se definen simbólicamente como TIMER0_DATA y TIMER0_CR, respectivamente, y que en el registro de control hay que activar los bits 7 y 6 para iniciar el *timer* y generar interrupciones, mientras que en los bits 1 y 0 hay que indicar el código de la frecuencia de entrada requerida, que en nuestro caso será 01 (de aquí el sufijo en el nombre de la rutina).

Se pide:

Programa principal y variables globales en C, RSI del *timer* 0 y rutina `inicializar_timer0_01()` en ensamblador.

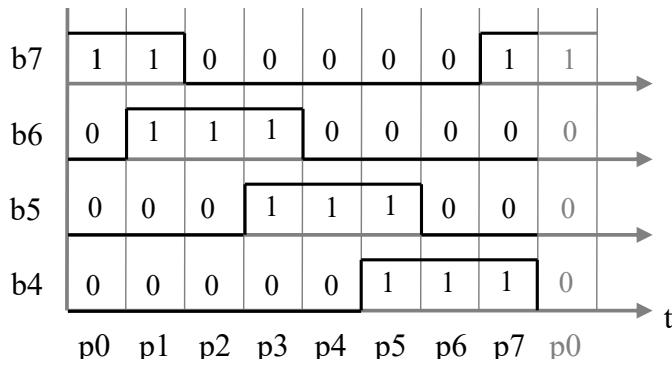
Problema 20 (1^a Conv. 2016-17): Motor de tracción

Se propone controlar un robot móvil tipo coche con la NDS, utilizando un motor para accionar las dos ruedas traseras de tracción y otro motor para fijar la orientación de las dos ruedas delanteras de dirección. En este problema, sin embargo, solo se pide el código del control del motor de las ruedas de tracción:

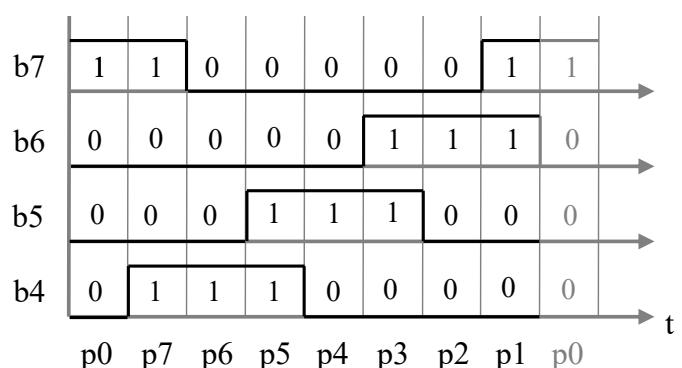


En el esquema anterior se ha representado la conexión de los 4 cables de control del motor paso a paso (*Stepper motor*) de tracción, que están asociados a los bits b7-b4 de un registro de E/S de 32 bits, al cual se accederá a través de la dirección de memoria 0x0A000000. El resto de bits del mismo registro se utilizarán para otras tareas de control del robot móvil, como la orientación de las ruedas de dirección. Por lo tanto, será necesario no modificar el estado del resto de bits cuando se actualicen los bits b7-b4. Para ello debemos suponer que se podrá leer, directamente del registro de E/S, el estado de los bits de salida fijado en su última escritura.

El motor paso a paso se controla mediante un determinado patrón en sus 4 bits de control. Existen 8 patrones posibles, que denominaremos **fases**, de la p0 a la p7. Al incrementar la fase, el motor avanza un paso, es decir, gira un cierto ángulo en un sentido (horario, por ejemplo). Al decrementar la fase, el motor retrocede un paso, es decir, gira el mismo ángulo en sentido contrario. Los siguientes esquemas muestran los patrones de bits para las 8 fases, además de la evolución de fases correspondiente a los dos sentidos de movimiento del robot móvil, hacia adelante (*Forward stepping*) y hacia atrás (*Backward stepping*):



Forward stepping



Backward stepping

La evolución de las fases es circular, es decir, la fase superior a la p_7 es la p_0 y la fase inferior a la p_0 es la p_7 . La frecuencia a la que se cambie de fase determinará la velocidad de rotación del motor. Según el fabricante, el motor utilizado admite hasta 1.000 cambios de fase (pasos) por segundo. Además, se nos indica que el motor requiere 64 pasos para dar una vuelta completa ($5,625^\circ/\text{paso}$). Sin embargo, el eje del motor está conectado a una serie de engranajes que reducen la rotación de las ruedas respecto a la del motor, de manera que se requieren 4.076 pasos para conseguir una rotación completa de las ruedas. Como las ruedas son de 5,34 cm de diámetro (16,78 cm de perímetro), serán necesarios 243 pasos para avanzar o retroceder 1 cm ($\pm 0,5\%$ error).

Se utilizará la RSI del *timer 0* para generar los cambios de fase, a la frecuencia necesaria para conseguir una velocidad del vehículo determinada. Como la frecuencia de cambios de fase no puede superar 1 KHz, la velocidad del vehículo, en valor entero, podrá ser de 4 cm/s (972 pasos/s), como máximo. Aunque es una velocidad relativamente lenta, respecto a un coche teledirigido, por ejemplo, hay que tener en cuenta que la ventaja de utilizar un motor paso a paso es que podremos determinar la posición del vehículo con mucha precisión (0,005 cm de error por cm avanzado), lo cual permitirá realizar aplicaciones de conducción automática y exploración del entorno.

El programa a implementar deberá ir ejecutando una serie de tareas (semi-)independientes al control del motor de tracción, como recibir órdenes, calcular la trayectoria, detectar obstáculos, etc. Concurrentemente, el código a implementar deberá obtener y ejecutar las consignas de movimiento del vehículo, generando los cambios de fase correspondientes. Cada consigna se obtendrá con una llamada a la rutina `siguiente_movimiento()`, que devolverá (por referencia) dos valores:

`vel` : valor entero entre -4 y 4, donde el valor absoluto indica la velocidad del robot en centímetros por segundo y el signo indica si hay que avanzar (+) o retroceder (-); si vale cero significa que no hay nueva consigna de movimiento, o sea, que el robot debe estar parado,

`avn`: valor natural entre 1 y 100, que indica los centímetros que tiene que avanzar o retroceder el robot, según el signo de la velocidad.

Es importante no llamar a la rutina `siguiente_movimiento()` hasta que no se haya terminado el movimiento anterior.

Además, en la pantalla inferior de la NDS se debe ir escribiendo cada nueva consigna recibida, en una línea la velocidad absoluta, por ejemplo, “`v = 3 cm/s`” y en la siguiente línea el avance, indicando ‘`f`’ si es hacia delante o ‘`b`’ si es hacia atrás, por ejemplo, “`b = 10 cm`” (ver pantalla inferior del gráfico inicial).

Se dispone de las siguientes rutinas, ya implementadas:

<i>Rutina</i>	<i>Descripción</i>
<code>inicializaciones()</code>	Inicializa <i>hardware</i> (pantalla, interrupciones, <i>timers</i> , etc.)
<code>tareas_independientes()</code>	Tareas que no dependen directamente del control del motor de tracción (ej. control de otro motor para fijar el ángulo de las ruedas de dirección); tiempo de ejecución entre 0,1 µs y 1 segundo
<code>activar_timer0()</code>	Activa el funcionamiento del <i>timer 0</i>
<code>desactivar_timer0()</code>	Desactiva el funcionamiento del <i>timer 0</i>
<code>fijar_frecuencia(int freq)</code>	Calcula y fija el divisor de frecuencia del <i>timer 0</i> para que genere interrupciones a la frecuencia de salida especificada por parámetro (en Hz)
<code>siguiente_movimiento(char *vel, unsigned char *avn)</code>	Devuelve por referencia los valores de una nueva consigna (ver descripción anterior), o velocidad igual a cero si no hay nueva consigna de movimiento; puede tardar entre 0,1 µs y 1 ms en ejecutarse
<code>swiWaitForVBlank()</code>	Espera hasta el próximo retroceso vertical
<code>printf(char *format, ...)</code>	Escribe un mensaje en la pantalla inferior de la NDS

Además del programa principal y la RSI del *timer* 0, se pide el código de la rutina principal de gestión de interrupciones (RPSI):

```
void intr_main();
```

la cual debe detectar si se ha activado el bit IRQ_TIMER0 en el registro REG_IF del controlador de interrupciones y, en caso afirmativo, debe invocar la RSI del *timer* 0. Además, debe notificar la resolución de cualquier IRQ que se haya producido sobre el propio registro REG_IF, así como sobre la posición de memoria INTR_WAIT_FLAGS, para conseguir desbloquear posibles llamadas a funciones de la BIOS que esperan la generación de interrupciones, como la función swiWaitForVBlank().

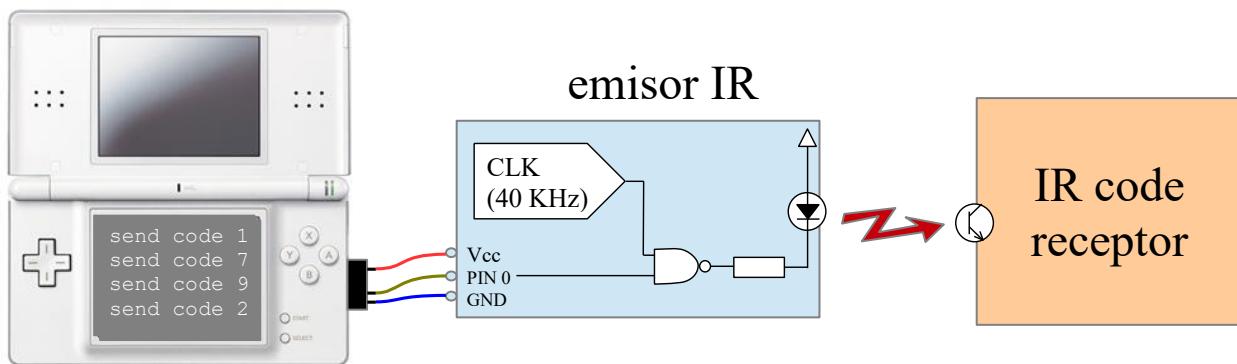
La rutina `inicializaciones()` instalará la dirección de la rutina `intr_main()` en la posición de memoria 0xB003FFC, destinada a almacenar la dirección de la RPSI de la NDS (bajo compilación con *devkitPro + libnds*).

Se pide:

Programa principal y variables globales en C, RSI del *timer* 0 y rutina `intr_main()` en ensamblador.

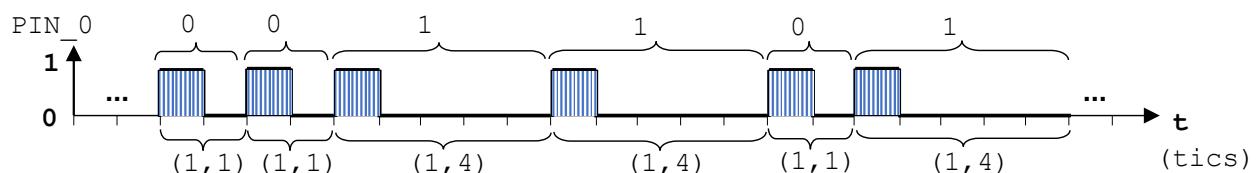
Problema 21 (2^a Conv. 2016-17): Emisor IR

Se propone controlar un circuito emisor de luz infrarroja (IR) con la NDS. El programa a realizar deberá permitir al usuario pulsar algunos botones de la NDS para emitir una determinada serie de ráfagas de luz infrarroja, que transmitirá un comando específico (encender/apagar, subir volumen, etc.) a un dispositivo receptor (televisión, reproductor de DVDs, etc.) compatible con el formato de códigos definido por la empresa NEC®:



En el esquema anterior se ha representado la conexión de la NDS con un circuito emisor IR, básicamente con un cable de datos denominado `PIN0` que controlará la transmisión de pulsos de luz a una frecuencia de 40 KHz. El estado del pin se podrá fijar escribiendo un 0 o un 1 en el bit 0 de un registro de E/S de nombre simbólico `REG_IR`; aunque el registro es de 32 bits, el resto de bits no tiene ninguna función asignada. Debido a la puerta NAND, durante el tiempo en que el `PIN0` esté a 1, el LED IR convertirá los pulsos eléctricos generados por el `CLK (CLocK)` en pulsos de luz infrarroja; esta señal periódica se denomina *portadora*. Si `PIN0` está a 0, no se enviará la señal portadora. A estos dos estados de enviar y no enviar portadora los denominaremos “ON” y “OFF”, respectivamente.

Para transmitir los códigos de los comandos, la señal portadora se activará y desactivará siguiendo unos determinados patrones de tiempo. Concretamente, se define la unidad 'tic' como el tiempo unitario de referencia. Los tics tendrán una frecuencia de 1.700 Hz, de modo que el tiempo de un tic será aproximadamente de 588 μ s.



Para codificar cada bit de información se especificará un par de tiempos expresados en tics, constituidos por un número de tics para el estado “ON” y otro número de tics para el estado

“OFF”. Un bit de datos 0 se codificará con el par (1, 1), mientras que un bit de datos 1 se codificará con el par (1, 4). Es decir, los dos tipos de bit de datos activan la portadora durante un único tic, pero los bits de datos a 0 la desactivan durante un solo tic, mientras que los bits de datos a 1 la desactivan durante cuatro tics. Esta diferencia en el tiempo de “OFF” es suficiente para que el receptor distinga el valor de cada bit de datos. El cronograma anterior muestra un ejemplo de transmisión de un trozo de secuencia binaria “...001101...”.

El programa a implementar deberá testar periódicamente los 10 botones de la NDS que se corresponden con los 10 bits de menor peso del registro REG_KEYINPUT, que son todos los botones menos **X** e **Y**. Para cada botón se deberá iniciar la transmisión de una determinada ráfaga de 32 bits de datos (32 pares ON/OFF), precedida de un par ON/OFF de inicio (*Lead In*) y seguida de otro par ON/OFF de final (*Led Out*). Según el formato NEC, el par de tiempos de *Lead In* es (15, 7), mientras que el par de tiempos de *Lead Out* es (1, 59). En resumen, se podrán transmitir hasta 10 comandos diferentes, cada uno de los cuales estará compuesto de 34 pares ON/OFF (1+32+1).

Se dispone de las siguientes rutinas, ya implementadas:

Rutina	Descripción
inicializaciones ()	Inicializa el <i>hardware</i> (pantalla, interrupciones, etc.)
activar_timer0 (int freqOut)	Activa la generación de interrupciones del <i>timer 0</i> , a la frecuencia de salida especificada por parámetro
desactivar_timer0 ()	Desactiva la generación de interrupciones del <i>timer 0</i>
scanKeys ()	Captura el estado actual de los botones de la NDS
int keysDown ()	Devuelve un patrón de bits con los botones activos (estado invertido de bits REG_KEYINPUT, 1 → botón pulsado, 0 → botón no pulsado)
swiWaitForVBlank ()	Espera hasta el próximo retroceso vertical
printf (char *format, ...)	Escribe un mensaje en la pantalla inferior de la NDS

Para generar las secuencias de pares ON/OFF de cada comando se proponen las siguientes variables globales:

```
unsigned short vCodes[10][34] =
{{0x0F07, 0x0101, 0x0101, 0x0101, 0x0104, 0x0101, ..., 0x013B},
{0x0F07, 0x0101, 0x0101, 0x0104, 0x0101, 0x0104, ..., 0x013B},
...
{0x0F07, 0x0101, 0x0104, 0x0101, 0x0101, 0x0101, ..., 0x013B}};
```

```
unsigned char current_code;           // índice código actual
unsigned char current_pair;         // índice par ON/OFF actual
unsigned char state;                // estado actual (1=ON, 0=OFF)
unsigned char tics;                 // tics pendientes
```

La matriz `vCodes [10] [34]` almacenará las 10 secuencias de 34 pares ON/OFF, donde cada par se codifica como un *halfword*, con los 8 bits altos para el tiempo del estado ON y los 8 bits bajos para el tiempo del estado OFF. Por ejemplo, un par (15, 7) se codifica como `0x0F07`, un par (1, 1) se codifica como `0x0101`, etc.

La variable `current_code` permitirá memorizar el índice del código actual (de 0 a 9). La variable `current_pair` permitirá memorizar el índice del par ON/OFF actual (de 0 a 33). La variable `state` permitirá memorizar el estado actual (0 o 1) del par actual. La variable `tics` permitirá memorizar cuantos tics faltan para que termine el estado actual.

En general, el programa principal debe consultar periódicamente los bits de los botones de la NDS. Cuando detecta uno pulsado, debe activar el *timer* 0 a una frecuencia de 1.700 Hz, inicializar las variables globales de control, activar el bit 0 del `REG_IR` y esperar a que termine la transmisión de toda la secuencia de 34 pares asociada al código del botón; no hay que realizar ninguna tarea independiente. Además, por la pantalla inferior de la NDS se debe escribir el mensaje “`send code x`”, donde ‘x’ debe indicar el índice del código a transmitir (de 0 a 9). Si se pulsan varios botones a la vez, solo se deberá transmitir el código de un único botón.

Por su parte, la RSI del *timer* 0 debe decrementar el número de tics pendientes del estado actual; cuando este número llegue a cero se debe pasar al siguiente estado y, si es necesario, pasar al siguiente par, actualizando el número de tics pendientes y el bit 0 del registro IR según el nuevo estado actual del par actual del código actual. Cuando se haya transmitido el último par del código actual, se debe detener el *timer* 0.

Según el formato NEC, los bits de información de todo comando siempre contienen 16 unos y 16 ceros, con lo cual se puede calcular el total de tics de cualquier secuencia, incluyendo los pares de *Lead In* y *Lead Out*. Este total es de 194 tics, que corresponden a un tiempo aproximado de 0,1141 segundos.

Para obtener el número de tics del estado actual del par actual del código actual, se pide realizar una rutina específica:

```
unsigned char obtener_tics(unsigned short codes[][][34],  
                           unsigned char ccode,  
                           unsigned char cpair,  
                           unsigned char cstate);
```

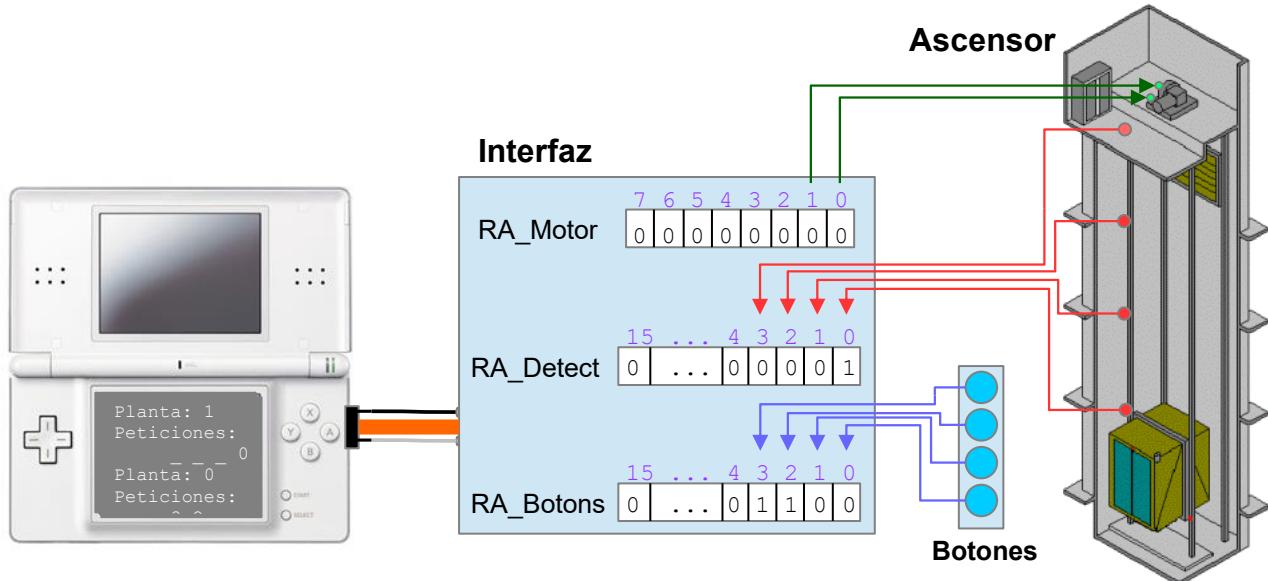
la cual recibe por parámetro la referencia de la matriz de códigos y los valores del código actual, par actual y estado actual, respectivamente. La rutina devuelve como resultado el número de tics correspondiente.

Se pide:

Programa principal en C, RSI del *timer 0* y rutina `obtener_tics()` en ensamblador.

Problema 22 (1^a Conv. 2017-18): Ascensor con memoria

Se propone conectar una interfaz electrónica para controlar un ascensor con la NDS, suponiendo que no habrá plantas de sótano:



La interfaz permite acceder a tres registros de Entrada/Salida, que tienen las siguientes funcionalidades:

Registro	Bits	Función
RA_Motor	8	Bit 0: activar (=1) / desactivar (=0) motor en subida Bit 1: activar (=1) / desactivar (=0) motor en bajada
RA_Detect	16	Bits $n-1..0$: el bit i -ésimo se activa (=1) automáticamente cuando el ascensor está situado justo delante de la puerta de la planta i ; el resto de bits estarán a 0; cuando el ascensor esté entre puertas, todos los bits estarán a 0
RA_Botons	16	Bits $n-1..0$: el bit i -ésimo se activa (=1) automáticamente cuando algún usuario pulsa sobre el botón de la planta i ; la petición queda memorizada en el propio registro hasta que, por programa, se escriba un 0 sobre ese bit; en todo momento pueden haber cero, una o más peticiones activadas (pendientes)

En el esquema de ejemplo, el número de plantas n es 4, de modo que el valor de i oscilará entre 0 (planta baja) y 3 (piso más alto). Sin embargo, hay que realizar el programa de control configurable para estructuras de ascensor entre 2 y 16 plantas ($2 \leq n \leq 16$).

La interfaz generará una interrupción IRQ_CART cada vez que se active un bit del registro RA_Detect, es decir, cuando el ascensor llegue (pase por delante) de la puerta de una determinada planta.

El programa a implementar deberá comprobar periódicamente si hay peticiones pendientes; en caso afirmativo y si el ascensor está parado, habrá que cerrar la puerta y activar el motor para mover el ascensor hacia las plantas solicitadas; si el ascensor está en marcha y llega a una nueva planta, habrá que escribir por pantalla información sobre el estado del ascensor (ver más adelante) y, si había una petición pendiente sobre esa planta, parar el ascensor y abrir la puerta.

Todas estas tareas se deberán realizar concurrentemente con determinadas tareas independientes. Se podrán realizar múltiples peticiones, con el ascensor en marcha o parado, y se deberán ir atendiendo todas las peticiones en un sentido (de subida o de bajada) y, cuando se hayan terminado las peticiones en ese sentido, pasar a atenderlas en el otro sentido. Al iniciar el programa supondremos que el ascensor está situado en la planta baja.

Se dispone de las siguientes rutinas, ya implementadas:

<i>Rutina</i>	<i>Descripción</i>
inicializaciones ()	Inicializa el <i>hardware</i> (pantalla, interrupciones, etc.)
tareas_independientes ()	Tareas que no dependen del movimiento del ascensor (ej. calcular el peso de los ocupantes del ascensor); tiempo de ejecución < 0,1 s
abrir_puerta()	Abre la puerta del ascensor y espera un cierto tiempo antes de retornar (10 segundos, aproximadamente)
cerrar_puerta()	Cierra la puerta del ascensor comprobando que no haya obstáculos, en cuyo caso la volverá a abrir y después repetirá el proceso de cerrarla (5 segundos, mínimo)
swiWaitForVBlank()	Espera hasta el próximo retroceso vertical
printf(char *format, ...)	Escribe un mensaje en la pantalla inferior de la NDS

Además de gestionar el control general del ascensor, el programa principal deberá escribir por la pantalla inferior de la NDS el número de planta actual y las peticiones pendientes, cada vez que el ascensor llegue o pase por delante de la puerta de una planta. El formato del texto deberá ser el siguiente (sin los comentarios entre paréntesis):

```
Planta: i          (0 ≤ i ≤ n-1)
Peticiones:      (si planta i está solicitada, pi = "i ")
(tabulador) pn-1 pn-2 ... p2 p1 p0      (sino, pi = "_")
```

Por ejemplo, para una configuración con 8 plantas ($n = 8$), si el ascensor llega a la planta 5 ($i = 5$) y están solicitadas las plantas 6, 2 y 1, la salida será la siguiente:

```
Planta: 5
Peticiones:
  _ 6 _ _ _ 2 1 _
```

Se sugiere el uso de las siguientes variables globales y definiciones:

```
#define NUM_PLANT 4           // número de plantas a gestionar

unsigned char planta_actual = 0; // número de planta actual
unsigned char sentido = 1;     // sentido de movimiento actual
                               //   = 1 → subida,
                               //   = 2 → bajada
```

El `#define` permitirá configurar el número de plantas a gestionar para una determinada estructura del ascensor (n). La variable `planta_actual` deberá registrar en todo momento el número de planta actual (i), cuyo valor siempre estará entre 0 y `NUM_PLANT`-1. La variable `sentido` deberá registrar el sentido actual del movimiento, ya sea de subida ($=1$) o de bajada ($=2$).

Por su parte, la RSI del ascensor deberá actualizar la variable global `planta_actual` para reflejar la planta donde se encuentra el ascensor en todo momento. Además, si la planta a la que acaba de llegar el ascensor estaba solicitada, deberá parar el motor inmediatamente, así como desactivar el bit correspondiente de petición de planta.

Por último, será necesario programar una rutina de soporte con la siguiente cabecera:

```
unsigned char siguiente_movimiento(unsigned short peticiones,
                                   unsigned char p_actual,
                                   unsigned char *s_actual);
```

Esta rutina calcula cual es el siguiente sentido del movimiento del motor, según el estado actual de los bits de petición (`peticiones`), la planta actual (`p_actual`) y el sentido actual del movimiento (`s_actual`); el tercer parámetro se pasa por referencia para poder ser modificado directamente en memoria desde la propia rutina.

Para determinar el nuevo sentido, la rutina tiene que recorrer los bits de petición de planta a partir del bit correspondiente a la planta actual, comprobando si existe alguna petición pendiente en el sentido de movimiento actual; si la encuentra, no habrá cambio del sentido actual; en caso contrario, se cambiará el sentido del movimiento.

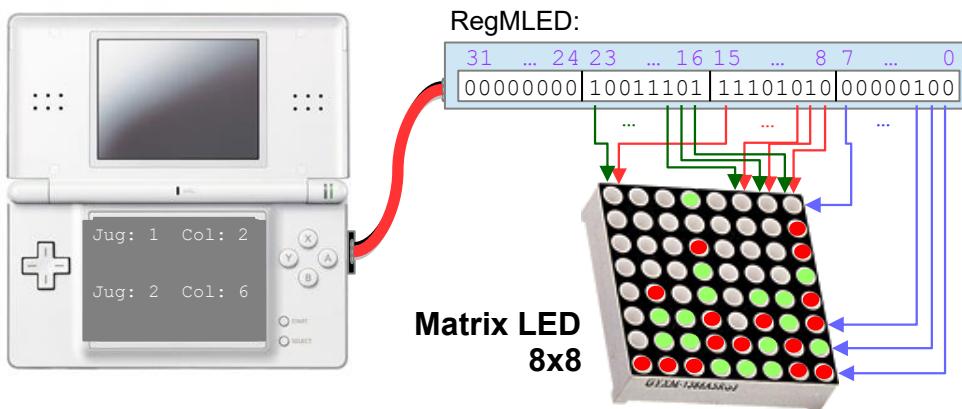
Además de cambiar directamente el contenido de la variable del sentido actual, su valor final se devuelve también como resultado de la rutina. Para simplificar el funcionamiento de esta rutina, se puede suponer que solo se llamará si existe alguna petición pendiente, aunque habrá que asegurarse de esta condición antes de llamar a la rutina. Además, vamos a suponer que la interfaz no registrará peticiones sobre la planta en la que esté situado el ascensor cuando esté parado.

Se pide:

Programa principal y variables globales en C,
RSI del ascensor y rutina `siguiente_movimiento()` en ensamblador.

Problema 23 (1^a Conv. 2017-18): Matriz 8x8 LEDs

Se propone conectar una interfaz para controlar una matriz de 8x8 LEDs bicolor (1588ABEG-5) con la NDS, con el fin de implementar el juego del 4 en ralla:

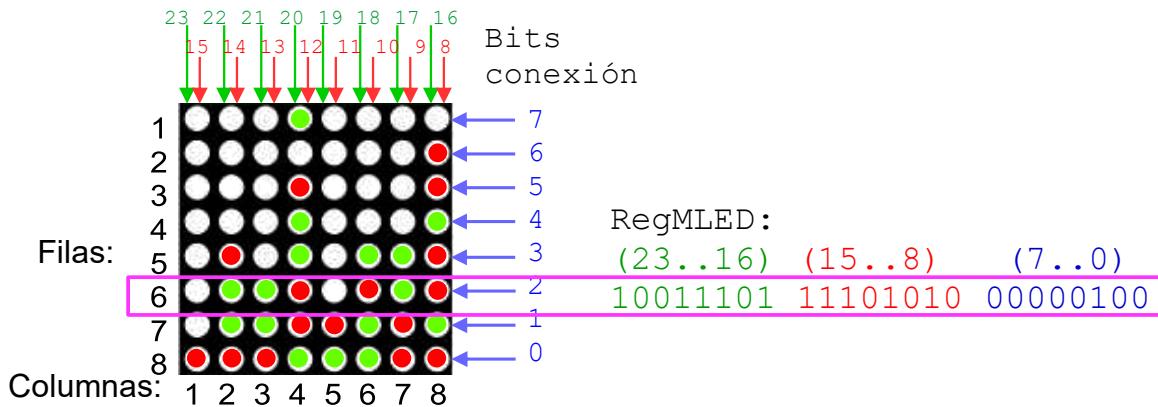


La interfaz dispone de un único registro de Entrada/Salida de 32 bits, de nombre simbólico RegMLED, pero solo se utilizarán los 24 bits de menor peso, los cuales están divididos en los siguientes campos:

Campo	Bits	Función
Columnas en Verde	23..16	Activan (=0) o desactivan (=1) el color verde de los 8 LEDs de la fila seleccionada por el campo de Filas
Columnas en Rojo	15..8	Activan (=0) o desactivan (=1) el color rojo de los 8 LEDs de la fila seleccionada por el campo de Filas
Filas	7..0	Activan (=1) la fila que se tiene que iluminar en cada momento

El problema de gestionar 64 LEDs, cada uno con 2 colores, es que resulta muy costoso conectar los 128 cables necesarios para el control individual del color de cada LED. Por este motivo, el dispositivo dispone de un cable por cada fila más dos cables por cada columna, de manera que se pueda aplicar la técnica del “barrido” por filas, que consiste en enviar corriente eléctrica sobre un único cable de fila y utilizar el voltaje de los 16 cables de las columnas para indicar el estado del color de los 8 LEDs de la fila seleccionada.

El siguiente esquema muestra un ejemplo de cómo se controla el color de los LEDs de la fila 6 (bit 2 = 1):



Si se recorren todas las filas secuencialmente, fijando el color de las 8 columnas de cada fila durante un breve periodo de tiempo (> 25 ms), repitiendo el proceso a una frecuencia suficientemente alta (≥ 30 Hz), el ojo humano no detecta el hecho de que solo hay una fila que está emitiendo luz en todo momento, sino que tiene la sensación de que la matriz está totalmente iluminada (fenómeno de persistencia visual).

La interfaz no genera interrupciones, de modo que se pide utilizar la RSI del *timer 0*, programada a una frecuencia de 240 Hz, para realizar el barrido periódico de la matriz.

Por otro lado, para implementar la dinámica del juego del 4 en ralla, se pide utilizar la RSI del *timer 1*, programada a una frecuencia de 3 Hz, que se encargará de generar el efecto de caída de las fichas (ver más adelante).

El programa a implementar controlará las fichas de dos jugadores. El jugador 1 llevará las rojas y el jugador 2 las verdes. El tablero consistirá en 7 filas (de la 2 a la 8) por 8 columnas. Por turnos, cada jugador tendrá que seleccionar la columna sobre la que quiere soltar su ficha. Para esto, sobre la fila superior (la 1) se mostrará una ficha del color del jugador que tiene el turno, posicionada inicialmente en la columna 4. El jugador podrá cambiar la posición (columna) de la ficha con las teclas KEY_RIGHT y KEY_LEFT. El jugador podrá soltar la ficha pulsando KEY_SELECT, si la columna actual no está llena (si no hay ficha en la fila inferior). Cuando se realice la selección de la columna, aparte de escribir dicha selección por la pantalla inferior de la NDS, la ficha empezará a caer hacia las filas inferiores hasta que llegue a la fila 8 o hasta que encuentre otra ficha en la posición inferior. En el momento que la ficha se detiene, el programa debe comprobar si hay cuatro en ralla; en caso negativo la partida continúa, en caso positivo se declara el vencedor por la pantalla de la NDS y la partida finaliza; el programa ya no hace nada más, hasta que se reinicie la NDS (no hay tareas

independientes).

Se dispone de las siguientes rutinas, ya implementadas:

<i>Rutina</i>	<i>Descripción</i>
inicializaciones()	Inicializa el <i>hardware</i> (pantalla, interrupciones, etc.)
scanKeys()	Captura el estado actual de los botones de la NDS
int keysDown()	Devuelve un patrón de bits con los botones activos
int comprobar_4(unsigned char *tablero, int nfil, int ncol)	Comprueba si existe 4 en ralla en un tablero de nfil x ncol posiciones, devolviendo 1 en caso afirmativo y 0 en caso negativo (tiempo de ejecución > 0,0001 s)
mostrar_ganador(int jugador)	Muestra por pantalla el mensaje de resultado de la partida
swiWaitForVBlank()	Espera hasta el próximo retroceso vertical
printf(char *format, ...)	Escribe un mensaje en la pantalla inferior de la NDS

En la pantalla inferior de la NDS de la figura inicial se muestran diversos ejemplos de la salida de texto que indican la selección de cada columna, que siguen el siguiente formato:

Jug: i (tabulador) Col: j (1 ≤ i ≤ 2) (1 ≤ j ≤ 8)

Se sugiere el uso de las siguientes variables globales:

```
unsigned char matriz[8][8]; // valores de los LEDs, en cada
                           // posición:
                           // = 0 → vacía (apagado)
                           // = 1 → ficha jugador 1 (rojo)
                           // = 2 → ficha jugador 2 (verde)
unsigned char turno = 1;      // ident. del jugador actual
unsigned char fil = 1, col = 4; // fila y columna de la ficha
                           // que se está colocando
```

Para la implementación del programa principal y la RSI del *timer* 1, puede ser conveniente usar una variable que indique en qué fase se encuentra el juego en cada instante (seleccionar columna, caída ficha, comprobar ganador, final partida).

A cada activación de la RSI del *timer* 1, si el programa se encuentra en la fase de caída, la ficha solo bajará una posición (fila inferior), en el caso de que sea posible, obviamente. Si la ficha se encuentra en la fila 8 o si se detecta otra ficha en la fila inferior a la actual, se habrá terminado la fase de caída.

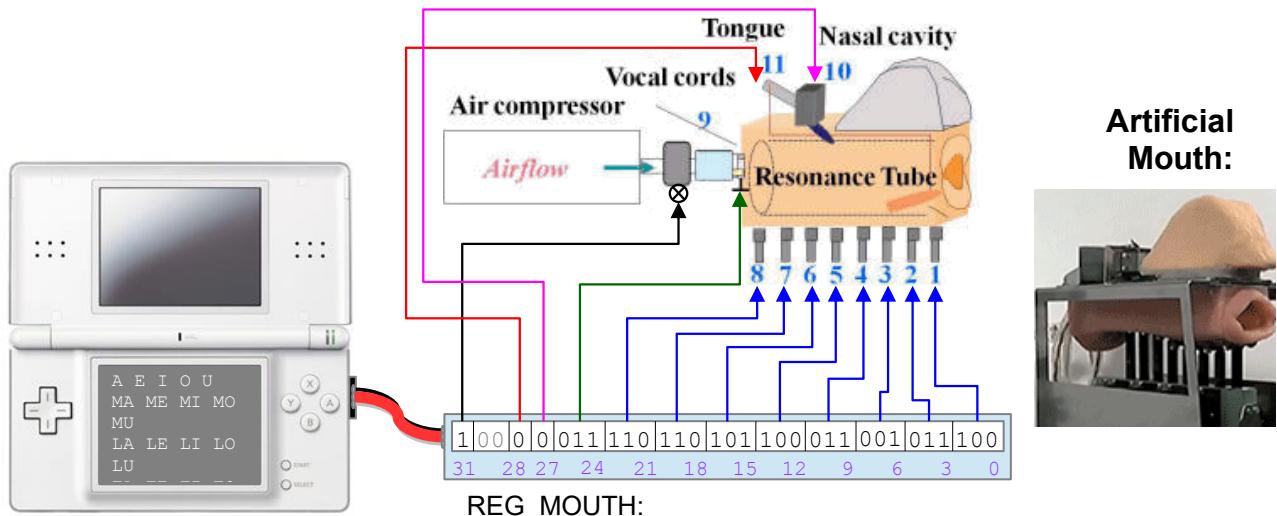
A cada activación de la RSI del *timer* 0 habrá que acceder a una de las filas de la matriz y generar los 24 bits de información para el refresco de esa fila. Obviamente, este proceso se debe realizar periódicamente para todas las filas de la matriz.

Se pide:

Programa principal y variables globales en C, RSIs de los *timers* en ensamblador.

Problema 24 (1^a Conv. 2017-18): Boca artificial

Hay que controlar una boca artificial que permite simular la vocalización humana (hasta cierto punto). Este dispositivo consiste básicamente en un tubo de resonancia por el que circula un flujo de aire que proviene de un depósito de aire comprimido. El tubo es flexible y su forma se puede modificar con una serie de 8 pistones que pueden generar 8 elevaciones diferentes cada uno. También se pueden regular 8 niveles de apertura de una válvula que simula las cuerdas vocales, una lengua artificial que se puede subir o bajar, y una cavidad nasal a la cual se puede desviar la salida del aire para reproducir sonidos nasales. Por último, hay una electroválvula que permite abrir o cerrar el flujo general del aire. A continuación se muestra un esquema de la conexión entre la NDS y el dispositivo, además de una foto real de dicho dispositivo:



El registro de Entrada/Salida REG_MOUTH permite controlar todos los actuadores del dispositivo mediante los siguientes campos:

Campo	Bits	Descripción
Tube shape	23..0	En grupos de 3 bits, indican el nivel de elevación de los 8 pistones que dan forma al tubo de resonancia, donde los 3 bits de menor peso controlan el pistón 1, los siguientes 3 bits controlan el pistón 2, etc.; el nivel 000 corresponde a la elevación más baja y el nivel 111 a la más alta
Vocal cords	26..24	Indican el nivel de apertura de la válvula que simula las cuerdas vocales, donde el nivel 000 es el más cerrado y el nivel 111 es el más abierto

Nasal cavity	27	Indica si el aire tiene que entrar en la cavidad nasal artificial (1) o no (0)
Tongue	28	Indica si la lengua artificial tiene que estar arriba (1) o abajo (0)
Air flow	31	Indica si se abre (1) o cierra (0) el flujo general de aire

El propósito general del programa a realizar es recibir las palabras a vocalizar a través de la wifi de la NDS, transcribir dichas palabras a “fonemas” específicos para el dispositivo propuesto (posición de los actuadores), visualizar la palabra actual en la pantalla inferior de la NDS y enviar cada fonema al registro de E/S durante el tiempo asociado al fonema, todo ello mientras se realizan una serie de tareas independientes.

Se dispone de las siguientes rutinas, ya implementadas:

Rutina	Descripción
inicializaciones()	Inicializa el <i>hardware</i> (pantalla, interrupciones, etc.)
tareas_independientes()	Tareas que no dependen de la vocalización de las palabras; supondremos que nunca tardarán más de 100 milisegundos en ejecutarse
activar_timer0(int freq)	Activa el funcionamiento del <i>timer</i> 0 a la frecuencia indicada por parámetro (en Hz), incluida la activación de las interrupciones correspondientes (IRQ_TIMER0).
desactivar_timer0()	Desactiva el funcionamiento del <i>timer</i> 0
int pal2fon(char *pal, unsigned int fon[], unsigned char tim[])	Transcribe una palabra que se pasa por parámetro como <i>string</i> (<i>pal</i>) a una secuencia de valores con la posición de los actuadores del dispositivo (excepto bit 31) sobre un vector que se pasa por referencia (<i>fon</i>), junto con el tiempo (en centésimas) de cada fonema sobre otro vector pasado por referencia (<i>tim</i>), y devuelve como resultado el número total de fonemas generados; el tiempo mínimo de ejecución de esta rutina es de 10 milisegundos
swiWaitForVBlank()	Espera hasta el próximo retroceso vertical
printf(char *format, ...)	Escribe un mensaje en la pantalla inferior de la NDS

Se propone usar las siguientes variables globales y definiciones:

```
#define MAX_CAR      16
#define MAX_FON      50
char palabra[MAX_CAR+1];           // palabra a vocalizar
unsigned int fonemas[MAX_FON];    // fonemas a vocalizar
unsigned char fcs[MAX_FON];        // tiempo de cada fonema, en
                                   // centésimas de segundo
unsigned char num_fon = 0;          // número de fonemas a vocalizar
```

El vector `palabra[]` permitirá almacenar los caracteres de la palabra actual a vocalizar, suponiendo que nunca recibiremos palabras de más de 16 caracteres. Los vectores `fonemas[]` y `fcs[]` permitirán almacenar el valor de los actuadores y el tiempo en centésimas de los fonemas correspondientes a la palabra actual. Por último, la variable `num_fon` permitirá almacenar el número total de fonemas obtenidos, suponiendo que nunca habrá más de 50 fonemas para cualquier palabra a vocalizar.

Para gestionar el envío de fonemas al registro de E/S, se pide usar la RSI del *timer 0* para detectar si se ha agotado el tiempo del fonema actual y, en caso afirmativo, que active el siguiente fonema, si es que hay fonemas pendientes. En caso de haber terminado con la vocalización del último fonema, habrá que cerrar la electroválvula del paso general del aire y desactivar el *timer 0*.

El inicio de la vocalización del primer fonema se tiene que gestionar desde el programa principal, además de activar el *timer 0* cuando se reciba una nueva palabra a vocalizar.

Para detectar la recepción de una nueva palabra, se pide realizar una rutina específica:

```
int siguiente_palabra(char *string);
```

la cual recibe por parámetro la dirección inicial de un vector sobre el cual almacenará los caracteres de la palabra, además de añadir un carácter centinela de final de *string* ('\0'). Como resultado hay que devolver la longitud del *string* (sin contar el centinela), o cero si no se ha recibido ninguna palabra en el momento de la llamada a la rutina.

Debido a que la comunicación con la wifi solo la puede gestionar el procesador ARM7 y el programa a realizar se ejecutará íntegramente sobre el ARM9, se pide utilizar el mecanismo IPC-FIFO de la NDS para traspasar el texto a vocalizar, palabra a palabra, como secuencias de caracteres sobre la cola FIFO de recepción del ARM9. Hay que recordar que esta cola dispone de 16 posiciones (*words*), de modo que el ARM7 podrá copiar una palabra entera sobre la cola, enviando hasta 16 caracteres seguidos cuando reciba una notificación de que la cola está vacía (IRQ_FIFO_EMPTY).

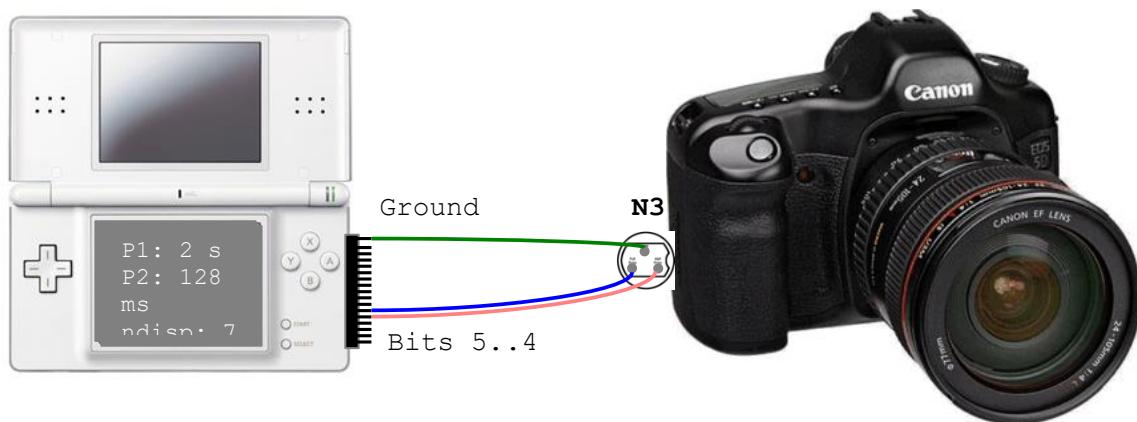
Por su parte, la rutina `siguiente_palabra()` podrá detectar si hay alguna palabra pendiente en la cola consultando el bit 8 del registro `REG_IPC_FIFO_CR`, que valdrá 1 si la cola de recepción está vacía o 0 si no lo está. Para obtener todos los caracteres almacenados en la cola, habrá que ir leyendo el registro `REG_IPC_FIFO_RX` (32 bits) secuencialmente, hasta que la cola esté vacía.

Se pide:

Programa principal en C, RSI del *timer 0* y rutina `siguiente_palabra()` en ensamblador.

Problema 25 (2^a Conv. 2017-18): Disparador de cámara réflex

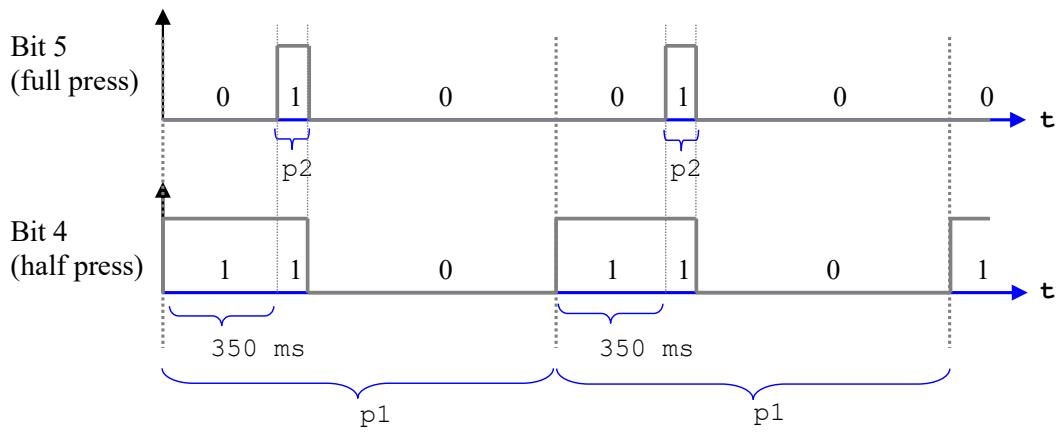
Hay que controlar el botón de disparo de una cámara fotográfica réflex a través de un conector N3 de Canon®. Este conector dispone de 3 pines, uno de masa eléctrica (*Ground*), otro que activa la función de media presión del botón de disparo (*half press*) y otro que activa la función de presión completa (*full press*). Se utilizará una NDS con una interfaz de Entrada/Salida de 16 bits, que conectará los bits 4 y 5 de un registro de nombre simbólico REG_SHUTTER a los pines de *half press* y *full press* (respectivamente) de una cámara con conector N3 o compatible. El resto de bits del registro de E/S no se utilizan. A continuación se muestra un esquema de la conexión entre la NDS y un dispositivo de ejemplo:



La función de *half press* indica a la cámara que enfoque y que calcule los ajustes de exposición según la luz entrante (apertura del diafragma, tiempo de exposición, sensibilidad del sensor de imagen, etc.). Se supone que cualquier cámara profesional tardará menos de 350 milisegundos en realizar dichas tareas. La función de *full press* da la orden de realizar la foto con los ajustes obtenidos. Además, se puede controlar el tiempo de exposición mediante el tiempo durante el cual el pin de *full press* esté a 1.

El programa a realizar enviará un 1 por el bit 4 durante 350 milisegundos. Después enviará un 1 por los bits 4 y 5 durante un determinado periodo de exposición especificado por el usuario (**p2**). Además, el usuario también podrá indicar al programa que realice un cierto número de fotos consecutivas, con otro determinado intervalo de tiempo entre foto y foto (**p1**).

El siguiente esquema muestra un ejemplo de la forma de onda de los bits 4 y 5 para dos fotos consecutivas:



Se dispone de las siguientes rutinas, ya implementadas:

Rutina	Descripción
inicializaciones ()	Inicializa el <i>hardware</i> (pantalla, interrupciones, etc.)
tareas_independientes ()	Tareas que no dependen de la gestión de los disparos (ej. medición de la cantidad de luz recibida por el sensor); t. ejecución ≈ 50 ms
activar_timers01(int period)	Activa los <i>timers</i> 0 y 1 en modo encadenado, para generar la activación de la IRQ del <i>timer</i> 1 cada cierto tiempo, especificado por parámetro en segundos
activar_timer2(int period)	Activa el <i>timer</i> 2 para generar la activación de su IRQ cada cierto tiempo, especificado por parámetro en milisegundos
desactivar_timers(int patrn)	Desactiva el funcionamiento de los <i>timers</i> indicados por parámetro mediante un patrón de cuatro bits (del bit 3 al bit 0), donde cada bit provocará la desactivación (1) o no (0) del <i>timer</i> correspondiente
scanKeys ()	Captura el estado actual de los botones de la NDS
int keysDown ()	Devuelve un patrón de bits con los botones pulsados
swiWaitForVBlank()	Espera hasta el próximo retroceso vertical
printf(char *format, ...)	Escribe un mensaje en la pantalla inferior NDS

Se propone usar las siguientes variables globales:

```
unsigned char num_disp = 0;           // número de disparos pendientes
unsigned char ind_p1 = 0;             // índice del periodo p1
unsigned char ind_p2 = 0;             // índice del periodo p2
unsigned char status = 0;             // estado actual (0 → parado,
                                     // 1 → half press,
                                     // 2 → half+full press)
unsigned char actualizar_info = 0;    // indica si hay que actualizar
                                     // información en pantalla
```

La variable `num_disp` indicará el número de disparos pendientes. El usuario podrá incrementar dicho número pulsando el botón `KEY_START`.

Las variables `ind_p1` e `ind_p2` almacenarán un índice entre 0 y 9, que permitirá obtener rápidamente el valor de los períodos **p1** y **p2** como 2 elevado al índice correspondiente. Se sugiere esta codificación porque proporciona un amplio rango de niveles con pocos valores. Concretamente, para el periodo de exposición **p2** obtendremos diez valores entre 1 y 512 milisegundos, lo cual se aproxima a los tiempos de exposición utilizados habitualmente por las cámaras fotográficas (1/1000, 1/500, 1/250, 1/125, 1/60, 1/30, 1/15, 1/8, 1/4, 1/2 segundos). El usuario podrá cambiar el valor del índice `ind_p2` con los botones `KEY_UP` y `KEY_DOWN`. Por otro lado, el periodo de los disparos **p1** abarcará un rango entre 1 y 512 segundos. El usuario podrá cambiar el valor del índice `ind_p1` con los botones `KEY_LEFT` y `KEY_RIGHT`.

La variable `status` permitirá controlar en qué estado del ciclo de un disparo se encuentra el sistema, pasando por los valores 1 (*half press*), 2 (*half+full press*), 0 (receso).

La variable `actualizar_info` sirve para indicar cuándo ha habido un cambio en alguno de los valores de configuración principales, esto es, en el número de disparos pendientes o en alguno de los dos períodos, ya sea por pulsación de botones o porque se ha generado un nuevo disparo automáticamente (mediante las RSIs). Cada vez que haya un cambio de información, por la pantalla inferior de la NDS habrá que visualizar los siguientes mensajes:

```
Periodo 1: p1 s
Periodo 2: p2 ms
Disparos pendientes: nd
```

donde *p1*, *p2* y *nd* se tienen que sustituir por su valor correspondiente.

Para gestionar la activación automática de disparos, se pide usar la RSI del *timer 1* para generar un nuevo disparo si quedan disparos pendientes, cada vez que pase el tiempo

correspondiente al periodo **p1**. Si no quedan más disparos, la propia RSI puede desactivar los *timers* 0 y 1 para detener el proceso automático.

Por otro lado, se pide usar la RSI del *timer* 2 para controlar el estado de los bits de cada disparo, pasando de *half press* a *half+full press* después de 350 milisegundos desde el inicio de la activación, y de *half+full press* a reposo después de los milisegundos correspondientes al periodo **p2**, desactivando el *timer* 2 en esta segunda transición.

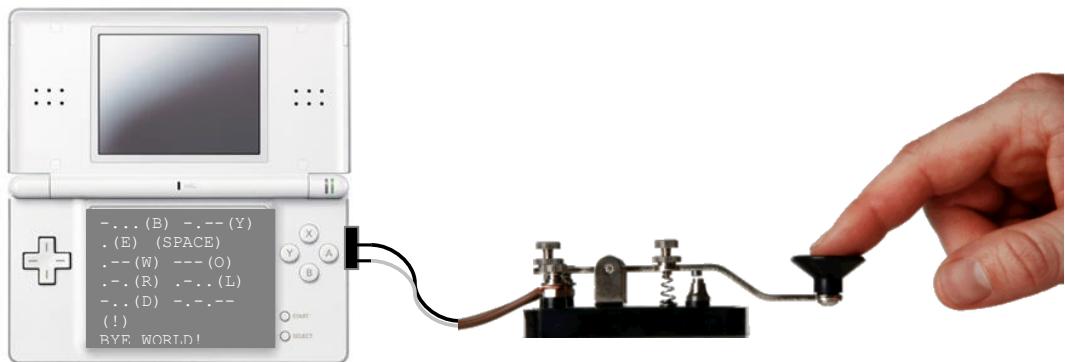
Por último, hay que tener en cuenta que, si el proceso de activación automática está detenido y el usuario pide un nuevo disparo, habrá que iniciar dicho proceso desde el programa principal. Además, el usuario podrá cambiar los valores de configuración (`num_disp`, `ind_p1`, `ind_p2`) en cualquier momento, incluso si el proceso de activación automática ya está en marcha.

Se pide:

Programa principal en C, RSIs del *timer* 1 y del *timer* 2 en ensamblador.

Problema 26 (1^a Conv. 2018-19): Lectura Morse

Se propone conectar un pulsador a la NDS para generar código Morse:



La interfaz que vamos a utilizar consiste en un único registro de Entrada/Salida de 32 bits, de nombre simbólico `REG_DATA`. El pulsador estará conectado al bit 15 del registro, aunque puede haber otros sensores o actuadores conectados al resto de bits.

El usuario utilizará el pulsador para codificar un mensaje en forma de puntos y rallas. Para distinguir los elementos del código Morse se utiliza la duración de las pulsaciones. Para detectar dicha duración, vamos a usar un tiempo de referencia que denominaremos *tic*, cuyo periodo se fijará al inicio del programa entre 50 y 150 milisegundos.

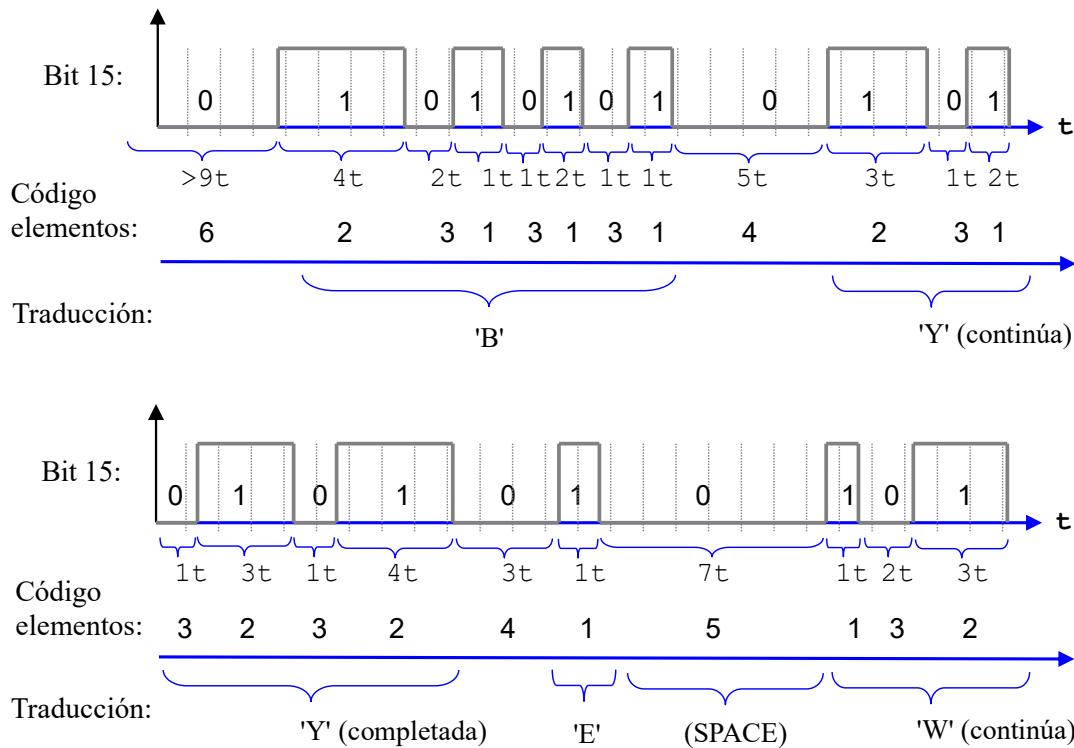
La siguiente tabla muestra el tiempo mínimo y máximo, en número de tics, que se deberá considerar para distinguir entre los símbolos '.' y '—', que son los elementos del código Morse generados con el pulsador a 1 (pulsado), pero también se presentan los tiempos mínimo y máximo de los espacios separadores entre símbolos, entre letras, entre palabras y entre mensajes, que son los elementos generados con el pulsador a 0 (soltado):

Código	Elemento	t_min	t_max	bit 15
1	'.' (símbolo punto)	1	2	1
2	'—' (símbolo ralla)	3	9	1
3	espacio entre símbolos	1	2	0
4	espacio entre letras	3	5	0
5	espacio entre palabras	6	9	0
6	espacio entre mensajes	10	∞	0

La variabilidad del tiempo de cada símbolo es debida a que el operador no puede generar dichos tiempos exactamente, de modo que será necesario establecer unos rangos que permitan cierta flexibilidad a la hora de introducir los pulsos. Para simplificar un poco el problema, se supondrá que el operador jamás mantendrá el pulsador a 1 más de 9 tics. Sin embargo, el

tiempo máximo del espacio entre mensajes no está limitado, aunque se podrá considerar que después de 10 tics ya se habrá terminado el último mensaje introducido.

El siguiente cronograma (partido en dos trozos consecutivos) muestra una introducción de pulsos para la frase “BYE WORLD!”, aunque solo llega hasta el inicio de la letra ‘W’; en este cronograma se muestran, en línea discontinua, los puntos de muestreo periódico de la señal de pulsación, así como el número de tics obtenido para cada estado del pulsador y sus correspondientes códigos de elemento y traducción para el mensaje:



El programa a implementar deberá realizar un seguimiento de las pulsaciones del usuario. Cuando se detecte el final de una letra, se deberán escribir por pantalla todos los puntos y rallas introducidos hasta el momento, además de su letra correspondiente, a continuación (sin espacios en blanco) y entre paréntesis. Cuando se detecte el final de una palabra, se deberá escribir “(SPACE)” (sin las comillas). Hay que dejar un espacio en blanco después de cada bloque de puntos y rallas/letra, así como al final del bloque “(SPACE)”. Cuando se detecte el final del mensaje, se deberá insertar un salto de línea y volver a escribir todo el mensaje como un *string* convencional, es decir, solo las letras y los espacios como un carácter de espacio normal (no hay que escribir “(SPACE)”). A continuación se muestra la visualización en pantalla correspondiente a la introducción de la frase de ejemplo:

```
- . . . (B) - . -- (Y) . (E) (SPACE) . -- (W) --- (O) . - . (R) . - . . (L) - . - . --
(!)
BYE WORLD!
```

Además, al finalizar la introducción de un mensaje, hay que enviarlo por la Wifi a otra NDS. Concurrentemente, el programa también deberá recibir mensajes por la Wifi de la otra NDS y escribirlos en la otra pantalla, como *strings* convencionales. Estas tareas de comunicación a través de la Wifi corresponden a las “tareas independientes” del proceso de captación, traducción y visualización de mensajes en código Morse descrito en el párrafo anterior, aunque el envío de un nuevo mensaje está supeditado al final de la captación de dicho mensaje, obviamente.

Para realizar este programa se dispone de las siguientes rutinas ya implementadas:

Rutina	Descripción
int inicializaciones()	Inicializa el <i>hardware</i> (pantalla, interrupciones, etc.); devuelve el valor del tiempo de referencia, en ms (no inicializa el <i>timer</i> 0)
inicializar_timer0(unsigned short freq)	Inicializa el <i>timer</i> 0 para que genere interrupciones a la frecuencia especificada por parámetro, en hercios.
recibir_mensaje()	Gestiona la recepción de nuevos mensajes por la Wifi, escribiéndolos en la pantalla superior de la NDS (tiempo de ejecución máximo = 150 ms)
enviar_mensaje(char *msg)	Gestiona el envío de un mensaje por la Wifi, que se pasa por parámetro como un <i>string</i> (formato C), pero sin escribirlo por pantalla (t. ejecución máx = 100 ms)
char traducir_morse(unsigned char *simbolos, int lon)	Traduce un vector de símbolos Morse (códigos 1 y 2) con el número de elementos indicado por el parámetro <i>lon</i> , y devuelve como resultado el código ASCII de la letra correspondiente (t. ejecución máx. = 30 µs)
swiWaitForVBlank()	Espera hasta el próximo retroceso vertical
printf(char *format, ...)	Escribe un mensaje en la pantalla inferior de la NDS

Se sugiere el uso de las siguientes variables globales y definiciones:

```
#define MAX_LETRAS 100          // máximo de letras por mensaje

unsigned char i_simb = 0;        // índice del vector simb[]
unsigned char simb[8];          // vector de símbolos Morse
                                // 1 → '.' (punto)
                                // 2 → '-' (ralla)
unsigned char i_mens = 0;        // índice del vector mensaje[]
char mensaje[MAX_LETRAS+1];    // vector de letras (string)
```

El `#define` permitirá configurar el número máximo de caracteres que se admiten por mensaje. La variable `i_simb` permitirá indexar el vector `simb`, cuyo propósito es registrar los símbolos Morse de una letra, con los códigos 1 y 2 para los puntos y las rallas. Todos los códigos Morse tienen como máximo 8 símbolos. La variable `i_mens` permitirá indexar el

vector `mensaje`, cuyo propósito es registrar los códigos ASCII de todas las letras del mensaje, en forma de *string* de lenguaje C (con centinela final).

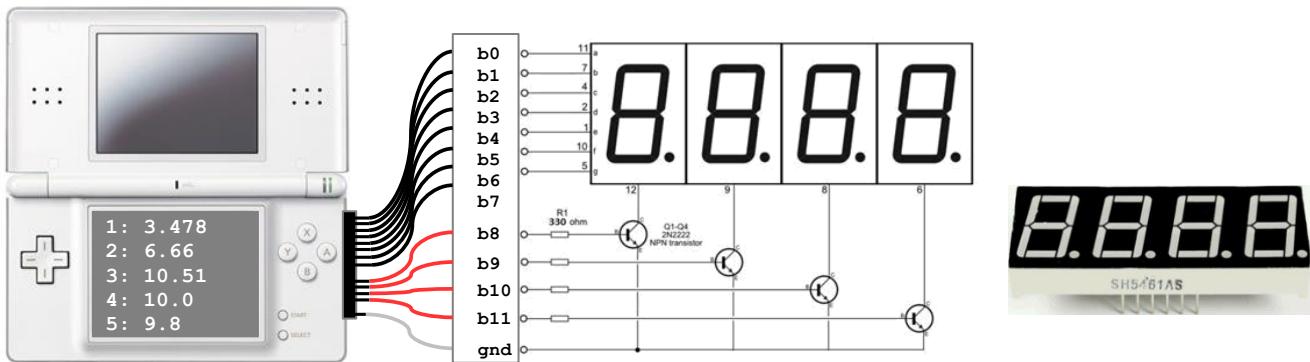
Para conseguir una sincronización con las pulsaciones suficientemente rápida, se deberá utilizar la interrupción del *timer 0* para muestrear periódicamente el valor del pulsador.

Se pide:

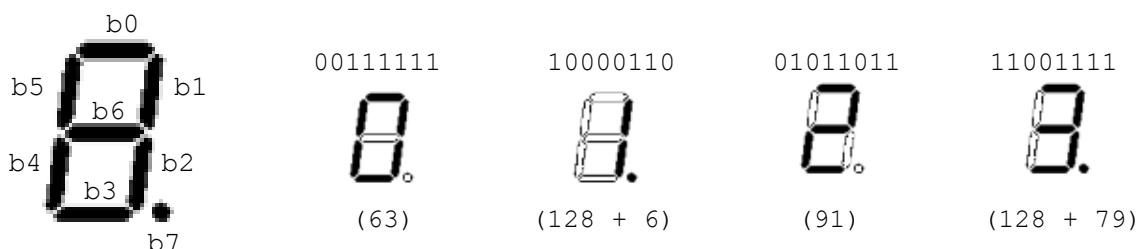
Programa principal y variables globales adicionales en C, RSI del timer0 en ensamblador.

Problema 27 (1^a Conv. 2018-19): Display numérico 4 dígitos

Se propone conectar a la NDS un dispositivo para visualizar números de hasta 4 dígitos, compuestos cada uno de ellos por 7 segmentos más un punto decimal. En el siguiente esquema se muestra la conexión lógica de los 12 bits necesarios para gestionar el display, junto con una foto de un dispositivo real (SH5461AS):



La interfaz que vamos a utilizar consiste en un único registro de Entrada/Salida de 16 bits, de nombre simbólico REG_DATA. Los 8 bits de menos peso sirven para determinar la activación (1) o no (0) de cada uno de los 7 segmentos más el punto decimal de cada dígito individual. El siguiente gráfico muestra la correspondencia entre los bits (b_0, b_1, b_2 , etc.) y los elementos luminosos de un dígito, así como la codificación numérica, en binario y en decimal, de los bits para generar los dígitos del 0 al 3, con el punto decimal activado en los casos 1 y 3:



En la codificación en base diez de los dígitos de ejemplo hemos separado expresamente el valor del punto decimal (128) del valor de los otros 7 bits, ya que la activación de dicho punto decimal es opcional. Para generar los 7 segmentos de cada uno de los diez dígitos decimales podemos utilizar el siguiente vector:

```
//vector de generación de los dígitos decimales en 7 segmentos
unsigned char Vsegments[] = {63, 6, 91, 79, 102, 109, 125, 7, 127, 111};
```

Para poder activar los 7 segmentos más el punto decimal de las 4 posiciones del display, será necesario realizar una multiplexación temporal usando los bits del b_8 al b_{11} del registro REG_DATA. Esto es, para controlar la activación/desactivación de los elementos luminosos del dígito de más peso (de más a la izquierda), los bits $b_{11}-b_8$ deberán fijarse a 0001 durante un instante, al mismo tiempo que los bits b_7-b_0 deberán enviar el código correspondiente a dicho dígito. En el instante siguiente, los bits $b_{11}-b_8$ se fijarán a 0010, mientras que los bits b_7-b_0 enviarán la codificación para el dígito de la segunda posición. El proceso se repetirá

para las posiciones tercera ($b_{11}-b_8=0100$) y cuarta ($b_{11}-b_8=1000$), para después volver a empezar por la primera posición. En cada instante, solo los elementos luminosos de una posición estarán emitiendo luz. Sin embargo, si el instante es suficientemente largo (≥ 25 ms) al mismo tiempo que la frecuencia de cambio de posiciones es lo suficientemente alta (≥ 25 Hz), el ojo humano no percibirá dicho proceso de multiplexación temporal, sino que experimentará la sensación de que todas las posiciones están emitiendo luz al mismo tiempo (persistencia visual).

El programa a implementar deberá capturar un número real (tipo `float`) con un dispositivo independiente al display, por ejemplo, un sensor de distancia, de temperatura, de presión, etc (tarea independiente). La captura se debe realizar durante todo el tiempo que el programa esté en funcionamiento, es decir, se deben ir obteniendo diversas muestras del valor real, aunque no se requiere que el proceso de captura siga una periodicidad estricta. Sin embargo, se pide que se capturen unos 10 valores por segundo y que se realice una media de todos los valores obtenidos durante cada segundo, con el fin de evitar una excesiva fluctuación del valor que se está visualizando.

A cada segundo, el resultado del promedio se convertirá en un vector de 4 valores decimales individuales, correspondientes a los dígitos de cada posición. Esta conversión la realizará una rutina ya implementada (ver tabla de rutinas), la cual también proporcionará el número de dígitos enteros y el número de dígitos decimales que contiene el vector. El siguiente esquema muestra cinco ejemplos de números promedio junto con su correspondiente transformación a vector de dígitos, números de dígitos entero y decimal, y visualización en el display:

Número	Vdgitos	Dent	Ddec	Visualización
3,478	3 4 7 8	1	3	8.8.8.8.
6,66	6 6 6 x	1	2	8.8.8.8.
10,51	1 0 5 1	2	2	8.8.8.8.
10,0	1 0 0 x	2	1	8.8.8.8.
9,8	9 8 x x 0 1 2 3	1	1	8.8.8.8.

Como se puede observar, es posible que para representar un número se requieran menos de cuatro dígitos ($\text{Dent} + \text{Ddec} < 4$). En este caso, la visualización temporal de dígitos solo deberá activar las posiciones del display necesarias, empezando siempre por la primera posición de la izquierda.

Para gestionar la visualización temporal, así como para controlar el paso de cada segundo para realizar el promedio de valores capturados, se deberá utilizar la RSI del *timer 0*, programada con un periodo de 25 milisegundos.

Para realizar este programa se dispone de las siguientes rutinas ya implementadas:

Rutina	Descripción
inicializaciones ()	Inicializa el <i>hardware</i> (pantalla, interrupciones, etc.); no inicializa el <i>timer 0</i>
inicializar_timer0 (unsigned short freq)	Inicializa el <i>timer 0</i> para que genere interrupciones a la frecuencia especificada por parámetro, en hercios.
float capturar_dato()	Captura y devuelve el valor del sensor; tiempo ejecución mínimo = 15 ms, tiempo ejecución máximo = 25 ms
convertir_numero(float valor, unsigned char *Vdigitos, unsigned char *Dent, unsigned char *Ddec)	Convierte un valor real en una serie de 4 dígitos decimales (redondeando el valor de entrada), que se guardan en el vector Vdigitos a partir de la primera posición del vector; devuelve por referencia el número de dígitos enteros (Dent) y el número de dígitos decimales (Ddec); tiempo ejecución máximo = 800 µs
swiWaitForVBlank ()	Espera hasta el próximo retroceso vertical
printf(char *format,...)	Escribe un mensaje en la pantalla inferior de la NDS

Se sugiere el uso de las siguientes variables globales, además del vector vsegment [] descrito anteriormente:

```
#define MAX_CAPTURAS    10          // máximo número de capturas
                                // (en un segundo)
unsigned char n_vals = 0;        // número de capturas actual
float valores[MAX_CAPTURAS];    // vector de valores capturados

unsigned char Vdigits[4];        // vector de dígitos
unsigned char num_dent;          // número de dígitos enteros
unsigned char num_ddec;          // número de dígitos decimales
```

La variable `n_vals` permitirá contar el número de elementos almacenados en todo momento dentro del vector `valores[]`, cuyo propósito es registrar todos los valores capturados por el sensor en el último segundo. Las variables `num_dent` y `num_ddec` permitirán registrar el formato decimal del número que se descompondrá en dígitos individuales sobre el vector `Vdigits[]`.

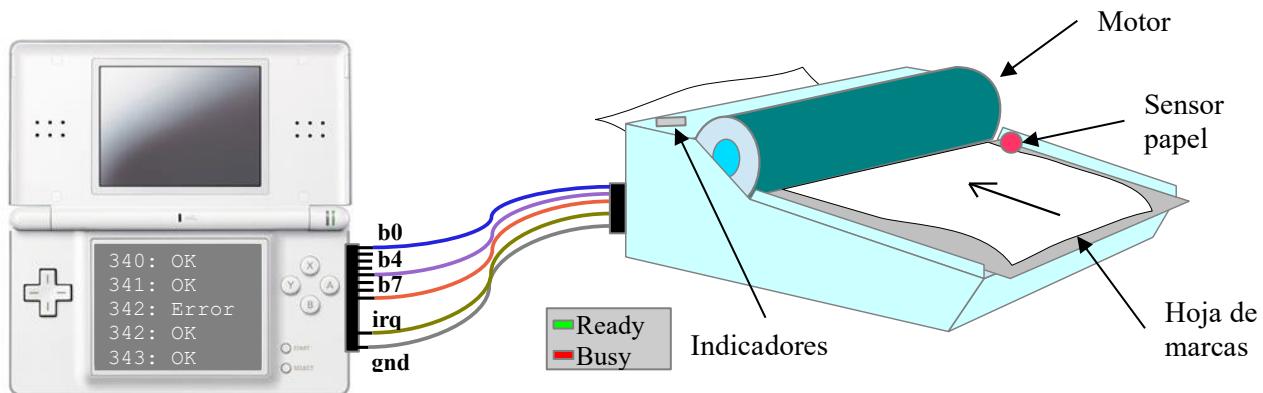
Cada segundo, el contenido de `Vdigits[]` se tienen que visualizar por la pantalla inferior de la NDS, junto con el número de segundo actual (valor secuencial). No se deberá escribir directamente el valor del número promediado porque éste puede contener más decimales de los que se obtienen con la rutina `convertir_numero()`. Tampoco se permite el uso de llamadas a funciones matemáticas como `round()`. Se puede suponer que siempre habrá al menos un dígito entero y un dígito decimal, aunque sean 0.

Se pide:

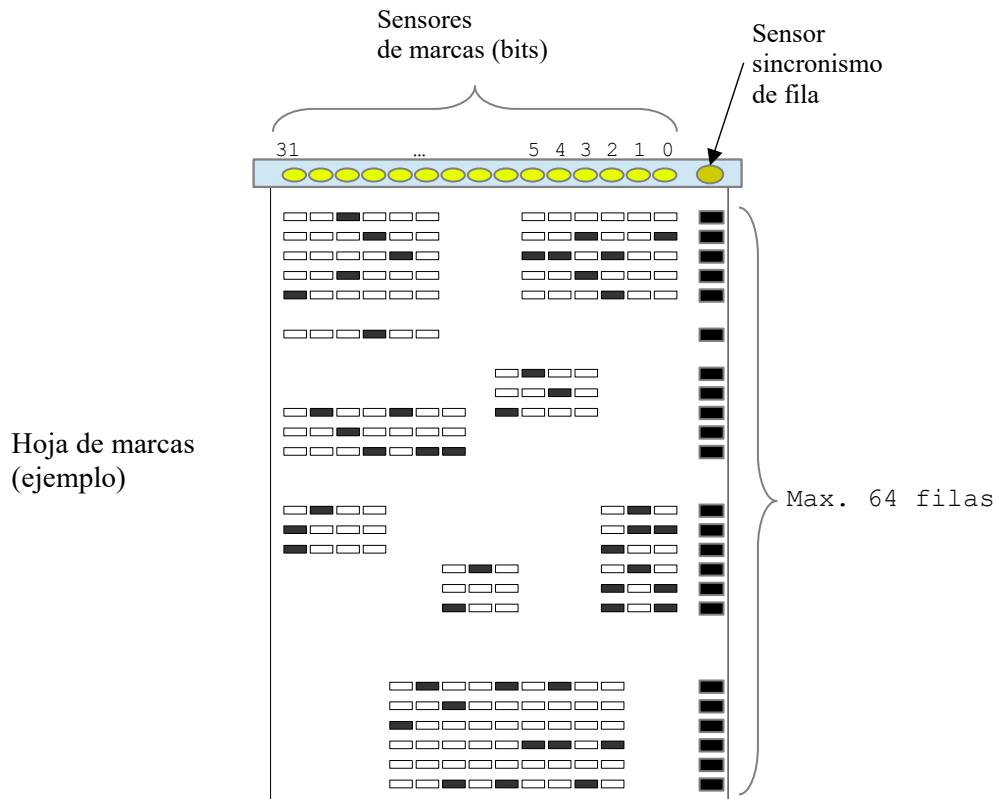
Programa principal y variables globales adicionales en C, RSI del timer0 en ensamblador.

Problema 28 (1^a Conv. 2018-19): Lector de hojas de marcas

Se propone controlar un dispositivo para leer hojas de marcas, es decir, formularios donde el usuario puede introducir información marcando determinados huecos para señalar opciones, valores, dígitos, etc. En el siguiente esquema se muestra una representación simbólica del dispositivo y de su conexión con el computador (NDS):



El siguiente gráfico muestra un ejemplo de hoja de marcas con una determinada disposición de elementos relevantes (huecos, marcas), además de los sensores ubicados dentro del dispositivo:



A la derecha de la hoja se encuentra la columna de marcas de sincronismo de fila, que vienen impresas en la hoja junto con los huecos a llenar y el texto descriptivo de cada información a introducir. Sin embargo, en el gráfico de ejemplo no se ha incluido ningún texto por motivos de espacio. La función de las marcas de sincronismo es la de indicar dónde hay filas de huecos. En nuestro caso, una hoja podrá incluir hasta 64 filas, como máximo (si todas son contiguas), aunque cada diseño de formulario puede definir su propia disposición de filas. La disposición de los huecos a llenar también dependerá del diseño concreto del formulario, pero siempre estarán alineados con los sensores de marcas. En nuestro caso disponemos de 32 sensores de marcas, por lo tanto, en el diseño del formulario se podrán utilizar hasta 32 columnas de huecos. Obviamente, el marcaje de los huecos depende de las respuestas de cada usuario, pero en el gráfico de ejemplo se han señalado (en negro) algunos huecos para proporcionar una idea de la apariencia de estas hojas de marcas.

La interfaz que vamos a utilizar consiste en un único registro de Entrada/Salida de 8 bits, de nombre simbólico `REG_MARCAS`, además de la interrupción `IRQ_MARCAS`, que se activa cuando el dispositivo envía un flanco ascendente por el cable `irq`. A continuación se describe la función de cada señal de la interfaz computador/dispositivo:

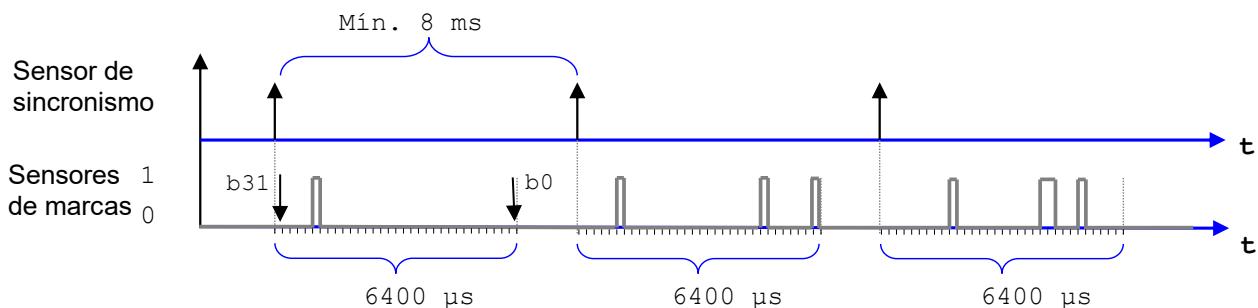
<i>Señal</i>	<i>Descripción</i>
Reg.E/S: bit 0	Indica el estado del sensor de papel: 0 → no hay hoja, 1 → hay hoja
Reg.E/S: bit 4	Transmite (en serie, empezando por el sensor 31) el estado de los huecos de marcas: 0 → hueco no marcado, 1 → hueco marcado
Reg.E/S: bit 7	Controla el estado de los indicadores luminosos y del motor de tracción de hojas: 0 → READY encendido, BUSY apagado, motor parado 1 → READY apagado, BUSY encendido, motor en marcha
<code>irq</code>	Se activa a cada detección de sincronismo de fila
<code>gnd</code>	Nivel de voltaje de referencia del 0 (<i>ground</i>)

El funcionamiento del sistema es el siguiente:

- inicialmente el dispositivo está en reposo (motor parado) y el indicador READY está iluminado,
- cuando el usuario introduce una hoja de marcas en el dispositivo, el sensor de papel lo detecta y lo comunica al computador (NDS) a través del bit 0 del registro de E/S,
- cuando la NDS está preparada para empezar a procesar la lectura de la hoja actual, envía un 1 a través del bit 7 del registro de E/S,
- entonces el dispositivo activa el indicador BUSY y pone en marcha el motor de tracción de hojas,
- al mover la hoja por debajo de los sensores, el dispositivo detecta las marcas de sincronismo de fila y captura las posibles marcas del usuario,
- por cada nueva marca de sincronismo de fila, el dispositivo activa la señal de petición de interrupción (`irq`) y empieza a transmitir el estado de los 32 sensores de marcas a través del bit 4 del registro de E/S, en serie, es decir, uno detrás del otro, a una

- velocidad constante (32 bits en 6,4 ms), empezando por el sensor 31,
- la velocidad máxima de tracción del papel determina un tiempo mínimo entre sincronismos de fila de 8 milisegundos, suponiendo que las filas estén contiguas; en el caso de que la separación entre dos filas sea superior a la mínima, el tiempo entre sus respectivas señales de sincronismos será superior a 8 ms, lógicamente,
 - cuando se ha pasado toda la hoja, el sensor de papel se pone a 0,
 - cuando la NDS detecta el final de la lectura de la hoja, debe transmitir toda la información de dicha hoja a un ordenador remoto (a través de una conexión Wifi),
 - el ordenador remoto contestará con un mensaje de lectura correcta (marcas del usuario coherentes) o incorrecta; entonces la NDS mostrará este resultado por su pantalla inferior, junto con el número de hojas leídas correctamente (ver esquema inicial),
 - después de mostrar dicha información por pantalla, la NDS desactivará el bit 7 del registro de E/S, con lo cual el dispositivo parará el motor de tracción y activará el indicador READY, para volver a repetir el proceso desde el principio.

Con el fin de exemplificar la transmisión en serie de las detecciones de marcas, el siguiente cronograma muestra la evolución de la activación de la señal `irq` y del bit 4 del registro de E/S para tres filas consecutivas:



Para realizar el programa de la NDS se dispone de las siguientes rutinas ya implementadas:

Rutina	Descripción
<code>inicializaciones ()</code>	Inicializa el <i>hardware</i> (pantalla, interrupciones, etc.); no inicializa el <i>timer 0</i>
<code>tareas_independientes ()</code>	Tareas que no dependen de la lectura de las hojas de marcas (máx. 100 ms)
<code>initializar_timer0 (unsigned short freq)</code>	Inicializa el <i>timer 0</i> para que genere interrupciones a la frecuencia especificada por parámetro, en hercios
<code>desactivar_timer0 ()</code>	Desactiva el <i>timer 0</i>
<code>unsigned char enviar_hoja (unsigned int m[], unsigned int n_fil)</code>	Envía el resultado de la lectura de una hoja de marcas, que se pasa por referencia como un vector de <i>words</i> , junto con el número de filas (posiciones del vector), y devuelve un booleano que indica si el contenido de

	la hoja es coherente (cierto) o no (falso); tiempo mínimo de ejecución = 100 ms
swiWaitForVBlank()	Espera hasta el próximo retroceso vertical
printf(char *format, ...)	Escribe un mensaje en la pantalla inferior de la NDS

Se sugiere el uso de las siguientes variables globales:

```
#define MAX_FILAS      64           // máximo número de filas por hoja
unsigned char num_filas = 0;          // número de filas actual
unsigned int marcas[MAX_FILAS];       // vector de marcas capturadas
unsigned short num_hojas = 0;          // núm. hojas leídas correctamente
```

La variable `num_filas` permitirá contar el número de filas almacenadas en todo momento dentro del vector `marcas[]`, cuyo propósito es registrar todos los valores (bits) capturados por los sensores de marcas. La variable `num_hojas` permitirá contabilizar el número de hojas leídas correctamente hasta el momento.

Para recibir el valor de los sensores de marcas enviados en serie se debe utilizar la RSI del `timer0`, programada a la frecuencia adecuada.

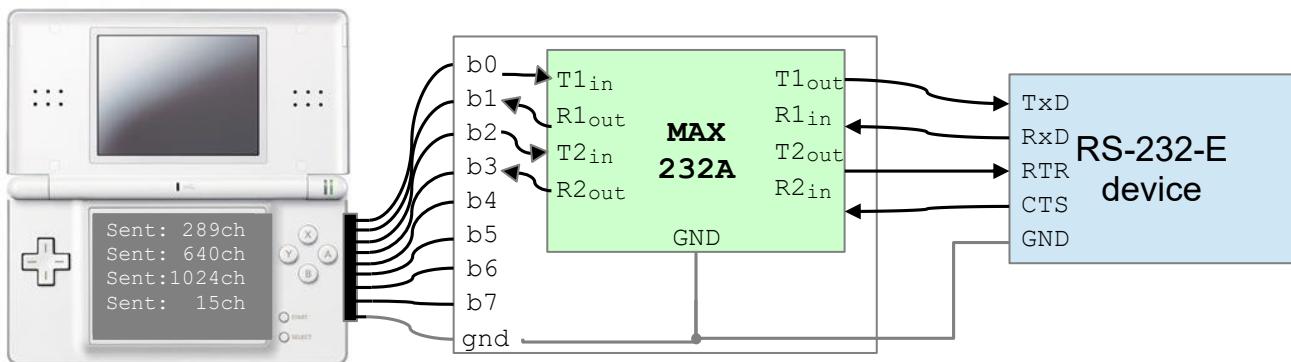
La gestión de la lectura de las hojas de marcas debe ser concurrente con la realización de las tareas independientes.

Se pide:

Programa principal y variables globales adicionales en C, RSIs del dispositivo y del `timer0` en ensamblador.

Problema 29 (2^a Conv. 2018-19): Envío de datos por RS-232-E

Se propone controlar un dispositivo que recibe datos serializados (bit a bit) a través de un conector compatible con el estándar RS-232-E, por ejemplo, una impresora serie, una máquina industrial, un foco de luz programable o un sintetizador de sonido. Los datos se enviarán desde un ordenador central hacia una NDS a través de una conexión wifi, y la NDS deberá transmitirlos al dispositivo a través de una interfaz como la del siguiente esquema:



Dicha interfaz consiste en un único registro de Entrada/Salida de 8 bits, de nombre simbólico REG_RS232, conectado a un circuito adaptador de señal eléctrica MAX232A, cuyo propósito es transformar señales lógicas TTL ($0 \rightarrow 0V..+0,8V$; $1 \rightarrow +2V..+5V$) a señales lógicas RS-232 ($0 \rightarrow +3V..+15V$; $1 \rightarrow -15V..-3V$). A continuación se describe la función de cada señal (bit del reg. E/S \leftrightarrow línea) de la interfaz NDS/dispositivo:

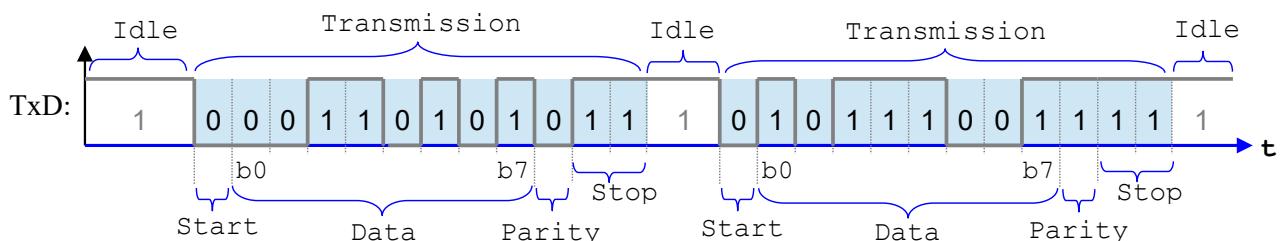
Interfaz NDS	Líneas RS-232	Descripción
Reg.E/S: bit 0	TxD (Transmitted Data)	Línea de transmisión de datos (bits), desde el computador (NDS) hacia el dispositivo
Reg.E/S: bit 1	RxD (Received Data)	Línea de recepción de datos (bits), desde el dispositivo hacia el computador (NDS)
Reg.E/S: bit 2	RTR (Ready To Receive)	El computador (NDS) activará esta señal (1) cuando esté preparado para recibir datos por la línea RxD
Reg.E/S: bit 3	CTS (Clear To Send)	El dispositivo activará esta señal (1) cuando esté preparado para recibir datos por la línea de transmisión de la NDS (TxD)
gnd	GND (GrouND)	Nivel de voltaje de referencia del 0

El funcionamiento del sistema para enviar datos al dispositivo será el siguiente:

- cuando se ponga en marcha la NDS, se establecerá la conexión wifi con el ordenador central y se determinará la velocidad de transferencia de datos (ver descripción de la rutina de inicializaciones),
- la NDS recibirá datos desde la wifi, en forma de paquetes de caracteres (vectores de bytes); como máximo, se recibirán 1024 caracteres por paquete; se dispone de una rutina ya implementada para gestionar esta comunicación,

- cuando se disponga de un paquete, cada uno de sus bytes se deberá transformar en un bloque de 12 bits, según un formato concreto para su transmisión asíncrona en serie (1 bit de *start*, 8 bits de datos, 1 bit de paridad y 2 bits de *stop*); este aspecto se detalla más adelante, aunque se dispone de una rutina ya implementada para realizar esta conversión,
- a continuación, si el dispositivo está preparado para la recepción de datos ($CTS = 1$), la NDS pasará a transferir los bits de todos los bloques del paquete actual a través de la línea TxD , bit a bit, a la velocidad (bits/s) establecida en la inicialización,
- al final de la transferencia de cada cada bloque de 12 bits, la NDS debe consultar el estado de la señal CTS ; en el caso de que esté a 0, la NDS debe detener temporalmente la transferencia del siguiente bloque,
- mientras el dispositivo no esté preparado ($CTS = 0$), la NDS tendrá que ir consultando periódicamente esta señal para detectar cuándo pasa a valer 1; en ese caso, la NDS deberá empezar la transmisión o reanudarla, si la había detenido previamente,
- cuando se hayan transferido todos los bloques del paquete actual, la NDS escribirá por su pantalla inferior el número de caracteres enviados al dispositivo (ver pantalla NDS en el esquema inicial) y pasará de nuevo a esperar la recepción del siguiente paquete que le pueda enviar el ordenador central,
- todo el proceso de recepción de paquetes, conversión a bloques y transmisión de los bits por la conexión RS-232 se debe realizar concurrentemente con ciertas tareas independientes; puede que dichas tareas accedan a los bits del registro `REG_RS232` no utilizados por este protocolo de envío de datos al dispositivo.

Con el fin de ejemplificar la transmisión en serie de los bloques, el siguiente cronograma muestra la evolución de la activación de la línea TxD para dos bloques consecutivos, correspondientes a los bytes 10101100 y 10011101:



Cada bloque empieza siempre con un bit de inicio (*Start*) que vale 0, seguido por los 8 bits del byte (*Data*), enviados de menor a mayor peso, seguidos por un bit de paridad (*Parity*), cuyo valor debe asegurar que el número de bits a 1 de todo el paquete sea siempre par, y termina con dos bits de final de bloque (*Stop*), que siempre valen 11.

El periodo de transmisión de cada bloque está indicado en la parte superior del cronograma como *Transmission*, mientras que los períodos de reposo están indicados como *Idle*, donde la línea TxD debe permanecer a 1. Aunque el cronograma de ejemplo muestra cierto tiempo de reposo entre los dos bloques, en realidad este *Idle* no es estrictamente necesario, ya que los bits de *Start* y *Stop* son suficientes para que el receptor identifique el final de un bloque y el inicio del siguiente.

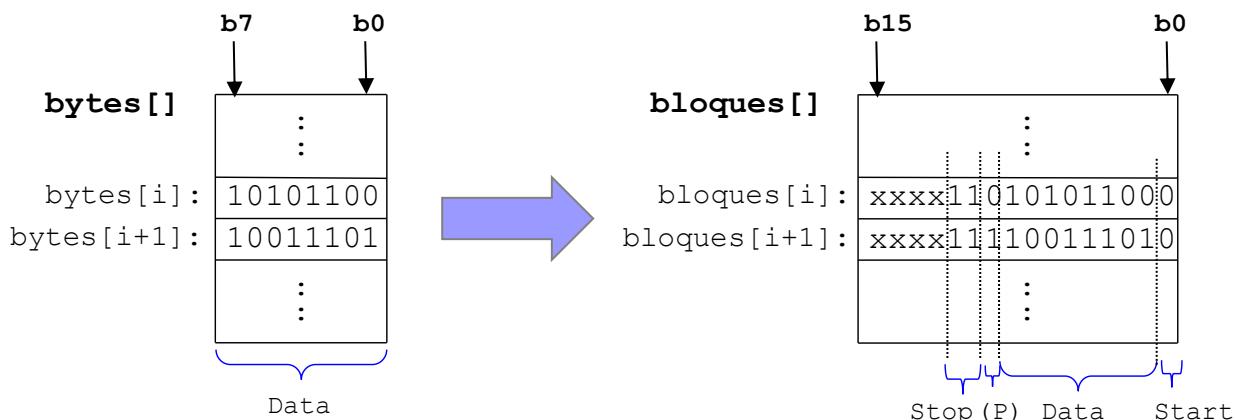
Para realizar el programa de la NDS se dispone de las siguientes rutinas ya implementadas:

<i>Rutina</i>	<i>Descripción</i>
<code>short inicializaciones()</code>	Inicializa el <i>hardware</i> (pantalla, interrupciones, etc.); no inicializa el <i>timer 0</i> ; establece la conexión wifi con el ordenador central y devuelve el número de bits por segundo al que debe realizarse la transmisión serie; los valores típicos serán 75, 110, 300, 1200, 2400, 4800, 9600 y 19200 bps.
<code>tareas_independientes()</code>	Tareas que no dependen de la transmisión de datos al dispositivo en cuestión (t. máx. = 100 ms)
<code>inicializar_timer0(unsigned short freq)</code>	Inicializa el <i>timer 0</i> para que genere interrupciones a la frecuencia especificada por parámetro, en hercios
<code>desactivar_timer0()</code>	Desactiva el <i>timer 0</i>
<code>unsigned short recibir_datos(unsigned char by[])</code>	Si el ordenador central está preparado para enviar un paquete de caracteres, esta rutina recibirá y copiará dichos caracteres dentro del vector que se pasa por referencia, devolviendo como resultado el total de caracteres copiados (máx. 1024); si no hay ningún paquete preparado, la rutina retornará directamente y devolverá cero como resultado; tiempo máximo de ejecución = 50 ms
<code>generar_bloques(unsigned short bl[], unsigned char by[], unsigned short nb)</code>	Rutina que convierte los bytes contenidos en el vector <code>by[]</code> en bloques de 12 bits, dentro del vector <code>bl[]</code> ; el parámetro <code>nb</code> indica cuantos bytes se deben convertir (t. máx. = 0,1 ms)
<code>swiWaitForVBlank()</code>	Espera hasta el próximo retroceso vertical
<code>printf(char *format, ...)</code>	Escribe un mensaje en la pantalla inferior de la NDS

Se sugiere el uso de las siguientes variables globales:

La variable `num_datos` permitirá registrar el número de datos que contiene el paquete actual de caracteres a transmitir. Dicho paquete se podrá almacenar dentro del vector `bytes[]`. Además, se define otro vector `bloques[]` para poder albergar los bloques de 12 bits correspondientes a cada carácter de 8 bits, según la conversión que realiza la rutina `generar_bloques()`.

Aunque dicha rutina ya está implementada, hay que saber cómo empaqueta los bits de cada bloque dentro del vector correspondiente. El siguiente gráfico muestra la codificación de dos caracteres consecutivos almacenados en el vector de bytes:



Como se puede observar, los 8 bits de datos quedan desplazados un bit a la izquierda, conservando su ordenación inicial, es decir, el bit de menos peso del dato (b_0) se almacena en el bit 1 del bloque y el bit de más peso del dato (b_7) se almacena en el bit 8 del bloque. El bit de menos peso del bloque se reserva para el *Start*, el bit de paridad (P) se almacena en b_9 , mientras que los bits de *Stop* se almacenan en b_{10} y b_{11} . Los cuatro bits de más peso en el vector de bloques quedan indeterminados (xxxx).

Para gestionar la transmisión en serie de los bits de los bloques del paquete actual se debe utilizar la RSI del *timer0*, programada a la frecuencia adecuada.

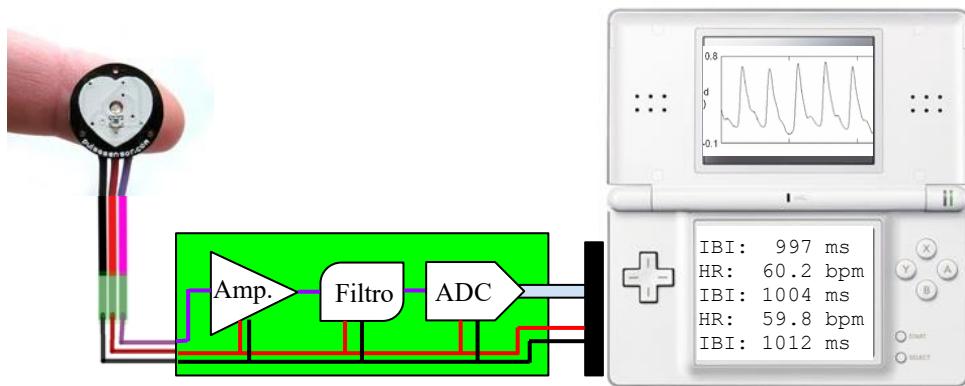
Se pide:

Programa principal y variables globales adicionales en C, RSI del *timer0* en ensamblador.

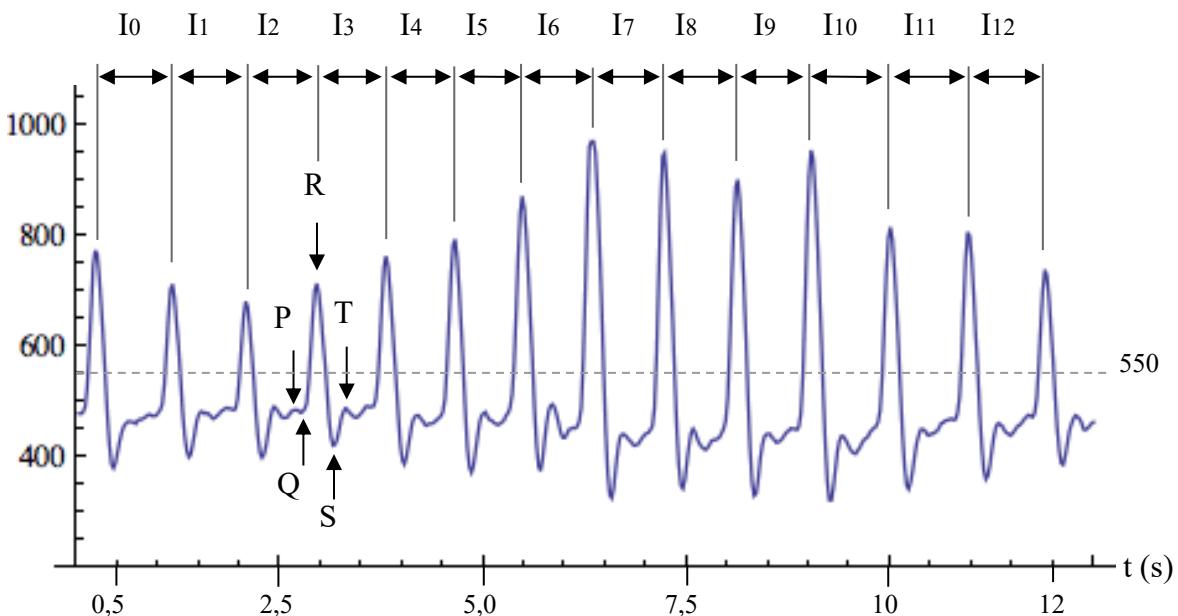
Problema 30 (1^a Conv. 2019-20): Frecuencia cardíaca

Se propone conectar a la NDS un dispositivo *PulseSensor* (pulsesensor.com) para detectar los latidos del corazón mediante fotopletismografía, es decir, estimando el volumen de las arterias a partir de la cantidad de luz absorbida por la sangre. Para realizar dicha estimación, el sensor dispone de un LED y un fototransistor que se deben enfocar hacia la piel de un dedo o del lóbulo de una oreja.

El esquema siguiente muestra la conexión del hardware, con un circuito electrónico compuesto por un amplificador, un filtro y un conversor analógico-digital (ADC) que convierte la señal eléctrica generada por el sensor en un número de 10 bits:



El siguiente esquema muestra un ejemplo de variación de señal, junto con unas etiquetas P, Q, R, S y T que indican los eventos significativos (sístole, diástole, etc.) de un latido concreto, aunque también se aplican al resto de latidos, obviamente. Además, se han representado los intervalos ($I_0, I_1, I_2, I_3, \dots$) entre los picos R, es decir, los tiempos entre sístoles:



(gráfico basado en contenidos de <https://community.wolfram.com/groups/-/m/t/272663>)

Como se puede observar, el nivel de la señal en los picos R es muy variable (entre 600 y 1000). Además, la señal proporciona otros picos de máximo local (P y T) y mínimo local (Q y S). Para simplificar el problema, vamos a suponer que la señal está suficientemente filtrada para que no se produzcan otros máximos o mínimos locales debidos a ruido de señal (variaciones de captación), aunque dicho proceso de filtrado puede eliminar los picos P y Q en algunos latidos. También vamos a suponer que todos los picos R obtendrán valores de señal por encima del umbral 550, mientras que los picos P y T siempre obtendrán valores por debajo de dicho umbral.

La interfaz que vamos a utilizar consiste en un único registro de Entrada/Salida de 16 bits, de nombre simbólico `REG_ADC`, que contiene el nivel actual de la señal (número entre 0 y 1024, sin unidades). El programa a realizar deberá capturar dicho nivel a una frecuencia suficientemente alta, con el doble fin de poder detectar los picos R (máximos locales) y poder calcular el intervalo entre latidos (IBI: *InterBeat Interval*) con alta precisión (del orden de milisegundos). Para realizar esta captura periódica se deberá utilizar la RSI del *timer 0*, que estará programado a una frecuencia de 500 Hz.

Se dispone de las siguientes rutinas ya implementadas:

Rutina	Descripción
<code>inicializaciones()</code>	Inicializa el <i>hardware</i> (pantalla, interrupciones, timers, etc.)
<code>tareas_independientes()</code>	Tareas que no dependen del cálculo de la frecuencia cardíaca, por ejemplo, cálculo de la presión arterial con otro sensor (tiempo de ejecución máximo = 100 ms)
<code>coherencia_cardiaca(unsigned short ibis[], unsigned char length)</code>	Calcula la coherencia de la variación de la frecuencia cardíaca, a partir de un conjunto de intervalos entre latidos (en milisegundos), pasados por parámetro con el vector <code>ibis[]</code> , de longitud <code>length</code> (t. de ejecución máximo = 24 ms)
<code>visualizar_grafica(unsigned short graph[], unsigned short length, unsigned short last)</code>	Visualiza una gráfica en la pantalla superior de la NDS, donde el parámetro <code>graph[]</code> es un vector con los valores a representar, <code>length</code> es la longitud del vector, y <code>last</code> es el último punto de inserción (índice del vector) en la gráfica (t. de ejecución máximo = 1,5 ms)
<code>swiWaitForVBlank()</code>	Espera hasta el próximo retroceso vertical
<code>printf(char *format, ...)</code>	Escribe un mensaje en la pantalla inferior de la NDS

El diseño del programa a realizar debe permitir entrelazar la ejecución de las tareas independientes con la captura de los niveles de señal, la detección de picos R y el cálculo del tiempo entre picos (IBI).

Los niveles de señal se deben guardar circularmente sobre un vector, para poder visualizarlos continuamente como una gráfica con la rutina `visualizar_grafica()`. Los intervalos entre

latidos también se deben almacenar un otro vector circular, para poder realizar cálculos sobre la variabilidad de la frecuencia cardíaca, con llamadas consecutivas a la rutina `coherencia_cardiaca()`. Estos vectores circulares, junto con sus longitudes e índices de inserción, estarán definidos como sigue:

```
#define MAX_LEVEL 2500           // núm. máximo de niveles
#define MAX_INTERVALS 15          // núm. máximo de intervalos

unsigned short Vlevel[MAX_LEVEL];      // vector de niveles de señal
unsigned short i_level = 0;             // índice actual en Vlevel[]
unsigned short Vintv[MAX_INTERVALS];    // vector de intervalos IBI
unsigned char i_intv = 0;               // índice actual en Vintv[]
```

Además, para poder detectar los máximos locales (picos), se aconseja gestionar dos variables globales que permitan almacenar los dos niveles anteriores de la señal:

```
unsigned short nivel_1, nivel_2;        // niveles anteriores de señal
```

De este modo, se puede detectar un pico R si el nivel actual es superior al valor umbral (550) e inferior al nivel inmediatamente anterior `nivel_1`, al mismo tiempo que `nivel_1` es superior al nivel anterior al anterior `nivel_2`. De este modo, el pico R estará ubicado en el momento de captura anterior al actual.

Por otro lado, a cada nuevo latido se debe escribir por la pantalla inferior de la NDS el último intervalo entre latidos obtenido, expresado en milisegundos, y la frecuencia cardíaca correspondiente a dicho intervalo, expresada en bpm (*beats per minute*, pulsaciones por minuto). A continuación se muestra una salida de ejemplo, con la información de cuatro latidos consecutivos:

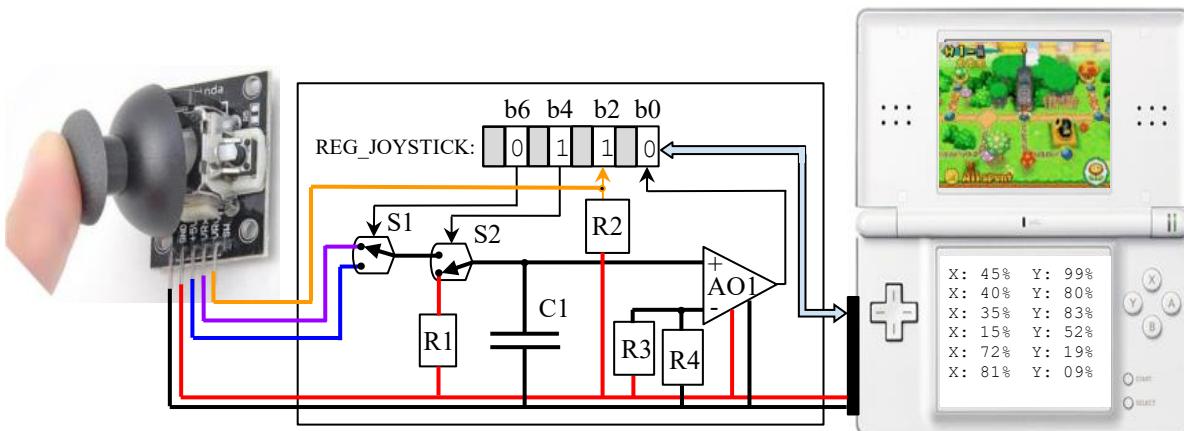
```
:
:
IBI: 997 ms      HR: 60.2 bpm
IBI: 1004 ms     HR: 59.8 bpm
IBI: 1012 ms     HR: 59.3 bpm
IBI: 1024 ms     HR: 58.6 bpm
:
:
```

Se pide:

Programa principal y variables globales adicionales en C, RSI del *timer 0* en ensamblador.

Problema 31 (1^a Conv. 2019-20): Joystick analógico

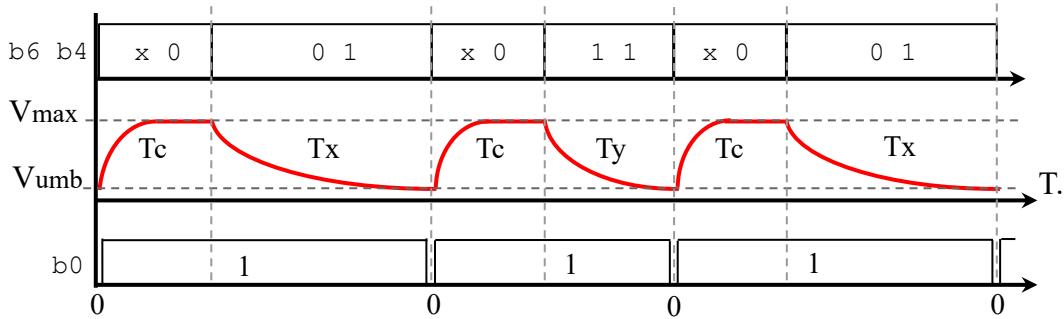
Se propone conectar a la NDS un *thumb joystick* (por ejemplo, [sparkfun.com/9032](https://www.sparkfun.com/9032)). Este dispositivo consta de dos resistencias variables (potenciómetros) que se pueden regular con la inclinación del *stick*, una para cada eje de movimiento (X e Y). Además, dispone de un microinterruptor que se activa pulsando sobre el *stick*. Para medir el valor de las resistencias, se dispone de la siguiente interfaz electrónica:



La medición del valor de cada potenciómetro se basará en el tiempo de descarga del condensador C1 a través de dicho potenciómetro. La interfaz proporciona un único registro de Entrada/Salida de 8 bits, REG_JOYSTICK, que permite la gestión del circuito mediante los siguientes bits:

- b6: controla el *switch* S1, el cual permite conectar el condensador a uno de los dos potenciómetros ($= 0 \rightarrow$ pot.X, $= 1 \rightarrow$ pot.Y),
- b4: controla el *switch* S2, el cual permite conectar el condensador a la resistencia R1 para el ciclo de carga, o a uno de los potenciómetros (según S1) para el ciclo de descarga ($= 0 \rightarrow$ carga, $= 1 \rightarrow$ descarga),
- b2: indica el estado del pulsador ($= 0 \rightarrow$ pulsado, $= 1 \rightarrow$ no pulsado),
- b0: indica si el condensador está prácticamente descargado ($= 0$) o tiene cierto nivel de carga ($= 1$); esta comprobación la realiza el amplificador operacional AO1 que compara el voltaje del condensador con un voltaje umbral (V_{umb}).

El siguiente cronograma muestra un ejemplo de operativa del condensador:



En la secuencia anterior, los bits *b6* y *b4* se han manipulado de modo que el voltaje del condensador ha variado según la gráfica intermedia y el bit *b0* ha marcado el final de los tiempos *Tx*, *Ty*. Esta secuencia se puede representar con los siguientes estados:

<i>estado</i>	<i>b6 b4</i>	<i>Tiempo</i>	<i>Descripción</i>
0	x 0	<i>Tc</i>	carga del condensador durante un tiempo fijo
1	0 1	<i>Tx</i>	descarga del condensador a través del potenciómetro X, durante un tiempo variable (hasta que <i>b0</i> = 0)
2	x 0	<i>Tc</i>	carga del condensador durante un tiempo fijo
3	1 1	<i>Ty</i>	descarga del condensador a través del potenciómetro Y, durante un tiempo variable (hasta que <i>b0</i> = 0)
0	x 0	<i>Tc</i>	carga del condensador durante un tiempo fijo
1	0 1	<i>Tx</i>	descarga del condensador a través del potenciómetro X, durante un (nuevo) tiempo variable (hasta que <i>b0</i> = 0)
etc.			

El tiempo de carga *Tc* se fijará en 10 milisegundos, para asegurar que el voltaje del condensador llega a su valor máximo (V_{\max}). La carga y descarga de un condensador sigue una ley exponencial que depende de los valores de las resistencias y la capacidad del condensador. Según la variación de resistencia de los potenciómetros, los tiempos de descarga *Tx*, *Ty* oscilarán entre 20 y 120 milisegundos (aprox.).

Se pide utilizar la RSI del *timer 0* para fijar el valor de los bits *b6* y *b4*, así como para detectar el momento en que el bit *b0* pasa de 1 a 0, con el fin de obtener los tiempos de descarga. Para que dichos tiempos tengan una precisión suficientemente alta, el *timer 0* estará programado a una frecuencia de 500 Hz.

Se dispone de las siguientes rutinas ya implementadas:

Rutina	Descripción
inicializaciones ()	Inicializa el <i>hardware</i> (pantalla, interrupciones, timers, etc.)
tareas_principales (unsigned char por_x, unsigned char por_y, unsigned char puls)	Tareas principales del programa (gestión de un juego); recibe por parámetro los porcentajes de las posiciones x e y del joystick, además de un booleano indicando si el botón está pulsado (tiempo de ejecución < 100 ms)
swiWaitForVBlank ()	Espera hasta el próximo retroceso vertical
printf (char *format, ...)	Escribe un mensaje en la pantalla inferior de la NDS

El programa a implementar consistirá en un juego, el cual se gestionará mediante llamadas consecutivas a la rutina `tareas_principales()`. Concurrentemente, la RSI deberá ir generando los estados de carga del condensador y descarga a través de los potenciómetros X e Y, alternativamente, además de capturar el estado del pulsador.

Después de cada ciclo completo de obtención de Tx y Ty, dichos tiempos se deberán convertir en un valor de porcentaje de variación del joystick en cada eje, donde 0% corresponderá a la posición mínima (izquierda en X, inferior en Y), 50% a la posición central y 100% a la posición máxima (derecha en X, superior en Y).

Para realizar dicha conversión será necesario disponer de los tiempos mínimos y máximos correspondientes a las posiciones mínimas y máximas de cada eje. Estos valores se almacenarán en las siguientes variables:

```
unsigned char tx_min, tx_max;           // tiempos límite (ms)
unsigned char ty_min, ty_max;
```

Para obtener estos valores con exactitud, ya que pueden variar respecto a los valores orientativos 20 y 120, el programa deberá realizar un proceso de calibración del joystick antes de que empiece el juego. Este proceso consiste en solicitar al usuario que posicione el joystick en la posición mínima y que pulse el botón del joystick. Cuando esto ocurra, habrá que memorizar los tiempos mínimos de cada eje. Este procedimiento se debe repetir para memorizar los tiempos máximos de cada eje.

La transformación continua de los tiempos variables Tx, Ty en valores de porcentaje se debe realizar con una regla de proporcionalidad entre el valor de tiempo mínimo y el valor de tiempo máximo ($T = t_{\min} \rightarrow 0\%$, $T = t_{\max} \rightarrow 100\%$). Esta transformación se debe realizar en el programa principal, para no sobrecargar innecesariamente el procesador dentro de la ejecución de la RSI.

Por otro lado, cada vez que se realice una transformación de los dos tiempos variables, se deberán escribir los valores de porcentaje y un booleano que indique el estado del botón (0 → soltado, 1 → pulsado) por la pantalla inferior de la NDS, mientras que el juego se visualizará en la pantalla superior. A continuación se muestra una salida de ejemplo:

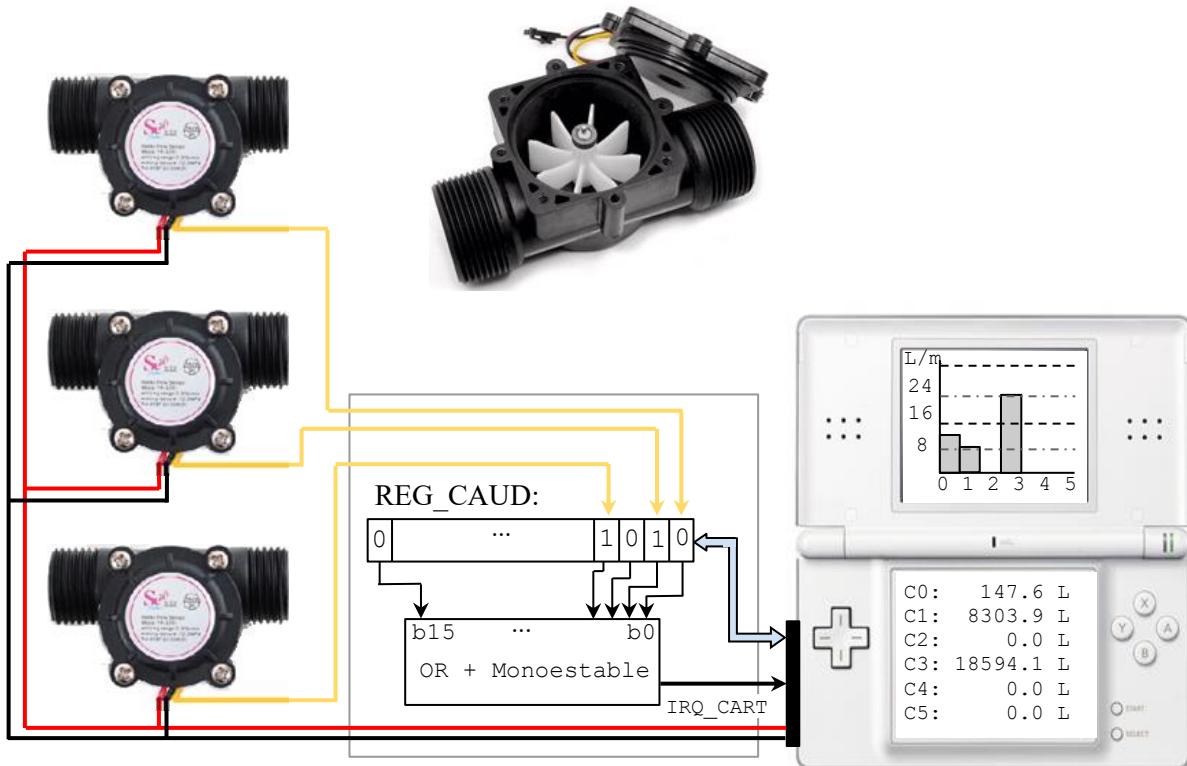
:
X: 45% Y: 100% P: 0
X: 40% Y: 80% P: 0
X: 35% Y: 83% P: 1
X: 15% Y: 52% P: 1
:

Se pide:

Programa principal y variables adicionales en C, RSI del *timer 0* en ensamblador.

Problema 32 (2^a Conv. 2019-20): Caudalímetros

Se propone conectar a la NDS un conjunto de hasta 16 caudalímetros tipo YF-S201 (hoja de especificaciones en hobbytronics.co.uk/datasheets/sensors/YF-S201.pdf). Este tipo de sensores constan de una hélice dentro de un receptáculo que se conecta a una tubería de 1/2 pulgada de diámetro. El paso del líquido hace girar la hélice, la cual incorpora un imán que provoca pulsos electrónicos sobre un transistor (efecto Hall). El transistor y el resto de los componentes electrónicos están fuera del receptáculo, de modo que no entran en contacto con el líquido. A continuación se muestra la foto de un sensor con el receptáculo abierto, junto con un esquema de conexión de varios caudalímetros con una interfaz electrónica que se conecta al puerto GBA de la NDS:



La interfaz proporciona un único registro de 16 bits, de nombre simbólico REG_CAUD. El propósito de este registro es doble:

- Lectura: en cada bit se puede leer un 1 si el caudalímetro que está conectado a ese bit ha generado un pulso desde la última vez que se reinició el bit,
- Escritura: se pueden poner a 0 un conjunto de bits del registro, escribiendo una máscara con un 1 en cada bit a reiniciar.

Además, la interfaz incluye una puerta OR que agrupa todos los bits del registro, de modo que genera una petición de interrupción (IRQ_CART) cuando hay algún bit a 1. Un circuito monoestable adicional asegura que, cuando se escribe el registro, la señal IRQ_CART se desactiva durante unos microsegundos, para volver a reactivarse en caso de que se hubieran

quedado algunos bits a 1 sin reiniciar.

El programa a implementar debe visualizar, en todo momento, los litros que han pasado por cada caudalímetro desde que se inició dicho programa (consumo), además de visualizar gráficamente una estimación del flujo instantáneo (caudal) del líquido que pasa por cada caudalímetro.

Para poder realizar los cálculos a partir del número y la frecuencia de pulsos (Freq) que genera cada caudalímetro, el fabricante nos proporciona los siguientes datos:

- 1 pulso \approx 2,22 mililitros
- 1 litro \approx 450 pulsos
- Flujo (litros/hora) \approx Freq (Hz) * 7,5
- Rango del flujo (litros/hora) \approx [100..1800]

Sin embargo, queremos visualizar el flujo en litros por minuto, de modo que vamos a trabajar con las siguientes conversiones:

- Flujo (litros/minuto) \approx Freq (Hz) * 7,5 / 60 = Freq (Hz) / 8
- Rango del flujo (litros/minuto) \approx [1..30]

Se dispone de las siguientes rutinas ya implementadas:

Rutina	Descripción
inicializaciones ()	Inicializa el <i>hardware</i> (pantalla, interrupciones, etc.)
initializar_timer0 (unsigned char preescalar, short divFreq)	Inicializa el <i>timer</i> 0, seleccionando una de las 4 posibles frecuencias de entrada según el parámetro preescalar (0: 33513,98 kHz; 1: 523,656 kHz, 2: 130,914 kHz, 3: 32,7285 kHz) y utilizando el divisor de frecuencia indicado en divFreq.
tareas_independientes ()	Tareas que no dependen del cálculo del caudal ni del consumo, por ejemplo, control de la presión de cada tubería (tiempo de ejecución < 50 ms)
grafico_barras (unsigned short bars[], unsigned char num_bars, unsigned short max_val)	Dibuja en la pantalla superior de la NDS un gráfico de barras según los valores del vector bars[], con número de elementos num_bars, escalando la altura de las barras respecto a un valor máximo max_val (tiempo de ejecución < 5 ms)
swiWaitForVBlank ()	Espera hasta el próximo retroceso vertical
printf_XY(unsigned char x, unsigned char y, char *format, ...)	Escribe un mensaje en la pantalla inferior de la NDS, a partir de la posición de pantalla con coordenadas x (núm. columna, de 0 a 31) e y (núm. fila, de 0 a 23) (tiempo de ejecución < 100 µs)

Con el apoyo de las rutinas anteriores, el programa a realizar debe ejecutar las tareas independientes concurrentemente con el conteo de pulsos de los caudalímetros conectados a la interfaz, que se deberá realizar con la RSI del dispositivo. Para obtener la frecuencia de los

pulsos de cada caudalímetro, se deberá utilizar la RSI del *timer 0* configurada a un hercio. Además, cada segundo se deberá representar gráficamente el flujo de cada caudalímetro con un gráfico de barras (pantalla superior), y el conteo total de litros de cada caudalímetro como una lista de números (pantalla inferior), con un formato como el del ejemplo (para 6 caudalímetros):

```
C0:    147.6 L  
C1:  8303.9 L  
C2:      0.0 L  
C3: 18594.1 L  
C4:      0.0 L  
C5:      0.0 L
```

Para gestionar el valor de los contadores y de los caudales se pide usar las siguientes variables y definición:

```
#define MAX_CAUD 6  
unsigned int cont_puls[MAX_CAUD];           //contadores de pulsos totales  
unsigned short caudal[MAX_CAUD];            // caudales (litros/minuto)
```

La definición `MAX_CAUD` permitirá adaptar el código fuente al número máximo de caudalímetros de cada instalación, con el fin de que el programa no tenga que realizar recorridos de vector innecesariamente largos. Se supone que los caudalímetros se conectarán todos en los bits bajos del registro `REG_CAUD`.

Para obtener el valor de los contadores con la máxima exactitud posible, se sugiere utilizar contadores de pulsos en vez de contadores de litros, puesto que la conversión de pulsos a litros es aproximada, de modo que es preferible realizarla con el máximo número pulsos posible (el total) para evitar pérdida de precisión.

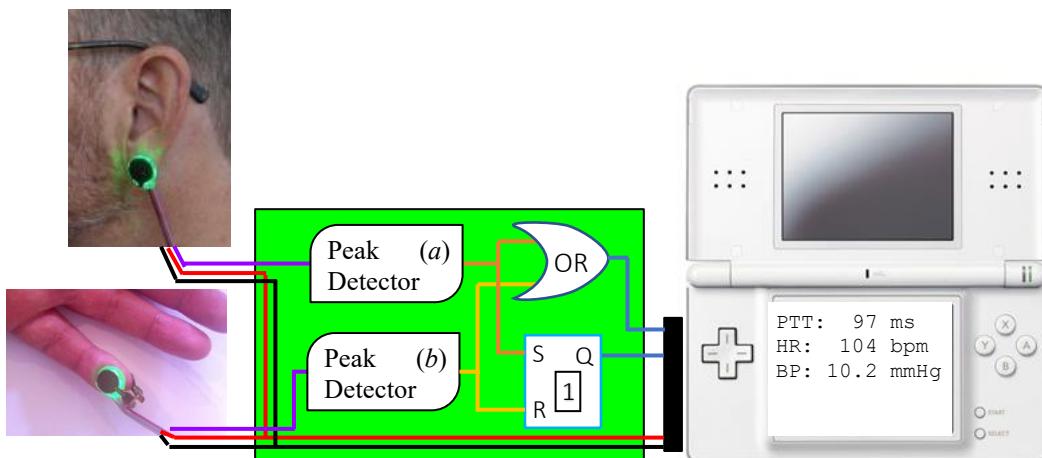
Por otro lado, el vector de valores de caudal podría haberse definido como `unsigned char` en vez de `unsigned short`, dado el rango de trabajo del flujo en litros por minuto. Sin embargo, se ha decidido usar un tipo de datos más grande de lo necesario para que dicho vector se pueda pasar directamente por parámetro a la rutina auxiliar `grafico_barras()`. Por último, aunque el rango de trabajo del valor del flujo sea [1..30], vamos a suponer que el valor cero también será admisible como resultado de los cálculos.

Se pide:

Programa principal y variables adicionales en C, RSI del dispositivo y RSI del *timer 0* en ensamblador.

Problema 33 (1^a Conv. 2020-21): Tiempo de tránsito del pulso

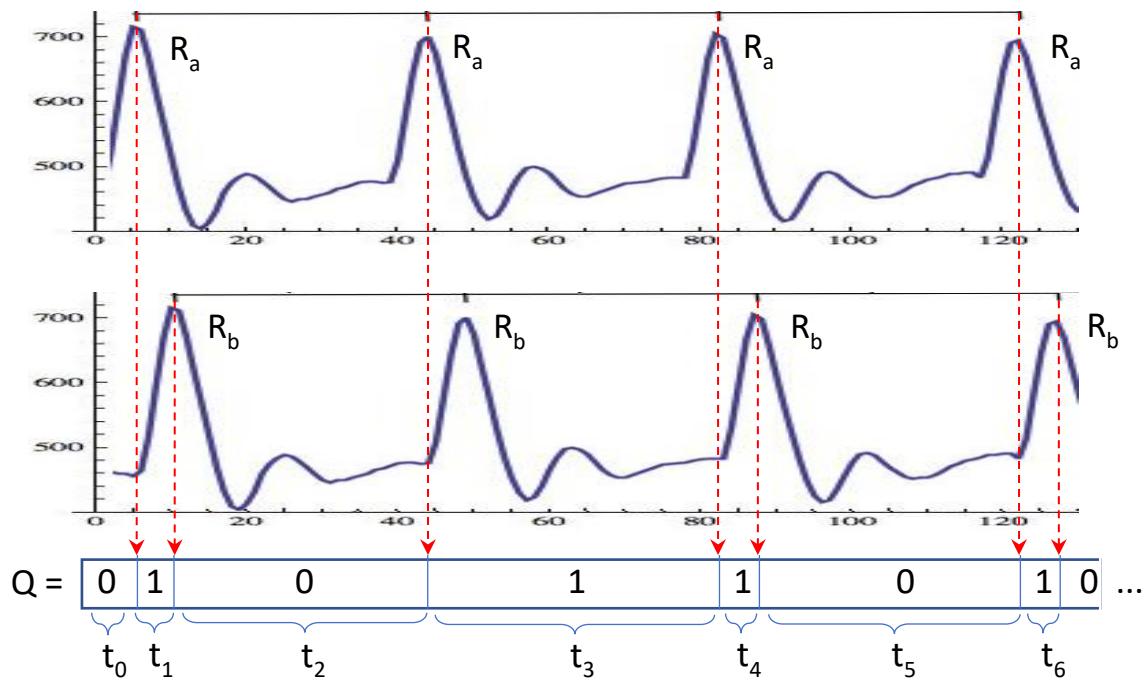
Se propone conectar a la NDS una interfaz electrónica equipada con dos sensores de pulso cardíaco (por ejemplo, del fabricante pulsesensor.com). Uno de los sensores se colocará en el lóbulo de la oreja izquierda y el otro en el extremo de un dedo de la mano izquierda. El siguiente esquema muestra la conexión de los sensores con una interfaz electrónica diseñada expresamente para comunicarse con la NDS:



En dicha interfaz hay un circuito electrónico *Peak Detector* para cada sensor, capaz de detectar el pico R de la señal generada por el sensor (ver gráficos en la siguiente página). La salida de cada detector de picos es un pulso electrónico de unos 5 microsegundos de duración, cuyo propósito principal es generar una interrupción en el computador. Debido a que solo disponemos de un único pin para enviar señales de petición de interrupción (IRQ_CART), las salidas de los dos detectores se han unido con una puerta OR, de modo que se generará una petición de interrupción si se detecta un pico R en alguno de los dos sensores.

Además, se ha incorporado un biestable S-R (Set-Reset) asíncrono, que memoriza un 1 cuando S = 1 y memoriza un 0 cuando R = 1. Cuando las dos entradas S y R valen 0, el biestable mantiene el último valor memorizado (en el esquema se muestra un 1, pero puede ser un 0). La salida Q envía el contenido del biestable hacia el computador mediante el bit 7 de un registro de Entrada/Salida de 16 bits mapeado en la dirección absoluta de memoria 0x0A000000. Además, puede ser que los otros bits del mismo registro se utilicen para otras tareas que no se detallan en este enunciado (tareas independientes).

El gráfico de la página siguiente muestra un ejemplo de evolución temporal (cronograma) de las señales de los sensores. Como los sensores se encuentran en posiciones del cuerpo separadas, las ondas de presión generadas por el latido del corazón siempre llegan a los sensores en momentos distintos. El propósito principal del programa a implementar será calcular la diferencia de tiempo en la detección de los picos R_a y R_b de un mismo latido, lo cual se denomina **tiempo de tránsito del pulso** (PTT: Pulse Transit Time).



(gráfico basado en contenidos de <https://community.wolfram.com/groups/-/m/t/272663>)

En el gráfico anterior se muestra la generación de las interrupciones debidas a la detección de un pico R_a o R_b , mediante flechas rojas de línea discontinua. También se muestra la evolución del valor de la salida Q del biestable S-R, así como una representación simbólica de los tiempos entre interrupción e interrupción (t_0 , t_1 , t_2 , etc.).

Como se puede observar, se ha omitido la interrupción correspondiente al segundo pulso R_b . Esta omisión representa la posibilidad de que cualquiera de los dos detectores de pico falle puntualmente, puesto que existen múltiples factores que pueden perturbar el funcionamiento del circuito (ruido de señal, mala colocación del sensor, etc.). Sin embargo, vamos a suponer que nunca fallarán los dos sensores simultáneamente (en el mismo latido).

Para mantener la máxima precisión posible en los cálculos posteriores, se pide explícitamente que se ignoren los valores erróneos. Para ello, hay que comprobar si el valor Q ha cambiado respecto a la interrupción anterior. En el ejemplo anterior, cuando el detector de picos b falla, Q no cambia porque la entrada Reset del biestable no se activa, mientras que la entrada Set del biestable recibe dos activaciones consecutivas. En este caso, el valor de t_3 se debe descartar.

El programa a realizar deberá capturar todos los tiempos válidos entre picos R , distinguiendo entre tiempos cortos y tiempos largos. Vamos a suponer que los tiempos cortos siempre serán los capturados con los picos R_b , es decir, cuando el biestable pase de 1 a 0, mientras que los tiempos largos serán los casos opuestos. Todos estos tiempos se irán almacenando en sendos vectores circulares de MAX_T posiciones cada uno (ver definiciones de variables globales).

Cuando se hayan capturado como mínimo $\text{MAX_T}/2$ pulsos en ambos sensores, el programa deberá realizar la media aritmética de los últimos MAX_T tiempos cortos y largos (media móvil). El programa debe escribir por pantalla el tiempo de tránsito de pulso (PTT), que será igual a la media móvil de los tiempos cortos, expresada en milisegundos. Además, el programa calculará la frecuencia de los latidos del corazón (HR: *heart rate*), escribiendo por pantalla su valor entero expresado en bpm (*beats per minute*: latidos por minuto).

El cálculo de la frecuencia cardíaca se puede obtener teniendo en cuenta que el periodo de un latido es la suma del tiempo corto más el tiempo largo correspondiente. El programa también realizará una estimación de la presión arterial (BP: *blood pressure*) a partir de los valores actuales de PPT y HR, utilizando una rutina de soporte ya implementada. El programa escribirá por pantalla el valor de BP en milímetros de mercurio (mmHg), utilizando un dígito decimal.

Los valores de PTT, HR y BP se escribirán cada uno en una línea individual, añadiendo una línea de separación respecto a los valores calculados anteriormente. Cada vez que se calculan los nuevos valores de estos parámetros se denomina “un ciclo de cálculo”. En el esquema inicial se muestra un ejemplo de salida de información por pantalla para un único ciclo de cálculo.

El diseño del programa a realizar debe permitir entrelazar la captura de los tiempos entre picos con la ejecución de las tareas independientes a dicha captura. Para ello es necesario utilizar la rutina de servicio de interrupción de las peticiones generadas por la interfaz electrónica, que denominaremos como `RSI_Peak`. Para calcular los tiempos entre picos se deberá utilizar un cronómetro de precisión que usará los *timers* 1 y 2 de forma encadenada.

Se dispone de las siguientes rutinas ya implementadas:

Rutina	Descripción
<code>inicializaciones()</code>	Inicializa el <i>hardware</i> (pantalla, interrupciones, <i>timers</i> , etc.)
<code>tareas_independientes()</code>	Tareas que no dependen del cálculo del tiempo de tránsito del pulso, por ejemplo, cálculo del nivel de oxígeno en sangre (tiempo de ejecución < 100 ms)
<code>calcular_presion(unsigned short pulse_tt, float h_rate)</code>	Calcula la presión arterial a partir del tiempo de tránsito del pulso (en ms) y de la frecuencia cardíaca (en bpm), devolviendo el resultado en mmHg (float, tiempo de ejecución < 350 microseg.)
<code>cpuStartTiming(int timer)</code>	Inicia un cronómetro de precisión usando el <i>timer</i> que se pasa por parámetro (0, 1 o 2) y el siguiente <i>timer</i>
<code>cpuGetTiming()</code>	Devuelve el conteo de ticks del cronómetro de precisión desde que se inició (unsigned int)

swiWaitForVBlank()	Espera hasta el próximo retroceso vertical
printf(char *format, ...)	Escribe un mensaje en la pantalla inferior de la NDS

Además, se propone el uso de las siguientes constantes y variables globales:

```
#define MAX_T     8                      // número máximo de tiempos
#define TICS_MS 33513                    // tics por cada milisegundo

unsigned int vect_tc[MAX_T];           // vector de tiempos cortos
unsigned int vect_tl[MAX_T];           // vector de tiempos largos
unsigned char ind_tc = 0;              // índice actual en vect_tc
unsigned char ind_tl = 0;              // índice actual en vect_tl
unsigned char num_tc = 0;              // número de tiempos cortos
unsigned char num_tl = 0;              // número de tiempos largos
```

Las variables `ind_tc` e `ind_tl` permitirán indexar los vectores para saber en qué posición se debe guardar un nuevo tiempo capturado. Por otro lado, las variables `num_tc` y `num_tl` permitirán saber cuántos tiempos se han guardado desde el último ciclo de cálculo. Como los vectores serán circulares se puede suponer que siempre estarán llenos, de modo que estas variables no se deben interpretar como el número total de valores dentro de los vectores, sino como el número de valores capturados en el ciclo de cálculo actual.

Por otro lado, la definición `TICS_MS` corresponde al número de tics contabilizados por el cronómetro de precisión en un milisegundo.

A modo de ejemplo se muestran algunos rangos típicos de los valores que tiene que manejar el programa a implementar:

- Tiempo corto (PTT): 60 – 260 ms
- Tiempo largo: 210 – 1240 ms
- Frecuencia cardíaca (HR): 40 – 220 bpm
- Presión arterial (BP): 70 – 160 mmHg

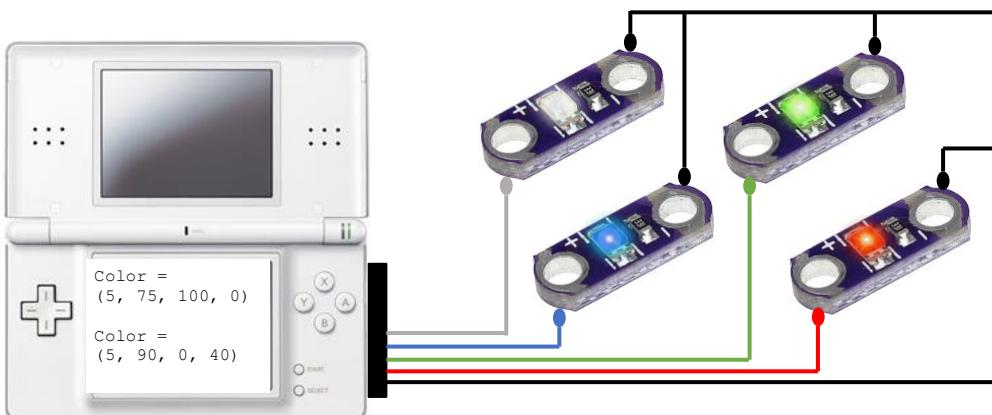
En el contexto de este problema, la presión arterial se refiere a la presión intermedia entre la sistólica (máxima) y la diastólica (mínima).

Se pide:

Programa principal y variables globales adicionales en C, RSI del dispositivo en ensamblador.

Problema 34 (2^a Conv. 2020-21): Combinaciones de color

Se propone conectar a la NDS un conjunto de minileds de la serie Arduino LilyPad® ([Sparkfun](#)), para generar un color determinado a partir de la combinación de intensidades de los distintos colores base de los leds. Cada led tiene un color base determinado (blanco, azul, verde, rojo, etc.), pero si se juntan varios leds y se regula la intensidad de luz de cada tipo, se puede formar un color combinado, por ejemplo, un verde Esmeralda, que se podría conseguir fijando las siguientes intensidades: blanco → 5%, azul → 75%, verde → 100%, rojo → 0%.



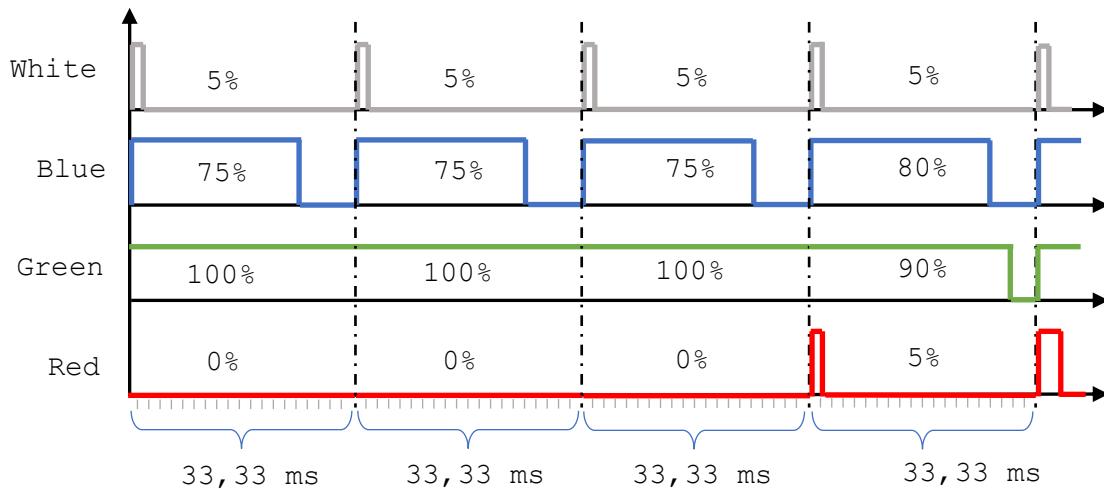
El esquema anterior muestra una conexión de la NDS con cuatro tipos de leds. Sin embargo, existe toda una gama de colores base para los leds, de modo que el programa de control deberá estar preparado para manejar un número predeterminado de tipos de led, que podrá variar entre 1 y 6 tipos.

Hay un cable de masa común (negro) para todos los leds, y un cable de señal por cada tipo de led. Cada cable de señal solo puede valer 0 (apagado) o 1 (encendido). Los cables de señal estarán conectados con los bits de menor peso de un registro de entrada/salida de 16 bits, de nombre simbólico REG_IO. Por ejemplo, se puede utilizar el bit 0 para el rojo, el bit 1 para el verde, el bit 2 para el azul y el bit 3 para el blanco. Obviamente, el número de bits utilizados dependerá del número de tipos distintos de led a controlar. El resto de bits del registro pueden tener otras funciones (para tareas independientes), de modo que no se deberán modificar.

Al encender un led, éste emite luz a su máxima intensidad. Para regular su nivel de intensidad podemos usar la técnica de la modulación por ancho de pulso (PWM: *Pulse Width Modulation*), que consiste en encender el led durante una fracción del tiempo, según el porcentaje de la intensidad que se requiera.

Para que esta técnica funcione, el proceso debe ser repetido a intervalos muy breves de tiempo para el ser humano. En el contexto del problema actual, se establece que los intervalos serán de 33,33 milisegundos, aproximadamente. Cada uno de estos intervalos estará dividido en 20 subintervalos, de modo que se podrán generar 21 niveles de intensidad entre el 0% y el 100%, en incrementos de 5%.

El siguiente cronograma muestra cuatro intervalos con diferentes ejemplos de intensidad para 4 tipos de led. El valor del bit de control de cada tipo de led se indica con un nivel bajo (0) o alto (1) de la señal, para diferentes momentos en el tiempo. Los 3 primeros intervalos repiten los mismos valores de intensidad, mientras que el cuarto intervalo introduce pequeñas variaciones. Estos ejemplos muestran diferentes formas de onda (anchos de pulso) para cada tipo de led.



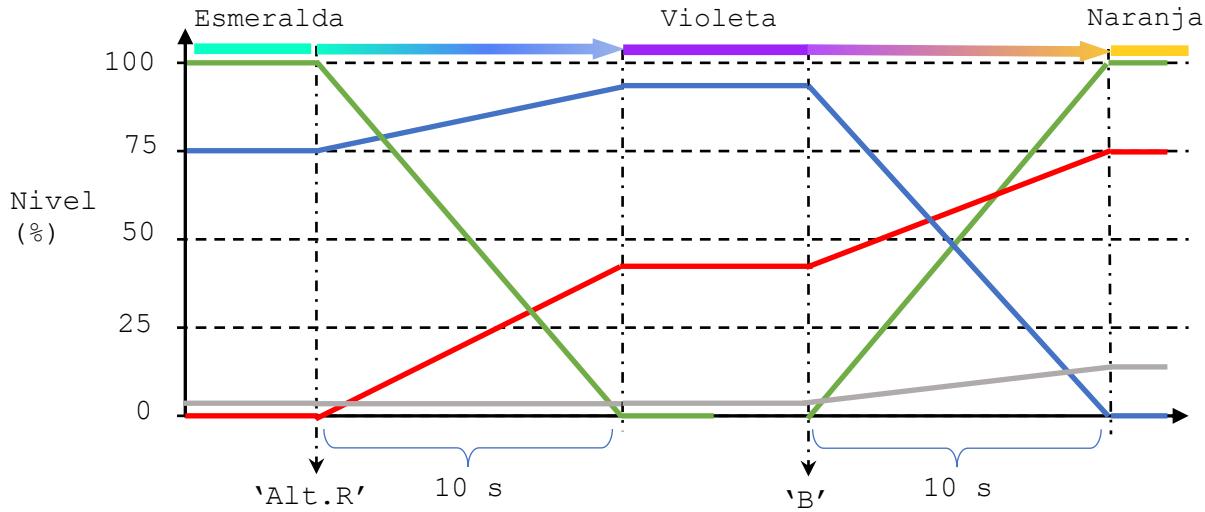
El programa a realizar debe controlar periódicamente el valor de los bits de cada tipo de led, utilizando para ello la rutina de servicio de interrupción del *timer 0*, que se programará a la frecuencia adecuada.

El programa también permitirá al usuario cambiar el color generado. Para ello se definirá un vector con diez combinaciones de niveles de intensidad. Las siguientes líneas de código fuente en C son un ejemplo de definición de este vector:

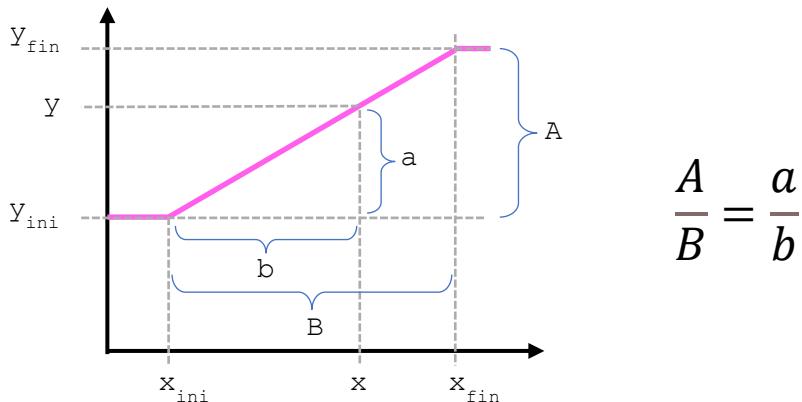
```
#define NUM_LEDS      4          // número de leds
                                // vector colores preestablecidos (R, G, B, W)
unsigned char colores[10][NUM_LEDS]={ {100, 0, 0, 0},           // Rojo
                                      {75, 100, 0, 15},      // Naranja
                                      {100, 100, 0, 0},      // Amarillo
                                      {15, 65, 0, 20},       // Caqui
                                      {0, 100, 0, 0},        // Verde
                                      {0, 100, 75, 5},       // Esmeralda
                                      {0, 15, 70, 30},       // Azul claro
                                      {0, 0, 100, 0},        // Azul
                                      {40, 0, 90, 5},        // Violeta
                                      {0, 0, 0, 100}         // Blanco
};
```

El usuario podrá cambiar de combinación pulsando alguno de los botones de la NDS. Al pulsar un botón, el programa deberá pasar del color actual al solicitado por el usuario de manera gradual, durante un periodo de tiempo perceptible, por ejemplo, 10 segundos.

El siguiente esquema muestra dos ejemplos de variaciones de color, al pulsar dos botones ('Alt. Right', 'B') en diferentes momentos:



Para conseguir generar los niveles intermedios durante el tiempo que dure la transición de colores, hay que usar interpolación. Para calcular el valor interpolado de la intensidad de cada tipo de led, se puede usar la propiedad de conservación de la ratio entre dos lados homólogos de dos triángulos semejantes. El siguiente esquema muestra cómo identificar estos dos triángulos en un gráfico de interpolación, junto con la fórmula de la propiedad:



El valor a calcular es y (nivel de intensidad) respecto de x (tiempo). Los niveles inicial y final se representan como y_{ini} e y_{fin} . Los valores concretos de tiempo x_{ini} y x_{fin} no son relevantes, puesto que lo importante es contar el tiempo relativo (b) desde que se ha pulsado el botón. Para contar el tiempo, se sugiere utilizar un contador de los intervalos usados en la generación de los pulsos de control PWM. Por ejemplo, para medir el paso de 10 segundos (aprox.) se deberán contar 300 intervalos.

Se recomienda que el control de la gradación se realice desde el programa principal. Sin embargo, será necesario que exista una sincronización con la RSI del *timer 0*, puesto que el programa principal **no** debe cambiar los niveles de intensidad en medio de un intervalo de generación de pulsos PWM. Para realizar esta sincronización de forma eficiente, se puede usar la rutina `swiWaitForIRQ()`, que pone el procesador en reposo hasta que se produzca

una interrupción cualquiera.

Después de cada proceso de cambio de color, por la pantalla inferior de la NDS se deberán mostrar los nuevos niveles de intensidad de cada tipo de led, con el siguiente formato:

```
color = (Pn-1, Pn-2, ..., P0)
```

donde P_i es el porcentaje de intensidad del led i -ésimo. También hay que dejar una línea vacía respecto a la línea del color anterior. A modo de ejemplo, a continuación se muestran dos escrituras consecutivas, correspondientes a los colores Esmeralda y Violeta:

```
color = (5, 75, 100, 0)
```

```
color = (5, 90, 0, 40)
```

Por último, el programa debe ir entrelazando la generación de los colores con la ejecución de unas tareas independientes. Sin embargo, mientras se esté realizando la gradación de colores, no será necesario invocar a las tareas independientes.

Se dispone de las siguientes rutinas ya implementadas:

Rutina	Descripción
inicializaciones ()	Inicializa el <i>hardware</i> (pantalla, interrupciones, etc.)
activar_timer0(int freq)	Activa el funcionamiento del <i>timer</i> 0 a la frecuencia indicada por parámetro (en Hz)
tareas_independientes ()	Tareas que no dependen de la gestión de los colores, por ejemplo, cálculo de la humedad relativa del aire (tiempo de ejecución < 100 ms)
scanKeys ()	Captura el estado actual de los botones de la NDS
keysDown ()	Devuelve un patrón de bits (int) con los botones activos
indice_boton(int keys)	Devuelve un índice entre 0 y 9 (unsigned char) de botón, según un patrón de bits <i>keys</i> que indica los botones activos; si no hay ningún botón activo, devuelve 255.
swiWaitForIRQ()	Espera hasta la próxima interrupción
swiWaitForVBlank()	Espera hasta el próximo retroceso vertical
printf(char *format,...)	Escribe un mensaje en la pantalla inferior de la NDS

Además, se propone el uso de las siguientes constantes y variables globales (junto con las sugeridas anteriormente):

```
#define BITS_LEDS      0xF          // bits de los leds a 1 (4 bits)
#define NUM_INTERPOL   300          // número de interpolaciones

unsigned char vectPWM[NUM_LEDS];    // vector de porcentajes PWM
```

La definición `NUM_INTERPOL` corresponde al número de intervalos PWM necesarios para realizar todas las interpolaciones de una gradación de colores.

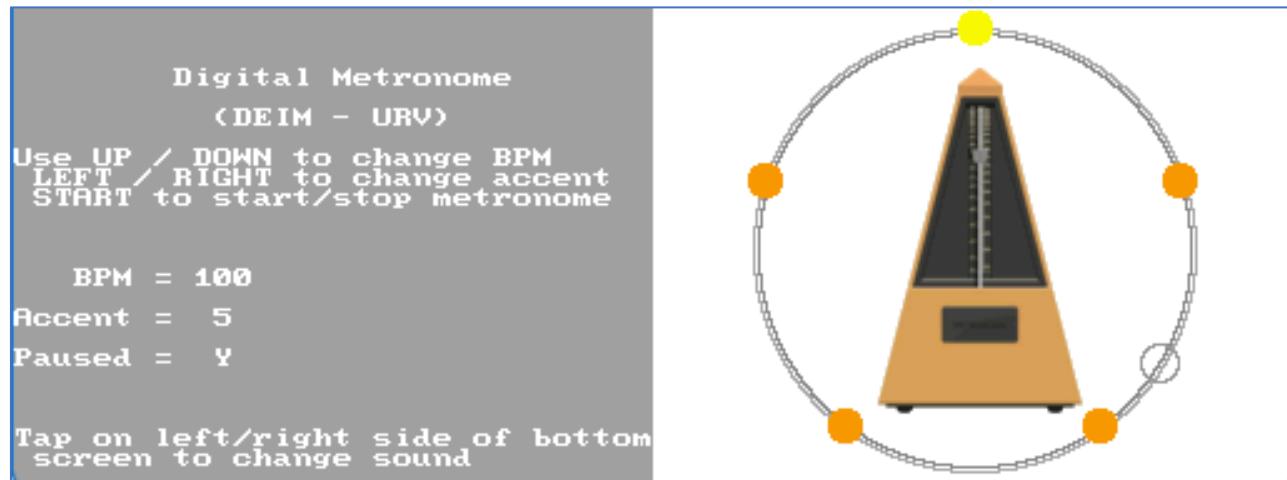
El vector `vectPWM[NUM_LEDS]` permitirá almacenar los niveles de intensidad actuales de los distintos tipos de led. Hay que recordar que `NUM_LEDS` es una constante (*define*) que se puede cambiar antes de cada compilación. Esto significa que el programa debe estar diseñado para poder ser recompilado con cualquier número de tipos de led (no tiene por qué ser 4).

Se pide:

Programa principal y variables adicionales en C; RSI del *timer* 0 en ensamblador.

Problema 35 (1^a Conv. 2021-22): Metrónomo digital

Se propone implementar un metrónomo sobre la plataforma NDS, es decir, un programa para marcar pautas sonoras periódicas. Para realizar este programa no se necesita ningún hardware adicional, puesto que solo se requiere de las pantallas, los botones y el sonido de la NDS. La interfaz de usuario propuesta es la siguiente (las dos pantallas, una al lado de la otra):



El usuario podrá fijar el valor BPM ("beats per minute" o pulsaciones por minuto), entre 10 y 180, usando los botones UP y DOWN, en saltos de 10 unidades.

Usando los botones LEFT y RIGHT, el usuario también podrá fijar un número para formar un grupo de varios "beats". La primera marca sonora del grupo será diferente del resto. A esta marca inicial la denominaremos *acento*. Por ejemplo, para un grupo de 4 "beats", sonará una marca sonora de acento y tres marcas sonoras normales. Después se repetirá el proceso para el siguiente grupo. Esto permitirá, por ejemplo, marcar el inicio de un compás musical. Los grupos de "beats" podrán ser entre 1 y 8, en saltos de 1 unidad.

En el gráfico de la pantalla derecha los "beats" del grupo se representan con círculos, uno amarillo para el "beat" de acento y el resto en color naranja. La circunferencia grande representa el tiempo total de un grupo. La circunferencia pequeña irá girando continuamente en el sentido de las agujas del reloj, para indicar el paso del tiempo. Cada vez que la circunferencia pequeña pase por encima de un círculo, sonará el "beat" correspondiente (de acento o normal).

El número de "beats" por minuto será independiente del número de "beats" por grupo, es decir, el BPM determinará la frecuencia de los "beats" individuales, no la frecuencia de los grupos (no cuantos acentos por minuto).

El usuario podrá cambiar los valores de BPM y número de "beats" por grupo en cualquier momento, de modo que el programa adaptará la reproducción de los "beats" dinámicamente.

Por último, el usuario podrá iniciar y parar el metrónomo en cualquier momento, pulsando el

botón START sucesivas veces.

Se dispone de las siguientes rutinas ya implementadas:

Rutina	Descripción
inicializaciones ()	Inicializa el <i>hardware</i> (pantalla, interrupciones, etc.)
tareas_independientes ()	Tareas que no dependen de la reproducción de las marcas sonoras ni de la representación gráfica de dichas marcas, por ejemplo, detección del compás en sonido capturado por el micrófono (tiempo de ejecución < 250 ms)
gestionar_botones ()	Detecta los botones de la NDS y actualiza las variables de configuración en consecuencia (var. globales); devuelve un valor booleano (<i>unsigned char</i>), que será diferente de cero si se ha cambiado el BPM o el número de "beats" por grupo
swiWaitForVBlank ()	Espera hasta el próximo retroceso vertical
actualizar_pantallas ()	Actualiza la visualización de las variables de configuración y de los círculos que indican las posiciones de las marcas sonoras, en las pantallas correspondientes de la NDS
activar_beat ()	Si al llamar a esta función la circunferencia pequeña está encima de un círculo, activa el sonido correspondiente (ácento o normal); aunque el sonido se reproducirá durante decenas de milisegundos, esta rutina retorna en menos de 5 microsegundos
SPR_moverSprite (int indice, int px, int py)	Mueve el <i>sprite</i> cuyo índice se indica por primer parámetro a las coordenadas de pantalla que se indican con los otros dos parámetros
SPR_actualizarSprites (u16* base, int limite)	Modifica los registros OAM de uno de los procesadores gráficos, según la dirección inicial de OAM que se pasa por primer parámetro, desde el <i>sprite</i> 0 hasta el <i>sprite</i> <i>limite-1</i> (segundo parámetro menos 1).

La tarea a programar consiste básicamente en mover la circunferencia pequeña sobre el perímetro de la circunferencia grande, a la velocidad adecuada para que pase por encima de cada círculo de marca sonora a la frecuencia que indique el valor actual de BPM. La circunferencia pequeña solo se debe mover si el metrónomo no está en pausa.

Se pide explícitamente que el movimiento de dicha circunferencia pequeña, que será visualizada con el *sprite* 0 (ya inicializado), se gestione desde la RSI del *Vertical Blank*. Se propone el siguiente programa principal, junto con las variables globales básicas:

```
typedef struct          // coordenadas de sprites
{
    short px;           // [-64..288]
    short py;           // [-32..224]
} t_pos;

t_pos ang_pos[360];   // vector con posiciones (px, py) para cada ángulo

unsigned char bpm = 60;          // "beats" por minuto
unsigned char accent = 4;        // divisiones para generar el acento
unsigned char paused = !0;       // metrónomo pausado? (!0: sí, 0: no)

unsigned int ang_actual = 0;     // ángulo actual (en formato Q12)
unsigned int fraccion = ???;    // incremento del ángulo en un VBL (Q12)

int main(void)
{
    inicializaciones();
    do
    {   tareas_independientes();
        if (gestionar_botones())
        {
            fraccion = ???;           // actualizar fracción según nuevos
                                         // valores de bpm i accent
        }
        swiWaitForVBlank();
        actualizar_pantallas();
        // activar_beat();           // ???
    } while (1);
    return(0);
}
```

El vector `ang_pos[360]` estará inicializado a los valores `(px, py)` adecuados para posicionar un *sprite* sobre la circunferencia grande, para los 360 grados sexagesimales (enteros). La variable `ang_actual` contendrá el ángulo actual de la circunferencia pequeña, expresado en coma fija con 12 bits para la parte fraccionaria (Q12). La variable `fraccion` será el incremento (Q12) que hay que sumar al ángulo actual a cada activación del *Vertical Blank*, para conseguir que la circunferencia pequeña pase por encima de los círculos de marca sonora a la frecuencia indicada en `bpm`. Hay que decidir si la rutina `activar_beat()` se llama desde el programa principal (descomentando la línea correspondiente) o desde la RSI.

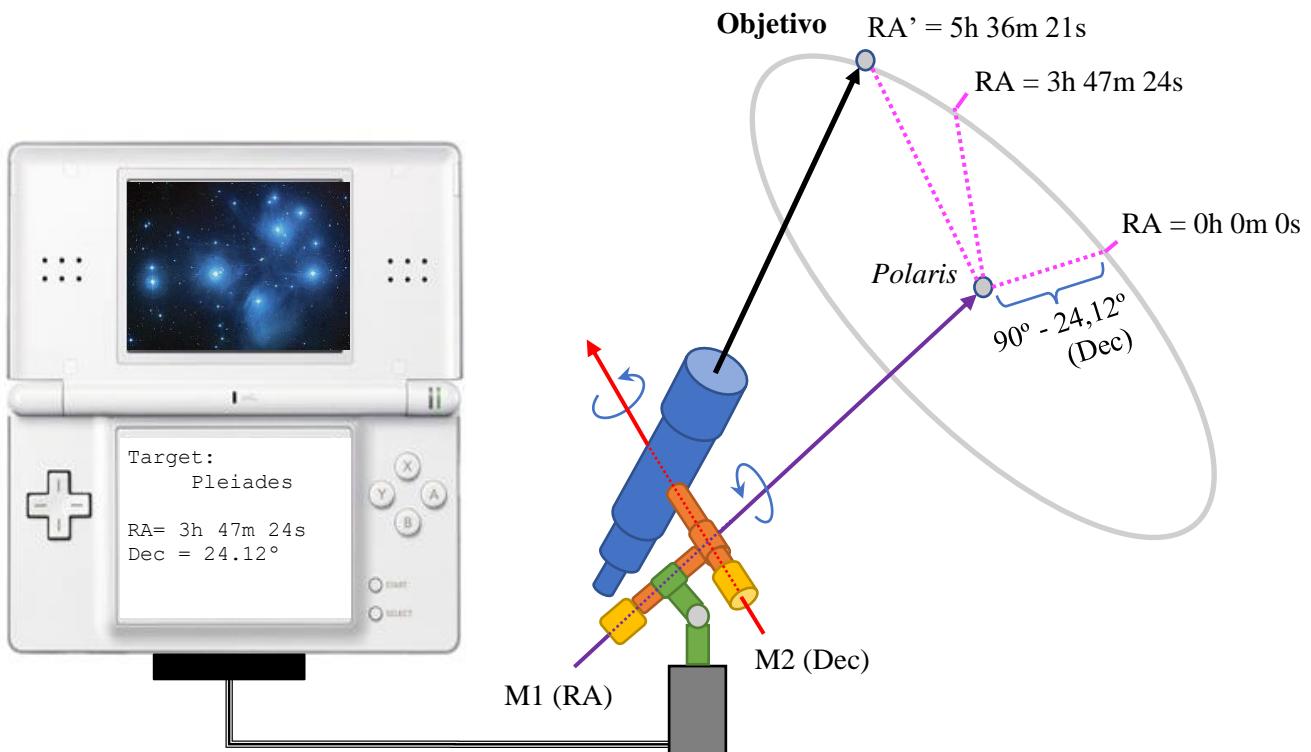
Se pide:

RSI de VBL en lenguaje ensamblador, expresión del cálculo de la fracción en lenguaje C, razonar si `activar_beat()` se debe llamar desde el programa principal o desde la RSI.

Problema 36 (1^a Conv. 2021-22): Control de telescopio

Se propone conectar a la NDS un sistema de motores para controlar automáticamente la orientación de un telescopio, utilizando una montura de tipo ecuatorial. Con este tipo de montura, cualquier objeto del firmamento se localiza a partir de dos coordenadas, RA (*Right Ascension*) y Dec (*Declination*). La coordenada Dec se codifica como un ángulo entre -90° y 90° sexagesimales, que indica la elevación respecto al ecuador de la Tierra, de modo que 90° corresponde al Polo Norte y -90° al Polo Sur. La coordenada RA se codifica en horas, minutos y segundos, tomando como inicio la dirección hacia el Sol cuando la Tierra está en el equinoccio de primavera, con un rango total de 24 horas para los 360° de circunferencia.

Las coordenadas RA y Dec son fijas para cada objeto celeste, pero como la Tierra está rotando, la orientación del telescopio hacia esa dirección se debe ir actualizando continuamente para compensar el giro aparente del firmamento. Esta es la principal ventaja de este tipo de montura, que debe estar correctamente orientada hacia el Polo Norte (o Sur) para que se pueda realizar el seguimiento de un objeto variando únicamente la coordenada RA. La coordenada Dec, sin embargo, no se debe modificar una vez localizado el objeto.



En el esquema anterior se muestra la montura correctamente orientada hacia la estrella Polar (*Polaris*), además de los dos motores que controlan la orientación: M1 (RA) y M2 (Dec). El objeto localizado en el ejemplo es el clúster de estrellas Pléyades, en la constelación de Tauro. Sus coordenadas son RA = 3h 47m 24s y Dec = 24,12°, pero en el momento de la observación el telescopio está apuntando a RA' = 5h 36m 21s, porque se supone que ha pasado alrededor de 1 hora y 49 minutos desde que se inició el seguimiento de este objeto. La separación respecto a la orientación Norte se expresa como 90° - 24,12° (Dec).

Se dispone de las siguientes rutinas ya implementadas:

Rutina	Descripción
inicializaciones()	Inicializa el <i>hardware</i> (pantalla, interrupciones, etc.)
activar_timer(int ntim, short div_freq)	Activa el funcionamiento e interrupciones del <i>timer</i> ntim, con frecuencia de entrada mínima (32728 Hz) y con el divisor de frecuencia div_freq
tareas_independientes()	Tareas que no dependen de la gestión de los motores del telescopio, por ejemplo, captura y procesamiento de las imágenes obtenidas con una cámara conectada al ocular del telescopio (tiempo de ejecución < 100 ms)
gestionar_interfaz(int *target_ra, short *target_dec)	Gestiona una interfaz de usuario que permite seleccionar un objeto celeste o unas coordenadas concretas, devolviendo un resultado (<i>unsigned char</i>) igual a 0 si no se ha seleccionado nada, igual a 1 si el usuario ha seleccionado un objeto a buscar, registrando sus coordenadas por referencia en target_ra y target_dec
circ_sub(int ra2, int ra1)	Realiza la resta de dos valores de RA (ra2 - ra1) expresados en segundos totales (86.400 segundos = 360°), teniendo en cuenta la circularidad de estos valores, es decir, obtiene el ángulo de separación mínimo entre los dos ángulos correspondientes a ra2 y ra1; devuelve la resta con signo por R0([-43.200..43.199]), y el valor absoluto de la resta por R1([0..43.200]), todo expresado en segundos
swiWaitForVBlank()	Espera hasta el próximo retroceso vertical
actualizar_pantallas()	Actualiza el contenido de las pantallas de la NDS, mostrando las variables globales de gestión del telescopio y las imágenes del objeto en observación

El programa a realizar debe permitir entrelazar la ejecución de las tareas independientes con la gestión de la interfaz de usuario y la búsqueda o seguimiento de un determinado objeto celeste.

Para realizar la búsqueda se deberán controlar los motores M1 y M2, mientras que para realizar el seguimiento solo se deberá controlar el motor M1. Dicho control se realizará enviando pulsos por unos determinados bits de un registro de Entrada/Salida a cierta frecuencia (ver descripción más adelante). Para que estos pulsos se puedan enviar concurrentemente con el resto de las tareas, se deberán generar desde las RSIs de los *timers* 1 y 2, respectivamente para cada motor. Además, para realizar el seguimiento se deberá ir reajustando la coordenada RA del objetivo continuamente, con la RSI del *timer* 0.

Se propone el siguiente programa principal:

```
#define FREQ_ENT 32728 // frecuencia de entrada mínima (Hz)

short divfreq_vmax = &a_?; // divisor freq. velocidad máx.
short divfreq_vmin = &b_?; // divisor freq. velocidad mín.

short dec_actual = 0, dec_objetivo = 0; // [-9000..9000] centígrados
int ra_actual = 0, ra_objetivo = 0; // [0..86399] segundos
unsigned char seek_ra = 0, seek_dec = 0; // búsqueda de objetivo
unsigned char track = 0; // seguimiento de objetivo

int main()
{
    inicializaciones();
    activar_timer(0, &c_?); // timer 0 siempre activo
    do
    {
        tareas_independientes();
        if (gestionar_interfaz(&ra_objetivo, &dec_objetivo))
        {
            activar_timer(1, divfreq_vmax); seek_ra = 1; track = 0;
            activar_timer(2, divfreq_vmax); seek_dec = 1;
        }
        swiWaitForVBlank();
        actualizar_pantallas();
    } while (1);
    return(0);
}
```

Básicamente, el programa utilizará dos variables globales `ra_objetivo` y `dec_objetivo` para determinar la posición del objeto a observar, junto con otras dos variables globales `ra_actual` y `dec_actual` para saber en qué dirección está apuntando el telescopio en cada instante. Las variables DEC almacenan centésimas de grado, mientras que las variables RA almacenan segundos totales, es decir, toda la circunferencia se puede codificar con valores entre 0 y 86.399 segundos ($24\text{ h} \cdot 60\text{ m/h} \cdot 60\text{ s/m} - 1\text{s}$).

En el caso de `ra_actual`, podrá tener valores negativos, por ejemplo, -43.200 segundos equivaldrían a -180° , o positivos iguales o superiores a 86.400, por ejemplo, 129.600 segundos equivaldrían a 540° ($360+180^\circ$). Esto puede pasar cuando se incremente o decremente esta variable para aproximarse a la posición objetivo (`ra_objetivo`). Esto no supondrá ningún problema a la hora de comparar dichas variables, si se utiliza la rutina `circ_sub()`, que tiene en cuenta la propiedad de circularidad de estas variables.

Además, otras tres variables globales booleanas determinan si se está moviendo el telescopio para fijar las coordenadas RA y DEC (`seek_ra` y `seek_dec`), o si se está moviendo el telescopio para compensar la rotación de la Tierra (`track`).

Como se puede observar, se han dejado unas marcas `&a_?`, `&b_?` y `&c_?` para indicar diversos divisores de frecuencia, que se piden como parte de la solución.

Por otro lado, se supone que las RSIs de los *timers* 0 y 2 ya están implementadas, con lo cual, solo hay que implementar la RSI del *timer* 1.

La RSI del *timer* 1 debe controlar los bits 5, 6 y 7 del registro REG_TEL, que tienen las siguientes funcionalidades:

Control	M2:	M1:	
	bit 1	bit 5	0 -> giro grueso, 1 -> giro fino
	bit 2	bit 6	0 -> incrementar, 1 -> decrementar
	bit 3	bit 7	pulso -> una variación de giro

Los motores pueden avanzar en dos resoluciones, según la selección del tipo de giro:

giro grueso:	1 pulso = 0,5°	(120 segundos de RA)
giro fino:	1 pulso = 0,00417°	(1 segundo de RA, aprox.)

La frecuencia máxima de pulsos que aceptan los motores, independientemente del tipo de giro, es de 15 pulsos por segundo. Cada pulso debe ser simétrico, es decir, el mismo tiempo a 1 que a 0. A la velocidad (frecuencia) máxima y con giro grueso, el motor M1 puede dar una vuelta completa en 48 segundos. Además, el motor M1 puede compensar la rotación de la Tierra si se configura en giro fino, con un pulso de incremento por segundo.

En general, la RSI del *timer* 1 se debe encargar de las siguientes tareas:

- generar pulsos por el bit 7, a la frecuencia adecuada,
- fijar el sentido de giro (incremento o decremento), según los valores de `ra_objetivo` y `ra_actual`,
- en modo de búsqueda, si la diferencia entre `ra_objetivo` y `ra_actual` es superior o igual a 120 segundos, usar el giro grueso a la velocidad máxima,
- en modo de búsqueda, si la diferencia entre `ra_objetivo` y `ra_actual` es inferior a 120 segundos y superior a 0 segundos, usar el giro fino a la velocidad máxima,
- en modo de búsqueda, si `ra_objetivo` = `ra_actual`, pasar a modo seguimiento, reconfigurando el *timer* 1 para girar el motor M1 a la velocidad mínima (1 pulso/seg),
- en modo de seguimiento, usar el giro fino a la velocidad mínima,
- actualizar el valor de `ra_actual`, teniendo en cuenta el tipo de giro (grueso o fino) y el sentido de giro (incremento o decremento).

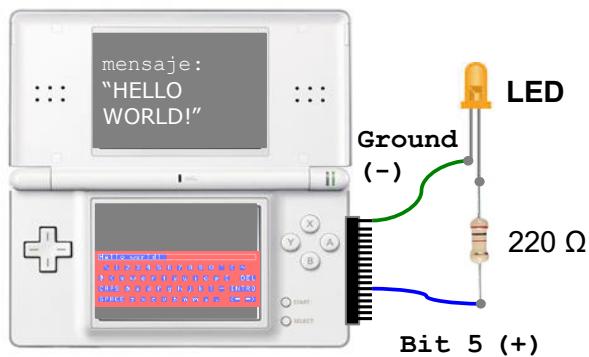
Por su parte, la RSI del *timer* 0 irá incrementando el valor de `ra_objetivo` continuamente, teniendo en cuenta la propiedad de circularidad (cuando llegue a 86.400 pasará a 0).

Se pide:

RSI del *timer* 1 en ensamblador, expresión del cálculo de los divisores de frecuencia `_a_`, `_b_` y `_c_` en lenguaje C.

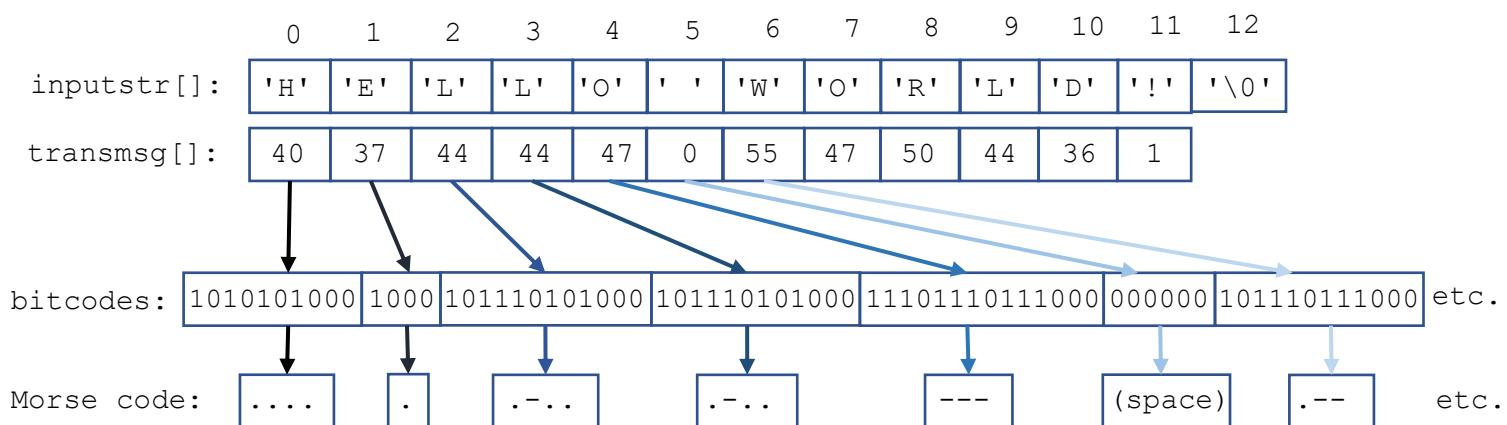
Problema 37 (2^a Conv. 2021-22): Escritura Morse

Se propone realizar un programa para la NDS que permita visualizar un mensaje en código Morse utilizando un diodo led conectado al bit 5 de un registro de entrada/salida de nombre simbólico REG_DATA (16 bits). A continuación se muestra un esquema del *hardware*:



El programa incluirá una función (ya implementada) para leer el mensaje que el usuario quiera reproducir en Morse. Esta función mostrará un teclado virtual sobre la pantalla táctil de la NDS, más un buffer (de una línea) que mostrará las letras que se van introduciendo. Cuando el usuario ya tenga el mensaje listo, pulsará la tecla virtual RETURN y el mensaje se preparará para ser retransmitido como señales luminosas cortas (puntos) y largas (rallas) a través del led. Mientras se esté retransmitiendo dichas señales, el usuario podrá ir introduciendo el mensaje siguiente con el teclado virtual.

Para reproducir el código Morse correspondiente, los códigos ASCII del mensaje, contenidos en un *string* de nombre `inputstr[]`, se transformarán en un vector de códigos numéricos, de nombre `transmsg[]`. Cada código numérico sirve para indexar otro vector, que se explicará más adelante, que contiene un *bitcode* para cada letra. Este *bitcode* consiste en patrones de bits que representan un punto con 1 uno y una raya con 3 unos seguidos, más 1 cero de separación entre puntos y rayas, 3 ceros de separación entre letras y 6 ceros de separación entre palabras. El siguiente esquema muestra un ejemplo de generación de una secuencia de *bitcodes* para el mensaje "HELLO WORLD!":

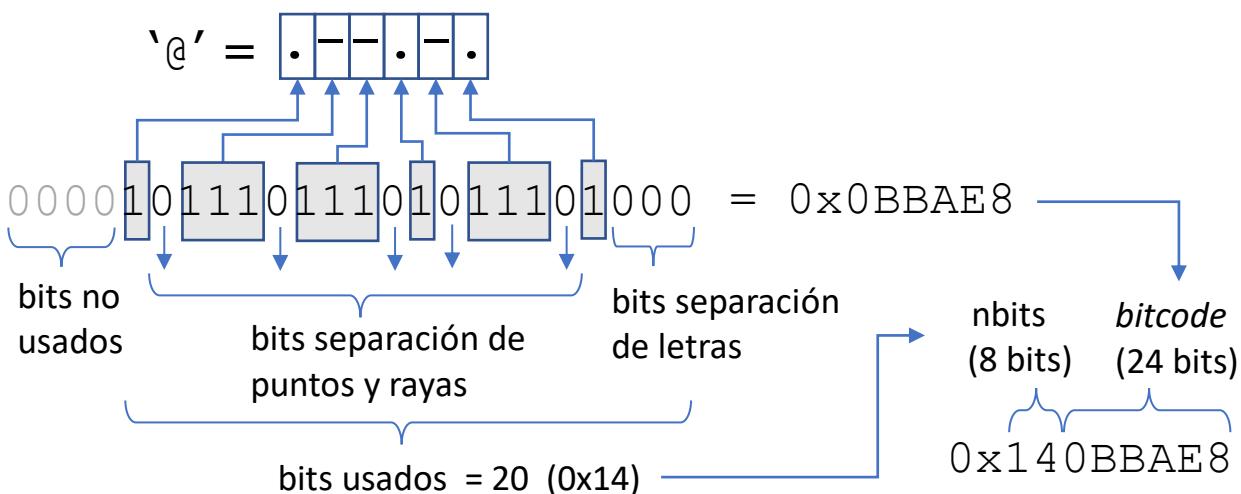


El vector `nbcodes[]` contiene los *bitcode* correspondientes a cada letra. Este vector ya está

inicializado con 59 *words*. A continuación se muestra una tabla que representa el contenido de algunas posiciones de dicho vector. Para cada posición se muestra el índice (*ind*), la letra correspondiente (*char*), el código de puntos y rayas (*Morse*), el número de bits útiles del *bitcode* (*nbits*), el código binario (*bitcode*) y el valor específico de 32 bits que está almacenado finalmente en el vector (*nbcodes*[*ind*]), expresado en hexadecimal:

ind	char	Morse	nbits	bitcode	nbcodes [<i>ind</i>]
0	' '	(space)	6	000000000000000000000000000000	0x06000000
1	'!'	-.-.--	22	001110101110101110111000	0x163AEBB8
2	'''	.-...-	18	000000101110101011101000	0x1202EAE8
:					
16	'0'	-----	22	001110111011101110111000	0x163BBBB8
17	'1'	.----	20	000010111011101110111000	0x140BBBB8
18	'2'	. .---	18	000000101011101110111000	0x1202BBB8
19	'3'	. . .--	16	000000001010101110111000	0x1000ABB8
:					
32	'@'	.---.-	20	000010111011101011101000	0x140BBAE8
33	'A'	.-	8	000000000000000010111000	0x080000B8
34	'B'	-....	12	000000000000111010101000	0x0C000EA8
35	'C'	-..-.	14	00000000000011101011101000	0x0E003AE8
36	'D'	-..	10	000000000000111010101000	0x0A0003A8
37	'E'	.	4	0000000000000000000000001000	0x04000008
:					
56	'X'	-...-	14	00000000000011101010111000	0x0E003AB8
57	'Y'	-.--	16	0000000000001110101110111000	0x1000EBB8
58	'Z'	--..	14	00000000000011101110101000	0x0E003BA8

Los *bitcode* tienen longitud variable, oscilando entre 4 y 22 bits, de modo que es necesario registrar la longitud de cada *bitcode* (*nbits*). Cada *word* almacenado en *nbcodes*[] empaqueta dicha longitud en los 8 bits altos y el propio *bitcode* en los 24 bits bajos. El siguiente esquema muestra un ejemplo de codificación para la letra '@':



Se dispone de las siguientes rutinas ya implementadas:

Rutina	Descripción
inicializaciones ()	Inicializa el <i>hardware</i> (pantalla, interrupciones, etc.)
activar_timer(int ntim, int nfreqin, short divfreq)	Activa el funcionamiento e interrupciones del <i>timer</i> ntim, con el identificador de la frecuencia de entrada nfreqin (0, 1, 2 o 3) y con el divisor de frecuencia divfreq
desactivar_timer(int ntim)	Desactiva el funcionamiento del <i>timer</i> ntim
leer_mensaje(char *str, int maxlen)	Gestiona la interfaz que permite al usuario introducir un <i>string</i> mediante un teclado virtual sobre la pantalla táctil; el <i>string</i> se devuelve por referencia sobre el parámetro str, limitado a la longitud indicada en maxlen (centinela excluido); la rutina no retorna hasta que el usuario valida el <i>string</i> pulsando la tecla virtual RETURN
transformar_mensaje(char *str, unsigned char vect[])	Realiza la conversión de códigos ASCII del <i>string</i> de entrada str en índices para el vector nancode[], que se guardan sobre el vector vect[]; devuelve el número de letras transformadas, excluyendo el centinela del <i>string</i> (el resultado es de tipo unsigned char)
swiWaitForVBlank()	Espera hasta el próximo retroceso vertical

El programa a realizar debe permitir entrelazar la ejecución de la rutina leer_mensaje() con la retransmisión del mensaje anterior. Para conseguir esta concurrencia de tareas, la retransmisión de los *bitcode* se realizará mediante la RSI del *timer* 0. Hay que tener en cuenta que el usuario podría introducir un nuevo mensaje antes de que la RSI haya terminado de retransmitir el mensaje anterior. En este caso, será necesario esperar al final de dicha retransmisión.

Se propone el siguiente programa principal:

```
#define FREQ_ENT0 33513982 // posibles frecuencias de entrada
#define FREQ_ENT1 523656 // de un timer (en Hz)
#define FREQ_ENT2 130914
#define FREQ_ENT3 32728
#define MAX_LON 50 // longitud máxima del mensaje

unsigned int nancode[59]; // vector códigos (ya inicializado)
char inputstr[MAX_LON+1]; // mensaje introducido por usuario
unsigned char transmsg[MAX_LON]; // mensaje transformado
unsigned char lontr = 0; // longitud mensaje transformado
unsigned char curr_ind = 0, // índice de letra actual
                          curr_bit; // número de bit actual
int main()
{
```

```

inicializaciones();
do
{
    leer_mensaje(inputstr, MAX_LON);
    while (curr_ind < lontr) // esperar el final de la reproducción
    {                         // del mensaje anterior (si la hay)
        swiWaitForVBlank();
    }
    lontr = transformar_mensaje(inputstr, transmsg);
    if (lontr > 0)
    {
        curr_ind = 0;
        curr_bit = nbcode[transmsg[0]] >> 24;
        activar_timer(0, ¿_a_?, ¿_b_?);
    }
} while (1);
return(0);
}

```

Después de leer mensaje, espera el final de una posible retransmisión anterior, transforma el mensaje y comprueba que haya alguna letra a retransmitir. En caso afirmativo, el cuerpo del `if` inicializa las variables globales `curr_ind` y `curr_bit` para establecer el índice de la letra a retransmitir (initialmente cero) y el número de bit actual, inicializado con el número de bits útiles del *bitcode* de la primera letra del mensaje, que se extrae de los 8 bits de más peso del valor almacenado en `nbcode[transmsg[0]]`.

A continuación, se activa el *timer* 0. Se han dejado unas marcas `¿_a_?` y `¿_b_?` para indicar el número de frecuencia de entrada (0, 1, 2 o 3) y el cálculo del divisor de frecuencia adecuado, que se piden como parte de la solución. Teniendo en cuenta que la RSI deberá transmitir un bit del *bitcode* a cada interrupción, hay que determinar los valores de `¿_a_?` y `¿_b_?` para que la retransmisión de la letra 'S', cuyo código Morse es '...' (3 puntos), junto con el espacio entre letras, se realice en un segundo.

En general, la RSI del *timer* 0 se debe encargar de las siguientes tareas:

- decrementar `curr_bit`,
- obtener el *bitcode* de la letra que se está retransmitiendo actualmente (según `curr_ind`),
- obtener el valor del bit actual del *bitcode* que se está retransmitiendo actualmente, y fijar el bit 5 de `REG_DATA` a ese valor (sin modificar el resto de bits de `REG_DATA`),
- si `curr_bit` ha llegado a 0 , pasar a la siguiente letra y reiniciar `curr_bit`,
- si ya se ha llegado al final del mensaje, detener el *timer* 0.

Se pide:

RSI del *timer* 0 en ensamblador, expresión de los parámetros `_a_` y `_b_` en lenguaje C.

Problema 38 (1^a Conv. 2022-23): Teléfono vintage

Se propone controlar un teléfono antiguo (analógico) con la plataforma NDS. Concretamente, en este problema se pide detectar la introducción de los dígitos del número de teléfono que se desea marcar. Esta introducción (o marcación) se realiza mediante un dial rotatorio, como el mostrado en la parte central del teléfono de la siguiente figura:

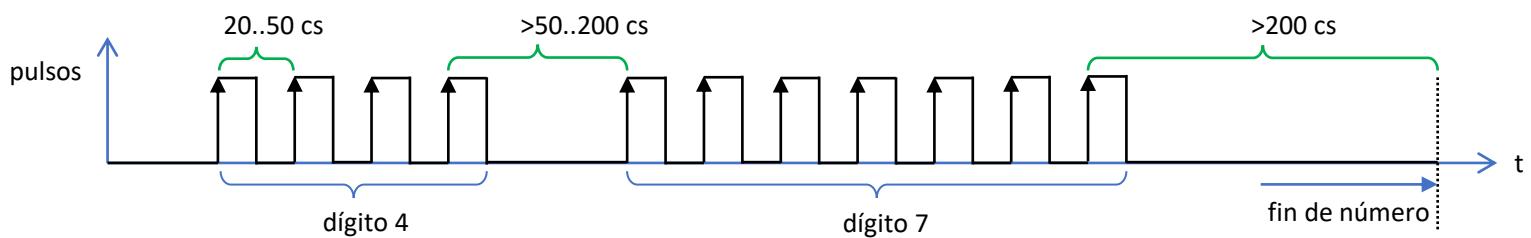


El dial funciona de la siguiente manera:

- el usuario introduce la punta de un dedo en el agujero correspondiente al dígito que desea marcar,
- luego gira el dial en sentido horario, hasta llegar a la cuña metálica que sirve de tope,
- después saca el dedo,
- entonces el dial gira automáticamente en sentido antihorario (gracias a un muelle), hasta que llega a su posición de inicio, preparado para marcar el siguiente dígito.

Para conseguir detectar la selección de los dígitos con la NDS, se ha acoplado una circuitería electrónica específica que envía pulsos hacia un pin del puerto de conexión GBA, cuando el dial está girando automáticamente. Este sistema permite generar un número de pulsos igual al dígito marcado, excepto para el dígito 0, que genera diez pulsos.

El siguiente cronograma muestra un ejemplo de introducción de dos dígitos, el 4 y el 7, seguidos de un tiempo de espera de 2 segundos:



En el cronograma anterior se han indicado los rangos de tiempo para distinguir entre pulsos de un mismo dígito y de dígitos diferentes, así como el final del número. Concretamente, el tiempo entre pulso y pulso de un mismo dígito oscilará entre 20 y 50 cs (centésimas de segundo), dependiendo de las características físicas del sistema que hace retroceder el dial. Cuando el tiempo entre el pulso actual y el anterior sea superior a 50 cs, se interpretará que el pulso actual es el primero del nuevo dígito. Si pasan 2 segundos sin introducir nuevos pulsos, se interpretará que el usuario ha terminado de introducir dígitos.

Se dispone de las siguientes rutinas ya implementadas:

<i>Rutina</i>	<i>Descripción</i>
inicializaciones ()	Inicializa el <i>hardware</i> (pantalla, interrupciones, etc.)
tareas_independientes ()	Tareas que no dependen de la detección de los pulsos de un nuevo número de teléfono, por ejemplo, gestión de una agenda telefónica (tiempo de ejecución < 1s)
swiWaitForVBlank()	Espera hasta el próximo retroceso vertical
printf(char *format,...)	Escribe un mensaje en la pantalla inferior de la NDS
realizar_llamada(unsigned char numtel[], unsigned char ndig)	Se comunica con la central telefónica para efectuar la marcación de los dígitos almacenados en el vector que se pasa por referencia, con el número de dígitos pasado por valor.

Para detectar los pulsos se usará la IRQ_CART (del cartucho GBA), donde está conectado el cable que envía los pulsos a la NDS, de modo que cada pulso generará una activación de la correspondiente RSI. Esta RSI, que denominaremos `rsi_sensor()`, se debe encargar de contar los pulsos de cada dígito, pero también debe controlar el tiempo entre pulsos.

Para realizar dicho control del tiempo se utilizará el *timer 0*, configurado con la frecuencia de entrada mínima ($\approx 32.728,498$ Hz) y con el divisor de frecuencia máximo, de modo que generará una interrupción cada 2 segundos aproximadamente.

Por lo tanto, la RSI del sensor NO podrá utilizar las interrupciones del *timer 0*, sino que deberá leer el registro de datos de dicho *timer*. De este modo, podrá obtener el valor actual del contador de tics (de entrada), el cual se carga con el divisor de frecuencia cuando se inicia o reinicia el *timer*, y se va incrementando hasta llegar a cero. De este modo, si la diferencia del número de tics actual respecto al valor del divisor de frecuencia es superior a 16.364 tics, significará que han pasado un poco más de 50 cs desde el último (re)inicio del *timer*.

Las interrupciones del *timer 0* se usarán para detectar si han pasado 2 segundos desde la última vez que se (re)inició el *timer*. De este modo, la RSI del sensor debe detectar cuándo se cambia de dígito, mientras que la RSI del *timer 0* debe detectar cuándo el usuario ha dejado de marcar dígitos. En ambos casos se debe almacenar el valor numérico del dígito anterior en un vector

global denominado `NTel[]`. Como esta tarea se debe realizar en ambas RSIs, se aconseja implementarla en la siguiente rutina auxiliar:

```
unsigned int capture_digit(unsigned int n_puls);
```

Esta rutina recibe como único parámetro el número de pulsos del dígito marcado y siempre devuelve cero como resultado.

Se propone el siguiente programa principal:

```
#define MAXDIGITS 14
unsigned char NTel[MAXDIGITS]; // vector para almacenar número tel.
unsigned char ind_digit = 0; // índice del dígito actual
unsigned char new_digit = 0; // si vale 1 es que hay nuevos dígitos
unsigned char num_pulses = 0; // número de pulsos actual

void main(void)
{
    unsigned char ind_digit_ant = 0; // índice del dígito anterior
    inicializaciones();
    TIMER0_DATA = i_a_?; // fijar divisor de frecuencia máximo
    do
    {
        tareas_independientes();
        if (new_digit) // si se han marcado nuevos dígitos
        {
            swiWaitForVBlank();
            if (¿_b_?) // si se trata del primer dígito
            {
                printf("Número tel.: ");
            }
            while (ind_digit_ant < ind_digit) // visualizar dígitos pendientes
            {
                printf(¿_c_?);
                ind_digit_ant++;
            }
            if (num_pulses == 0) // si se ha terminado la marcación
            { // de dígitos
                printf("\n");
                realizar_llamada(NTel, ind_digit);
                ind_digit_ant = 0;
                ind_digit = 0; // reiniciar proceso de marcación
            }
            new_digit = 0;
        }
    } while (1);
}
```

El programa principal entrelaza la realización de tareas independientes con la visualización por pantalla de los nuevos dígitos marcados por el usuario. Hay que prever que se pueden haber introducido más de un nuevo dígito mientras se estaban realizando las tareas

independientes. Cuando se detecten nuevos dígitos, pero la variable `num_pulses` sea igual a cero, significará que la marcación del número ha terminado y que se puede realizar la llamada telefónica, invocando a la rutina `realizar_llamada()`.

Para simplificar la complejidad del problema, vamos a suponer que la rutina `realizar_llamada()` no finaliza hasta que el usuario cuelga el teléfono, y que el usuario nunca marcará números telefónicos de más de 14 dígitos.

Además, se pide que a cada detección de pulso se active una de las diez luces LED que se han instalado en el teléfono, al lado de cada dígito. Concretamente, se debe acceder al registro `REG_TEL` (16 bits):

REG_TEL:	10 9 8 7 6 5 4 3 2 1 0															
	x	x	x	x	x	0	0	0	0	0	1	1	1	1	x	
	L0	L9	L8	L7	L6	L5	L4	L3	L2	L1						

En el esquema anterior se supone que se llevan detectados cuatro pulsos: hay que ir activando los LEDs a medida que se detecta cada nuevo pulso, pero hay que dejar activos los LEDs anteriores, hasta que se detecte un nuevo dígito o se acabe la marcación; entonces hay que apagarlos todos.

Por último, a modo de recordatorio se proporcionan los siguientes datos respecto a la programación del *timer* 0:

- `TIMER0_DATA` (0x04000100): registro de datos (16 bits) del *timer* 0:
 - se escribe para fijar el valor del divisor de frecuencia,
 - se lee para obtener el valor actual del contador de tics, que irá incrementando desde el valor del divisor de frecuencia hasta cero, momento en el que se disparará la IRQ del *timer* (si las interrupciones están habilitadas) y se reiniciará el contador de tics.
- `TIMER0_CR` (0x04000102): registro de control (16 bits) del *timer* 0; dispone de los siguientes campos (el resto de bits no se utilizan):
 - bits 1..0: indican la selección de la frecuencia de entrada

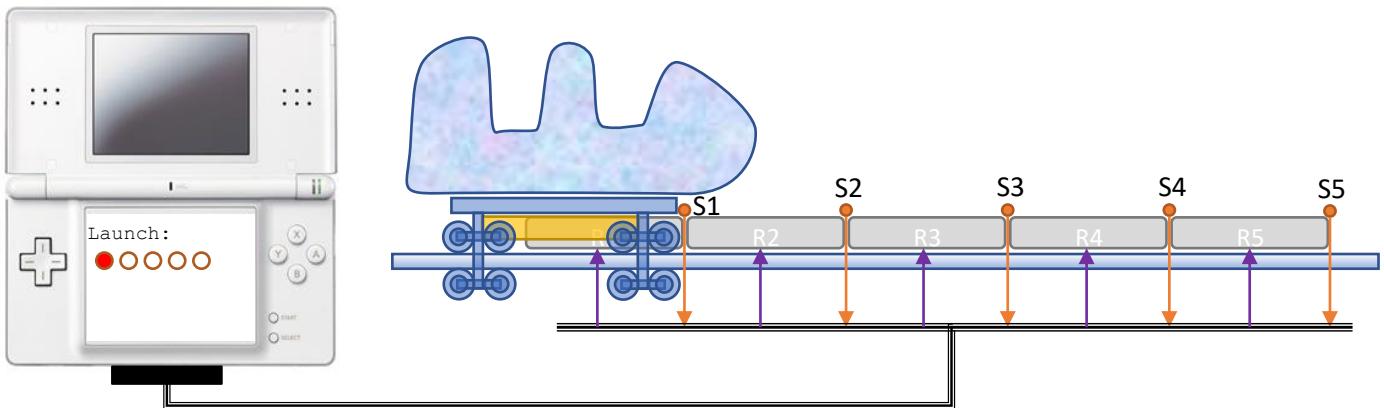
00 → F/1 (=33.513.982 Hz)	01 → F/64 (\approx 523.656 Hz)
10 → F/256 (\approx 130.914 Hz)	11 → F/1024 (\approx 32.728,5 Hz)
 - bit 2: encadena el *timer* con el anterior (la salida del *timer* anterior es la entrada del *timer* actual),
 - bit 6: permite habilitar la generación de interrupciones,
 - bit 7: 0 → *timer* parado, 1 → *timer* en marcha; la secuencia "poner a cero, poner a uno" reinicia el contador de tics.

Se pide:

RSI del sensor y rutina `capture_digit()` en lenguaje ensamblador, partes del programa principal marcadas con `i_x_?` en lenguaje C. No es necesario implementar la RSI del *timer*.

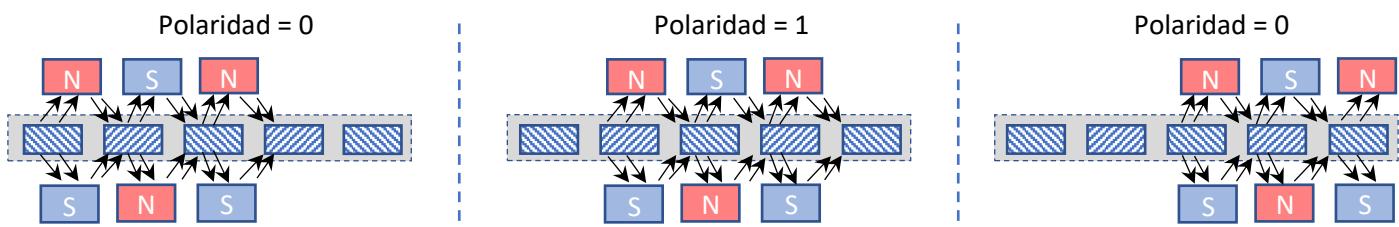
Problema 39 (1^a Conv. 2022-23): Lanzamiento con electroimanes

Se propone controlar con la NDS el lanzamiento de un coche de montaña rusa propulsado por un motor electromagnético lineal. La siguiente figura muestra un esquema del sistema:



En el esquema anterior se muestra una serie de cinco bloques etiquetados del R1 al R5. Cada uno de estos bloques contiene seis electroimanes. Estos bloques están situados en medio del raíl y son estrechos, de modo que forman una especie de mampara. El esquema muestra la mampara de lado (alto x ancho); con una vista superior del raíl, observaríamos una línea.

Los electroimanes son bobinas (rodillos) de cable conductor que generan un campo magnético cuando circula corriente eléctrica. Cambiando el sentido de circulación de la corriente se consigue cambiar la polaridad magnética (Norte/Sur). A su vez, el coche de la montaña rusa lleva instalados doce imanes permanentes, seis a cada lado de la mampara de bloques.



Para ilustrar cómo actúa el motor, el esquema anterior muestra una vista superior de una versión reducida de los componentes descritos inicialmente: un bloque de cinco electroimanes (rectángulos con un patrón de rayas oblicuas), y seis imanes laterales (tres + tres) que van unidos a la parte inferior del coche. Con la circulación de corriente por todos los electroimanes del bloque se producen unas fuerzas de atracción y repulsión (representadas con flechas) con los imanes permanentes que hacen que el coche avance. Cuando el coche ha avanzado hasta la siguiente posición de electroimanes, se debe cambiar la polaridad para seguir moviendo el coche en el mismo sentido de la marcha. La polaridad se ha indicado como 0 y 1, que en realidad se corresponde con los dos sentidos de circulación de la corriente eléctrica. Evidentemente, el cambio de polaridad debe ir sincronizado con el movimiento del coche.

Además, para conseguir resincronizar el cambio de polaridad con el movimiento del coche, se ha instalado un sensor al final de cada bloque de electroimanes. Cada sensor está etiquetado desde S1 hasta S5. El control de los bloques de electroimanes y la detección del estado de los sensores se realizará mediante dos registros de 16 bits:

REG_EM:	9	8	7	6	5	4	3	2	1	0
	x	x	x	x	x	0	0	0	1	0

R5 R4 R3 R2 R1

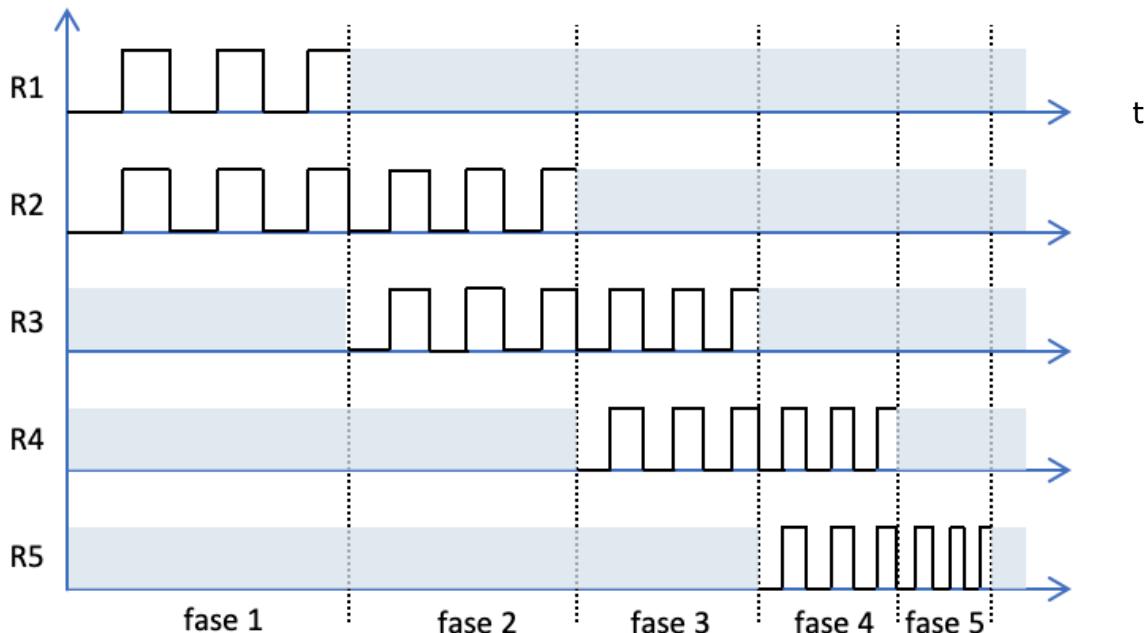
REG_SENS:	5	4	3	2	1	0
	x	x	x	x	x	0

S5 S4 S3 S2 S1

Los bits del 1 al 5 del registro REG_SENS indican si el coche está pasando por encima del sensor correspondiente (1) o no (0). El resto de bits no se utilizan.

Los bits del 0 al 9 del registro REG_EM permiten controlar la activación de cada bloque de electroimanes y su polaridad. Concretamente, para cada bloque se reservan un par de bits, tal como se indica en el diagrama del registro. El bit de más peso del par sirve para activar (1) o desactivar (0) la transmisión de corriente eléctrica a través de los electroimanes del bloque, mientras que el bit de menos peso del par sirve para modificar el sentido de la circulación de la corriente. El resto de bits no se utilizan.

Cada bloque de electroimanes consume una cantidad muy elevada de potencia eléctrica (centenares de kilovatios). Por lo tanto, se requiere que no se activen todos los bloques a la vez, sino solo dos bloques como máximo, cuando el coche esté entre esos dos bloques. El siguiente cronograma representa la evolución de la activación y cambio de polaridad de los cinco bloques de electroimanes, durante cinco fases.



Cuando el bloque está desactivado se muestra una banda sombreada. Cuando el bloque está activado, se muestran los pulsos que representan la polaridad de los electroimanes. A medida que el coche avanza, se incrementa la frecuencia de los pulsos para acelerarlo.

Se dispone de las siguientes rutinas ya implementadas:

Rutina	Descripción
inicializaciones()	Inicializa el <i>hardware</i> (pantalla, interrupciones, etc.)
tareas_independientes()	Tareas que no dependen del lanzamiento del coche, por ejemplo, control del acceso de pasajeros, preparación de los sistemas de frenado, iluminación, etc. (tiempo de ejecución < 0,1 s)
scanKeys() / keysDown()	Funciones de lectura de botones de la librería <i>libNDS</i>
swiWaitForVBlank()	Espera hasta el próximo retroceso vertical
show_sensor(unsigned id_sens, unsigned status)	Visualiza por pantalla el estado del sensor indicado por parámetro (<i>id_sens</i> : [1..5]), mostrando un círculo lleno o vacío según si el segundo parámetro es cierto o falso
prog_timer(unsigned id_tim, short div_freq);	Programa el <i>timer</i> indicado con el primer parámetro (<i>id_tim</i> : [0..3]), con las interrupciones habilitadas y el divisor de frecuencia pasado como segundo parámetro, usando la frecuencia de entrada mínima. Si se pasa un cero como divisor de frecuencia, detiene el <i>timer</i> .

El sistema electrónico de control generará una petición de interrupción (IRQ_CART) cada vez que se active un nuevo sensor. A esta petición se vinculará una RSI que denominaremos *rsi_sensor()*. Esta RSI se debe encargar del control de la fase actual del lanzamiento, modificando convenientemente los bits de activación de los bloques de electroimanes.

Para controlar el cambio de polaridad de los electroimanes se utilizará la RSI del *timer* 0: cada vez que se active la *rsi_timer0()*, se debe cambiar el estado de los bits de polaridad de al menos los bloques que estén activos en ese momento. Sin embargo, para simplificar el código se pueden cambiar los bits de polaridad de todos los bloques a la vez, ya que estos cambios solo afectan a los electroimanes de los bloques activos.

Para determinar el divisor de frecuencia adecuado a cada cambio de polaridad se dispone de un vector global denominado *DivFreq[]*. Este vector contiene 30 valores negativos, que están precalculados para obtener la velocidad óptima de todos los cambios de polaridad. En un sistema real, los tiempos entre cambios de polaridad se deben ir reajustando dinámicamente según evoluciona el coche, puesto que este movimiento depende de muchos factores físicos (peso de los pasajeros, rozamiento, viento, vibraciones, etc.), pero para simplificar el problema vamos a suponer que este reajuste no será necesario.

Por su parte, el programa principal deberá atender a múltiples tareas de control de la montaña rusa. La tarea de preparación del lanzamiento se supone que ya está implementada dentro de las tareas independientes. Esta tarea es compleja, puesto que hay que asegurar que el coche está en la posición inicial correcta, es decir, justo encima del primer bloque de electroimanes, con velocidad inicial cero. Para simplificar el problema, vamos a suponer que hay una variable

global denominada `launch_ready`, que será cierta si ya se puede realizar el lanzamiento, en cuyo caso el operario podrá pulsar el botón START. Mientras se realiza el lanzamiento, hay que ir mostrando por pantalla el estado de los sensores. Se propone el siguiente código:

```
#define MAXDIVFREQ 30
const short DivFreq[MAXDIVFREQ]; // vector con los divisores de frecuencia
unsigned char ind_polx = 0; // indice del cambio de polaridad actual
unsigned char phase = 0; // fase actual del lanzamiento
unsigned char launch_ready = 0; // indica si el lanzamiento está preparado

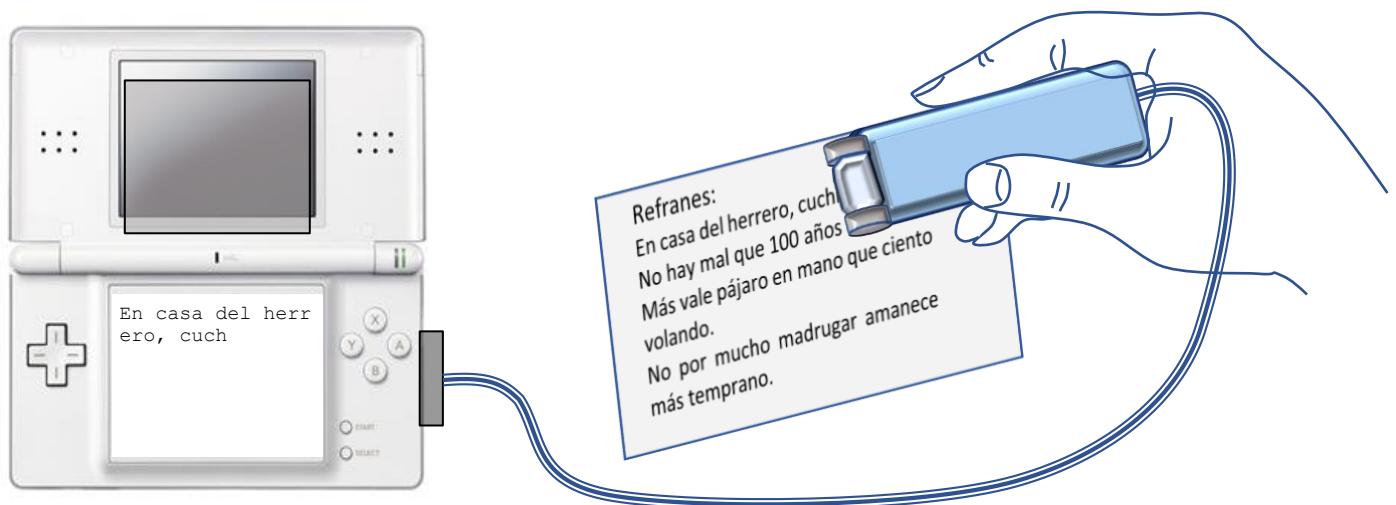
void main(void)
{
    unsigned char i;
    inicializaciones();
    do
    {
        tareas_independientes();
        if (launch_ready) // si está preparado para lanzamiento
        {
            scanKeys();
            if (keysDown() & KEY_START) // detectar pulsación START
            {
                phase = 1; ind_polx = 0;
                REG_EM = &a_?; // primera activación electroimanes
                prog_timer(0, DivFreq[0]);
                launch_ready = 0;
            }
        }
        else if (&b_?) // si está realizando el lanzamiento
        {
            swiWaitForVBlank();
            for (i = 5; i > 0; i--)
            {
                show_sensor(i, &c_?);
            }
            if (ind_polx == MAXDIVFREQ) // si ya se han realizado todos
                // los cambios de polaridad,
                phase = 0; // volver al estado inicial
                REG_EM = 0; // desactivar el último bloque de em
            }
        }
    } while (1);
}
```

Se pide:

`rsi_sensor()` y `rsi_timer0()` en lenguaje ensamblador, partes del programa principal marcadas con `&x_?` en lenguaje C.

Problema 40 (2^a Conv. 2022-23): Escáner de líneas de texto

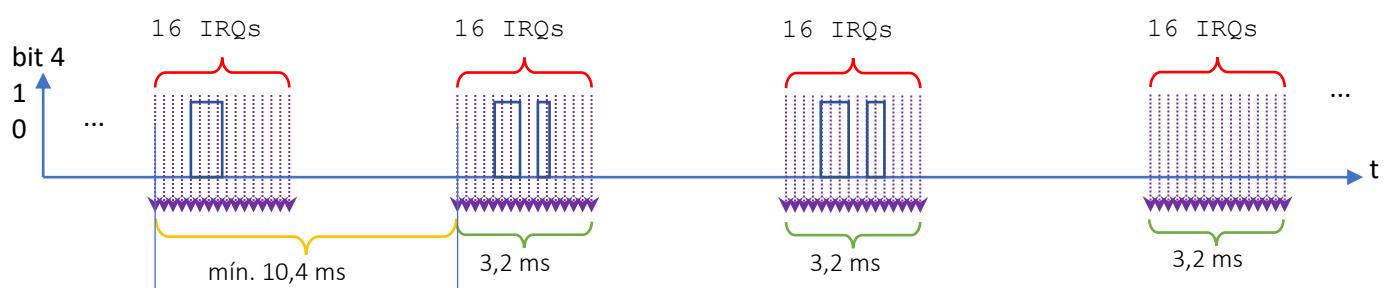
Se propone controlar con la NDS un escáner de líneas de texto impreso. La imagen capturada se puede procesar para reconocer los caracteres impresos: a este proceso se le denomina OCR (*Optical Character Recognition*). La siguiente figura muestra un esquema del sistema:



El escáner se sujetá con la mano y se hace deslizar por encima de la línea de texto que se quiere capturar, de izquierda a derecha. El cabezal del escáner dispone de un sensor de 16 píxeles, donde cada píxel se codifica con un único bit: 0 → blanco, 1 → negro. Vamos a denominar 'columna' a los 16 píxeles verticales capturados en cada posición.

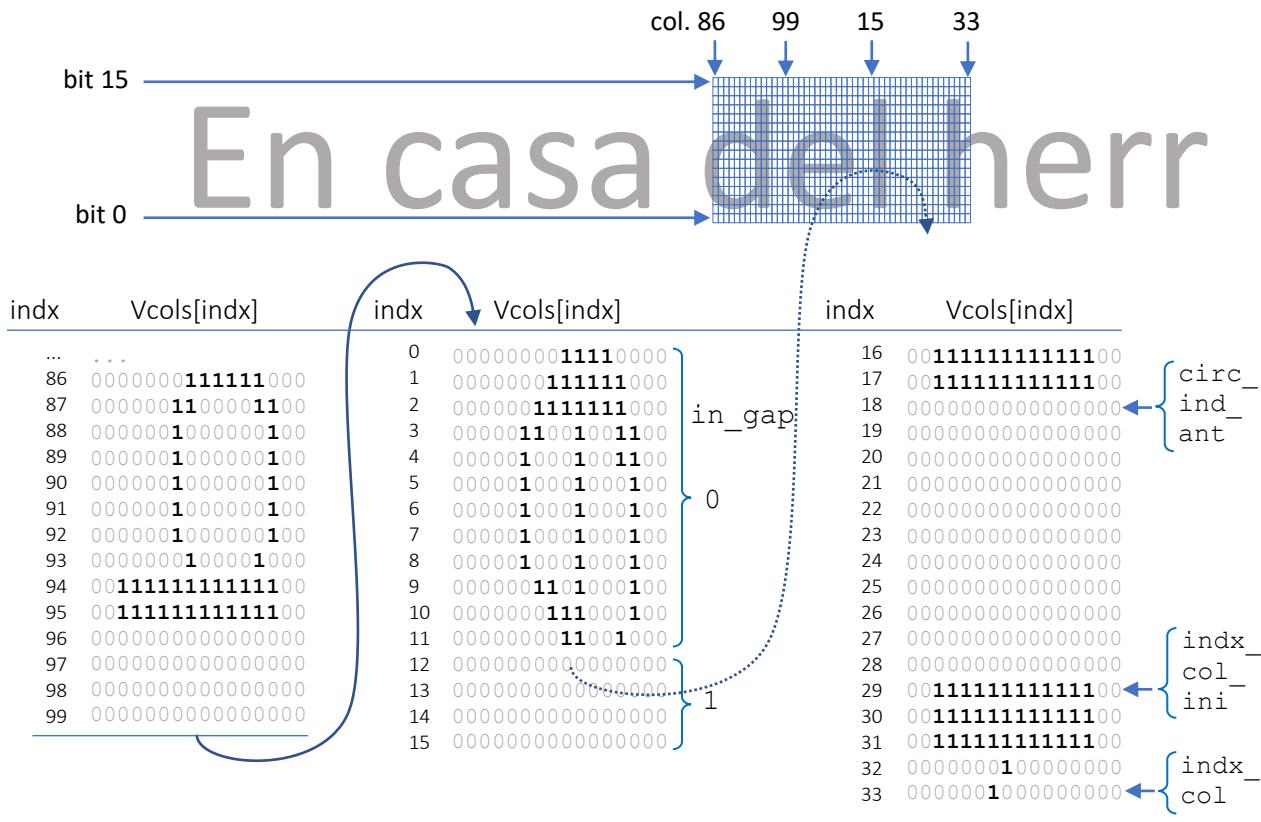
El cabezal dispone también de unas ruedas que permiten detectar el deslizamiento del sensor por encima del papel. El escáner captura 24 columnas por centímetro. Para cada columna capturada, el sensor enviará los bits de los 16 píxeles en serie (uno detrás de otro, empezando por el píxel superior) por un cable conectado al bit 4 de un registro de Entrada/Salida denominado REG_IO, pero el resto de los bits pueden tener otros usos.

El escáner utiliza otro cable para generar una petición de interrupción (IRQ_CART) por cada bit que se transmite en serie, a una frecuencia de 5 KHz. El siguiente esquema muestra un cronograma de ejemplo:



En el cronograma de ejemplo se muestra la captura de cuatro columnas, la última de las cuales está en blanco (todos los bits a cero). Cada columna genera 16 IRQs, con un tiempo total de 3,2 milisegundos. Se advierte al usuario que no deslice el sensor a más de 4 cm/s, por tanto, el tiempo mínimo entre el inicio de una columna y el inicio de la siguiente será 10,4 ms.

Los bits se envían empezando por el píxel superior. En el siguiente esquema se muestra un ejemplo de captura de los bits para 48 columnas consecutivas situadas en medio de una frase:



A cada activación de IRQ se ejecutará una RSI que denominaremos `rsi_scanner()`. Esta RSI se debe encargar de ir recibiendo los 16 bits de cada columna, e ir almacenando dichos bits en un vector global de nombre `vcols[]`. Los bits se deben ir insertando (uno a uno) en la posición del vector indexada por la variable global `indx_col`. En el esquema anterior se muestra el contenido de `vcols[]` correspondiente a las 48 columnas del ejemplo: se ha usado binario para resaltar la forma de las letras capturadas ('d', 'e', 'l' y parte de la 'h'). Hay que observar que el píxel superior de la columna se almacena en el bit 15, mientras que el píxel inferior se almacena en el bit 0 de cada posición del vector.

Para saber el número de bits insertados en la columna actual dispondremos de otra variable global denominada `bit_count`. Cuando `bit_count` llegue a 16, se debe avanzar el índice `indx_col` hacia la siguiente posición del vector. Hay que tener en cuenta que el recorrido del vector será circular, es decir, cuando se llega a la última posición (99), después se continúa por la primera (0). En el ejemplo anterior ha coincidido el salto circular con la captura de la primera columna de la 'e'.

Además, la RSI deberá tener constancia de si está capturando columnas de carácter o de separación. Para ello se sugiere usar la variable global `in_gap`, la cual valdrá 0 para columnas de carácter o 1 para columnas de separación. En el esquema anterior se muestra el valor de `in_gap` para las columnas de la 0 a la 15.

Se supone que todos los caracteres son compactos, es decir, no hay ninguna columna de separación en medio del carácter. Por este motivo, cuando la variable `in_gap` vale 1 y se recibe una columna no vacía, sabremos que empieza un nuevo carácter. En este momento hay que guardar el valor de `indx_col` en otra variable global `indx_col_ini`. Esta nueva variable de índice de columna inicial del carácter se usará en el programa principal para procesar todas las columnas del carácter. En el esquema anterior se indica como ejemplo que la variable `indx_col_ini` memoriza el índice inicial de la letra 'h' (29).

Por otro lado, cuando la variable `in_gap` vale 0 y se recibe una columna vacía, sabremos que se ha terminado un carácter. En este momento hay que avisar al programa principal poniendo a 1 la variable global `init_ocr`.

Se dispone de las siguientes rutinas ya implementadas:

<i>Rutina</i>	<i>Descripción</i>
<code>inicializaciones ()</code>	Inicializa el <i>hardware</i> (pantalla, interrupciones, etc.)
<code>swiWaitForVBlank ()</code>	Espera hasta el próximo retroceso vertical
<code>printf(char *format, ...)</code>	Escribe un mensaje en la pantalla inferior de la NDS
<code>reconocer_caracter(unsigned short image[], int lon);</code>	Aplica un reconocimiento óptico de caracteres sobre la imagen en blanco y negro (1 bpp) registrada en el vector <code>image[]</code> (16 píxeles/posición), de longitud <code>lon</code> ; devuelve un código ASCII (tipo <code>char</code>) del carácter reconocido, o 255 si no puede reconocer ningún carácter. Tiempo máximo de ejecución: 60 ms.

Por su parte, el programa principal deberá detectar la activación de `init_ocr` y copiar todas las columnas del carácter que empieza en `indx_col_ini` en un vector temporal `vtemp[]`. El número de columnas copiadas dependerá del ancho del carácter, que podrá tener entre 2 y 23 columnas. Siempre se añadirá una columna vacía al final del carácter. La copia se debe realizar teniendo en cuenta la circularidad de `vcols[]`, por este motivo se usará la variable local `circ_ind`. Después de la copia se llamará a la rutina `reconocer_caracter()`, cuyo resultado se puede escribir directamente por pantalla con `printf()`, incluso si no ha podido reconocer el carácter (el código 255 mostrará un cuadrado).

Además, el programa principal deberá escribir los espacios en blanco necesarios entre palabras. Para ello se registrará en la variable local `circ_ind_ant` el índice de la última columna vacía del carácter anterior. Se supone que cada espacio en blanco ocupará 10 columnas vacías, y puede haber más de un espacio en blanco entre palabras.

Se propone el siguiente código para el programa principal:

```
#define MAX_COLS 100
unsigned short Vcols[MAX_COLS]; // vector con píxeles de las columnas
unsigned short indx_col = 0; // índice de la columna actual
unsigned short indx_col_ini; // índice inicial de un nuevo carácter
unsigned char bit_count = 0; // número de bits recibidos [0..16]
unsigned char in_gap = 1; // indica si está dentro de separación
unsigned char init_ocr = 0; // indica que se puede iniciar el ocr

void main(void)
{
    unsigned short Vtemp[24]; // vector temporal de columnas de un car.
    unsigned short i, circ_ind, circ_ind_ant = 0;
    short dist;

    inicializaciones();
    do
    {
        swiWaitForVBlank(); // relaja CPU y sincroniza con pantalla
        if (init_ocr) // si se ha capturado un nuevo carácter
        {
            init_ocr = 0; // reset del indicador
            // calcular distancia entre última columna
            // copiada y nueva columna inicial de car.
            dist = indx_col_ini - circ_ind_ant;
            if (dist < 0)
            {
                dist = dist + MAX_COLS; // ajuste circular de distancia
            }
            for (i = 0; i_a_?; i++)
            {
                // escribir espacios en blanco entre
                printf(" "); // palabras (bloques de 10 columnas
                // vacías)
            }
            i = 0;
            do // copiar columnas del último carácter
            {
                circ_ind = i_b_?; // índice circular de col. a copiar
                Vtemp[i] = Vcols[circ_ind];
                i++;
            } while (Vcols[circ_ind] != 0); // hasta primera col. vacía
            circ_ind_ant = circ_ind; // registrar índice última col. copiada
            printf("%c", reconocer_caracter(i_c_?));
        }
    } while (1);
}
```

Se pide:

rsi_scanner() en lenguaje ensamblador, partes del programa principal marcadas con *i_x_?* en lenguaje C.

Problema 41 (1^a Conv. 2023-24): Semáforo con botón emergencia

Hay que implementar parte del código fuente de un programa para simular el funcionamiento de un semáforo mediante la NDS. La siguiente figura muestra un ejemplo del aspecto de las dos pantallas:



El semáforo funcionará de la siguiente manera:

- en el modo "normal", se mostrarán cíclicamente las luces de color rojo, verde y amarillo durante unos tiempos determinados, por ejemplo, 3 segundos en rojo, 2 segundos en verde y 1,5 segundos en amarillo;
- en el modo "atención", se mostrará la luz amarilla intermitentemente, es decir, un determinado tiempo encendida y otro tiempo apagada, por ejemplo, 0,5 segundos en cada estado;
- cuando se pulse el botón A se cambiará inmediatamente de modo, volviendo al inicio de la secuencia correspondiente al nuevo modo.

Se dispone de una función ya implementada, de nombre `actualizar_luces()`, a la que se le pasa un código binario por parámetro: el valor de los tres bits de menos peso de dicho código determinará qué colores se deben mostrar, rojo (bit 0), amarillo (bit 1) y verde (bit 2).

Para codificar las secuencias de cada modo de trabajo del semáforo se proponen dos tablas, `luces[2][4]` y `tiempos[2][4]`:

```
// variables globales

// luces[2][4]: fija el código binario de luces para los dos patrones
//           bit 0 -> Rojo, bit 1 -> Amarillo, bit 2 -> Verde
//           el valor -1 indica final de códigos de cada patrón (centinela)
char luces[2][4] = { {1, 4, 2, -1},           // rojo > verde > amarillo
                     {0, 2, -1, -1} };      // apagado > amarillo

// tiempos[2][4]: fija el tiempo para cada luz de cada patrón, en décimas
//                 de segundo, por ejemplo, un 15 indica 1,5 segundos
//                 el valor -1 indica final de tiempos de cada patrón (centinela)
short tiempos[2][4] = { {30, 20, 15, -1},        // 3,0s > 2,0s > 1,5s
                        {5, 5, -1, -1} };      // 0,5s > 0,5s
```

El propósito de estas tablas es flexibilizar la especificación de las secuencias de cada modo. A cada secuencia se la denominará *patrón*. Se han definido dos patrones, con índice 0 para el modo "normal" y con índice 1 para el modo "atención". Aunque la longitud máxima de las secuencias es de 3 elementos, se ha decidido reservar 4 posiciones por secuencia para incluir un elemento centinela que marcará el final de cada secuencia.

Para gestionar el índice del patrón actual y el índice del elemento actual dentro de la secuencia, se proponen las siguientes variables globales:

```
unsigned char patron = 0;           // indica el patrón actual
unsigned char indx = 0;           // índice del elemento actual del patrón actual
```

Además, se requieren otras dos variables globales booleanas ($=0 \rightarrow$ falso, $\neq 0 \rightarrow$ cierto):

```
unsigned char botpul = 0;           // indica si se ha pulsado el botón A
unsigned char timer0_on = 0;         // indica si el timer 0 está en marcha
```

El funcionamiento propuesto para el programa principal es el siguiente:

- se inicializarán los gráficos para mostrar las luces (*sprites*) y los mensajes
- se inicializarán las interrupciones del *Vertical Blank*, de los *timers* 0 y 1, y del botón **A**
- se inicializará el divisor de frecuencia del *timer* 1 para que tenga un periodo aproximado de un segundo, utilizando la frecuencia de entrada 3 (≈ 32.728 Hz)
- el bucle principal se encargará de ir mostrando los colores del semáforo según el patrón e índice actuales, retardándose el tiempo necesario para cada elemento de secuencia con llamadas a la rutina `retardo_ms()`
- dicha rutina de retardo retornará por uno de los siguientes dos motivos:
 - porqué se ha agotado el tiempo de retardo: en este caso, en el bucle principal se incrementará el índice actual y, si llega al final de la secuencia, se reinicializará dicho índice
 - porqué se ha pulsado el botón **A** antes del final del retardo: en este caso, en el bucle principal se actualizarán los mensajes que indican la secuencia del patrón actual

Para implementar la rutina `retardo_ms()` se usará el *timer* 0 como cronómetro y la RSI del botón **A** para poder interrumpir dicho retardo. El funcionamiento propuesto es el siguiente:

1. la rutina recibe por parámetro el número de milisegundos a esperar (`unsigned short milliseconds`)

2. declara una variable local `milis` que servirá para calcular el periodo del cronómetro (*timer 0*); en principio, `milis` será igual a `miliseconds`, pero hay que tener en cuenta que un *timer* de la NDS no puede manejar un periodo superior a 2 segundos (2.000 milisegundos), por lo que la variable `milis` servirá para dividir el número total de milisegundos en segmentos de 1.024, por ejemplo, en el caso de un retardo de 3.000 ms se producirán tres segmentos, dos de 1.024 ms más uno 952 ms (=3.000 - 2.048)
3. la rutina programará el divisor de frecuencia del *timer 0* para que el periodo del *timer* sea prácticamente igual al número de milisegundos del segmento de tiempo actual obtenido con la variable `milis`, e iniciará dicho *timer* en modo interrupciones, con frecuencia de entrada 3 (≈ 32.728 Hz)
4. luego fijará el valor de la variable global `timer0_on` a uno, y entrará en un bucle de espera bloqueante llamando a la rutina de soporte `swiWaitForIRQ()`, la cual pone el procesador en reposo hasta que se produce una interrupción cualquiera
5. dicho bucle de espera solo terminará en el caso de que la variable `timer0_on` valga cero: este cambio de valor de la variable se realizará dentro de la RSI del *timer 0*, cuando haya finalizado el periodo de espera, o bien dentro de la RSI de los botones, cuando el usuario pulse el botón A
6. una vez terminado el bucle de espera bloqueante, la rutina `retardo_ms()` comprobará por qué motivo se ha salido de dicho bucle:
 - si la variable global `botpul` es diferente de cero, la rutina finalizará su ejecución
 - si no, se restará el periodo del segmento de tiempo finalizado al parámetro `miliseconds` y, en caso de que queden milisegundos restantes, se repetirá el proceso desde el paso 3; en otro caso, la rutina finalizará su ejecución
7. cuando finaliza su ejecución, la rutina `retardo_ms()` devuelve (por el registro R0) el número de milisegundos restantes; por lo tanto, el código que invoque a esta rutina podrá determinar el motivo del retorno comprobando si el número de milisegundos restantes es igual a cero (fin del retardo) o diferente de cero (pulsación del botón A).

Por otro lado, la RSI de los botones se activará solo cuando se pulse el botón A, puesto que así se configurará en las inicializaciones del controlador de botones. Esta RSI deberá realizar las siguientes tareas:

- poner la variable global `botpuls` a uno,
- intercambiar la variable global `patron` para pasar al patrón contrario, y poner la variable global `indx` a cero para empezar la nueva secuencia desde el principio,
- parar el *timer 0* y poner la variable global `timer0_on` a cero, para provocar la finalización del bucle de espera bloqueante de la rutina `retardo_ms()`

- desactivar las interrupciones de los botones y poner en marcha el *timer* 1, con el propósito de evitar nuevas interrupciones del botón A durante un pequeño periodo de tiempo (un segundo)
- la RSI del *timer* 1 se activará después de un segundo de haberse puesto en marcha; entonces reactivará las interrupciones del botón A y parará el propio *timer* 1.

La siguiente tabla resume el funcionamiento de las rutinas y funciones ya implementadas:

Rutina	Descripción
initGraficosA() initGraficosB()	Inicializa las pantallas de la NDS
swiWaitForVBlank()	Espera hasta el próximo retroceso vertical
actualizar_luces(unsigned char bincode)	Muestra las luces como círculos (<i>sprites</i>) de color rojo, amarillo y verde, según un código binario de 3 bits que se pasa por parámetro
mensajes(patron)	Escribe en la pantalla superior la secuencia de luces y tiempos del patrón que se pasa por parámetro
INT_instalarRSIPrincipal(u32* irq_handler, void* rpsi, int mascara)	Instala la rutina principal de servicio de interrupciones cuya dirección inicial se pasa por el parámetro rpsi, en la posición de memoria indicada en irq_handler, y activando los bits de mascara en el registro REG_IE
rsi_principal()	Rutina principal de servicio de interrupciones, que invocará a las RSIs pertinentes según el contenido del registro REG_IF, además de resetear los flags activados y escribir dichos flags en la posición de memoria INTR_WAIT_FLAGS
rsi_vblank()	RSI del <i>Vertical Blank</i> : actualiza los registros OAM
rsi_timer0()	RSI del <i>timer</i> 0: escribe un cero en la variable global timer0_on y para el propio <i>timer</i>
rsi_timer1()	RSI del <i>timer</i> 1: activa el bit 14 del registro REG_KEYCNT (habilita las interrupciones de botones) y para el propio <i>timer</i>

Se pide:

Completar en la hoja de respuestas los trozos de código fuente que faltan en la propuesta de implementación.

Problema 42 (1^a Conv. 2023-24): Seguimiento de caminos

Hay que implementar parte del código fuente de un programa para la NDS cuyo propósito es establecer la capacidad de una persona para mantener uno, dos o tres objetos (bolas) dentro de sus respectivos caminos o pistas. Este tipo de programas se usan para medir los reflejos del usuario, por ejemplo, como parte de un test psicotécnico para obtener el permiso de conducir. La siguiente figura muestra un ejemplo del aspecto de las pantallas cuando se están siguiendo dos caminos:



El funcionamiento general del programa, una vez fijado el número de caminos o pistas, será el siguiente:

- inicialmente aparecerán los caminos rectos, con los objetos centrados en cada camino y en una posición de pantalla bastante baja;
- los caminos se irán desplazando hacia abajo, generando curvas aleatorias;
- el usuario deberá ir pulsando los botones de la NDS correspondientes para intentar mantener cada objeto dentro de su camino;
- si un objeto toca el borde del camino, el programa moverá automáticamente el objeto hacia dentro del camino otra vez, pero acumulará puntos negativos en un contador interno (no se muestra en el gráfico anterior);
- periódicamente, el programa irá desplazando los objetos hacia la parte superior de la pantalla, de modo que cada vez será más difícil mantener los objetos dentro de sus respectivos caminos;
- cuando los objetos lleguen prácticamente a la parte superior de la pantalla, el proceso de seguimiento se detendrá y se mostrará por pantalla el resultado de la prueba (puntos negativos acumulados, consideración de apto o no apto, etc.).

La parte que hay que programar en este problema es básicamente la relativa al movimiento de los objetos, además del desplazamiento del camino. Para gestionar el número de pistas (*tracks*), la posición de los objetos (*trackers*) y los botones del movimiento, se proponen las siguientes variables globales:

```
#define MAX_TRACKS 3
// variables globales
unsigned char num_tracks;           // número actual de pistas
unsigned char track_width;          // ancho de pistas
unsigned int keyMasks[] = {KEY_LEFT, KEY_RIGHT,      // máscaras para
                          KEY_B, KEY_A,        // comprobar la pulsación de
                          KEY_L, KEY_R};      // botones de cada tracker
unsigned char pos_hori[MAX_TRACKS];   // posición horizontal de trackers
unsigned char pos_vert[MAX_TRACKS];   // posición vertical de trackers
short divFreq[MAX_TRACKS];          // divisores de frecuencia de los timers
```

Las variables `num_tracks` y `track_width` mantendrán en todo momento el número actual de caminos y el ancho de cada camino (en píxeles).

El vector `keyMasks[]` almacenará las máscaras binarias correspondientes a los botones de desplazamiento hacia la izquierda y hacia la derecha de cada objeto, en orden consecutivo, es decir, las posiciones 0 y 1 para el primer objeto, las posiciones 2 y 3 para el segundo objeto y las posiciones 4 y 5 para el tercer objeto.

Los vectores `pos_hori[]` y `pos_vert[]` almacenarán la posición actual horizontal y vertical de cada objeto.

Por último, el vector `divFreq[]` almacenará el divisor de frecuencia actual de los tres *timers* que se usarán para detectar los botones de desplazamiento de cada objeto (*timers* 1, 2 y 3).

La siguiente tabla resume el funcionamiento de las rutinas y funciones ya implementadas:

<i>Rutina</i>	<i>Descripción</i>
<code>inicializaciones ()</code>	Efectúa las inicializaciones del hardware
<code>initTracks(unsigned char nt);</code>	Efectúa las inicializaciones de los elementos gráficos para el número de caminos indicado
<code>swiWaitForVBlank ()</code>	Espera hasta el próximo retroceso vertical
<code>desplazar_caminos ()</code>	Mueve todo el fondo gráfico de los caminos un píxel hacia abajo, generando una nueva línea de píxeles en la primera fila de la pantalla

<code>comprobar_colisiones()</code>	Detecta si existe algún objeto que esté tocando el borde de su camino; en caso afirmativo, mueve dicho objeto hacia el centro del camino y acumula la correspondiente puntuación negativa
<code>SPR_moverSprite(int indice, int px, int py)</code>	Mueve el <i>sprite</i> indicado por el primer parámetro a las coordenadas <i>px</i> y <i>py</i> de la pantalla. Atención , para que el movimiento sea visible en pantalla, será necesario poner a 1 la variable global <code>update_spr</code>
<code>mostrar_resultados()</code>	Visualiza, por la pantalla de mensajes de texto, los resultados finales del test para el número actual de caminos
<code>rsi_vblank()</code>	RSI de <i>Vertical Blank</i> : actualiza los registros OAM
<code>scanKeys()</code>	Captura el estado de los botones
<code>keysDown()</code>	Devuelve el estado capturado de los botones (1: pulsado, 0: soltado)

El funcionamiento propuesto para el programa principal es el siguiente:

- inicializaciones generales
- ir desplazando los caminos hacia abajo, a 60 filas por segundo para un camino, 30 filas por segundo para dos caminos y 20 filas por segundo para tres caminos
- ir comprobando las posibles colisiones, resituar objetos y contabilizar penalizaciones
- cada medio segundo, desplazar los objetos una fila hacia arriba
- cuando los objetos llegan a una posición suficientemente alta:
 - detener el proceso de seguimiento
 - mostrar los resultados del test
 - esperar a que el usuario pulse el botón START
 - reiniciar un nuevo test, con un nuevo número de caminos

Para representar los objetos se usarán los *sprites* 0, 1 y 2.

Para detectar las pulsaciones de los botones correspondientes, se propone usar las RSIs de los *timers* 1, 2 y 3. Estos *timers* estarán inicialmente configurados con un divisor de frecuencia tal que la activación de la RSI se producirá a una frecuencia de salida de aproximadamente 20 Hz. Sin embargo, a medida que el usuario mantenga pulsado cualquiera de los dos botones para cambiar la posición horizontal del objeto, el divisor de frecuencia se irá modificando progresivamente hasta llegar a generar una frecuencia de 80 Hz. Cuando el usuario libere los dos botones, el divisor de frecuencia se irá restituyendo progresivamente hasta llegar a generar los 20 Hz originales. De este modo se pretende generar un efecto de aceleración, que permitirá al usuario realizar movimientos rápidos cuando sea necesario.

Para implementar las rutinas de los tres *timers*, se propone definir una rutina común de nombre `rsi_timerB()`, que se llamará desde la rutina principal de servicio de interrupciones (RPSI) pasando un parámetro por el registro `R0`, que será 0, 1 o 2 según el *timer* que haya provocado la IRQ (1, 2 o 3, respectivamente). El funcionamiento de esta rutina común `rsi_timerB()` básicamente será el siguiente:

1. la rutina recibe por parámetro el índice del objeto (0, 1 o 2) que se debe gestionar,
2. lee el estado de los botones de la NDS (a nivel *hardware*, 0: pulsado, 1: soltado),
3. comprueba si se ha pulsado alguno de los dos botones asociados al objeto,
4. incrementa o decrementa la posición horizontal del objeto,
5. actualiza la visualización del *sprite* correspondiente,
6. si ha habido pulsación y no llega al límite de velocidad máxima, o bien si no ha habido pulsación y no llega al límite de velocidad mínima, actualizar el divisor de frecuencia asociado; los registros de datos de los *timers* se encuentran separados por 4 bytes en el mapa de memoria; la frecuencia de entrada de los *timers* 1, 2 y 3 será la más lenta, es decir, 32.728 Hz.

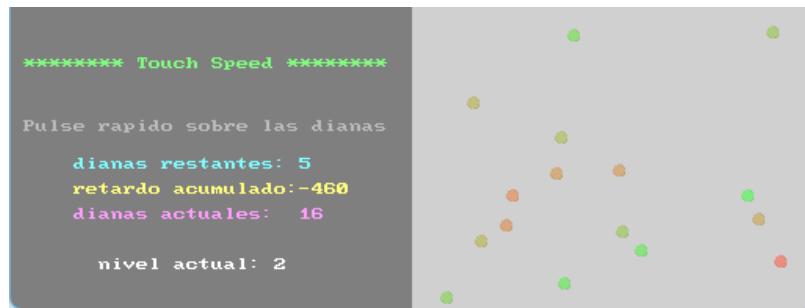
Por otro lado, la RSI principal deberá comprobar los bits de activación de IRQs del registro `REG_IF`, para llamar a la rutina `rsi_timerB()` según la activación de dichos bits; hay que recordar que el bit correspondiente a `IRQ_TIMER1` es el 4, y que para los siguientes *timers* 2 y 3 están reservados los bits consecutivos de más peso.

Se pide:

Completar en la hoja de respuestas los trozos de código fuente que faltan en la propuesta de implementación.

Problema 43 (2^a Conv. 2023-24): Velocidad de toque

Hay que implementar parte del código fuente de un programa cuyo propósito es indicar la velocidad de toque sobre dianas que aparecen en la pantalla sensible al tacto de la NDS. La siguiente figura muestra un ejemplo del aspecto de las pantallas cuando se está pulsando sobre las dianas, que se representan como círculos de diversos colores:



Para cada uno de los cinco niveles de dificultad (del 0 al 4), el funcionamiento del programa será el siguiente:

- inicialmente se calcula el número de dianas pendientes de ser generadas (dianas restantes): 10, 20, 30, 40 o 50 dianas, respectivamente para los niveles 0, 1, 2, 3 y 4;
- se inicializa a cero el retardo acumulado del nivel, así como las dianas visibles en cada momento (dianas actuales);
- el programa irá generando dianas en posiciones aleatorias de la pantalla sensible al tacto durante 10 segundos (en cualquier nivel) a ritmo constante; para cada nueva diana generada se decrementará el número de dianas restantes y se incrementará el número de dianas actuales;
- el usuario irá tocando sobre las dianas visibles; cuando se toque una diana, ésta desaparecerá de pantalla y se decrementará el número de dianas actuales;
- periódicamente, el programa irá cambiando el color de las dianas visibles; las dianas aparecen inicialmente con un color verde, e irán pasando por 14 gradaciones hasta llegar a un color rojo (en total, 16 colores); una vez que una diana llega al último color, ya no habrá más cambios para esa diana;
- la frecuencia de cambio de gradaciones de color se establece (para todos los niveles) en 16 cambios en 8 segundos.
- la gradación del color indica un factor de retardo para cada diana; este factor de retardo variará desde 0 (color verde) a 15 (color rojo), incrementándose de uno en uno para cada gradación;
- el factor de retardo se usa para calcular el retardo acumulado: cada vez que se deba cambiar, se restará dicho factor al retardo acumulado actual; esto significa que las dianas restarán más cuanto más tiempo permanezcan en pantalla; incluso cuando una diana ha llegado a su factor de retardo máximo (15), dicha diana continuará restando 15 unidades al retardo acumulado actual, mientras no se elimine de pantalla;

- una vez generadas todas las dianas del nivel, el programa permitirá al usuario seguir eliminando dianas durante 1, 2, 3, 4 o 5 segundos, respectivamente para cada nivel;
- agotado este tiempo extra, el programa irá eliminando automáticamente (recolectando) todas las dianas que queden todavía en pantalla, a una velocidad de diez dianas por segundo; durante el tiempo que tarde el programa en recolectar todas las dianas actuales, el usuario podrá seguir eliminando dianas por su cuenta;
- una vez finalizado este último proceso, el programa pedirá al usuario que pulse el botón START para pasar al siguiente nivel; después del nivel 4 se retornará al nivel 0;
- obviamente, el objetivo es conseguir que el valor final del retardo acumulado de cada nivel resulte lo menos bajo (negativo) posible.

Para gestionar la información de todas las dianas visibles y del funcionamiento general del programa, se proponen las siguientes variables globales:

```
#define MAXLEVEL      5    // número de niveles
#define MAXTARGETS   50    // número de dianas a generar en el nivel máximo

typedef struct           // información de cada diana
{
    unsigned char act;          // =1 -> activa, =0 -> no activa
    unsigned char px;           // coordenada x (0..248)
    unsigned char py;           // coordenada y (0..184)
    unsigned char df;           // factor de retardo (0..15)
} target_t;

unsigned char Level = 0;    // nivel de dificultad (0..MAXLEVEL-1)
unsigned char Tcurr = 0;    // número de dianas actualmente en pantalla
unsigned char Trest;       // núm. de dianas pendientes de ser generadas
unsigned char Tend = 0;     // núm. máximo de entradas en vector targets[]
short Delay;               // retardo acumulado global (negativo)

target_t targets[MAXTARGETS]; // vector de información de las dianas
```

Los comentarios de cada variable son suficientemente explícitos para todas las entidades, excepto para `Tend` y su relación con el contenido del vector `targets[]`, que se mostrará más adelante con un ejemplo.

La siguiente tabla resume el funcionamiento de las rutinas y funciones ya implementadas:

<i>Rutina</i>	<i>Descripción</i>
<code>initGraficosA()</code> <code>initGraficosB()</code>	Inicializaciones de los dos procesadores gráficos
<code>reinicio(unsigned char nivel)</code>	Efectúa las inicializaciones del nivel indicado
<code>swiWaitForVBlank()</code>	Espera hasta el próximo retroceso vertical

<code>generar_diana(unsigned char * end_targets)</code>	Genera una nueva diana sobre pantalla y sobre el vector <code>targets[]</code> ; el parámetro pasado por referencia indica el número más alto de entradas del vector utilizadas hasta el momento; la función busca una posición libre en <code>targets[]</code> entre 0 y <code>end_targets-1</code> ; si no encuentra ninguna, utilizará una nueva posición de <code>targets[]</code> e incrementará <code>end_targets</code>
<code>mostrar_contadores(unsigned char tr, short dly, unsigned char tc)</code>	Escribe en la pantalla de control los valores pasados por parámetro, <code>tr</code> : dianas restantes, <code>dly</code> : retardo acumulado, <code>tc</code> : dianas actuales
<code>SPR_cambiarSprite(int indice, int baldosa, int paleta)</code>	Cambia los índices de baldosa inicial y paleta de color del <i>sprite</i> indicado por el primer parámetro, a los valores del segundo y tercer parámetros; para este problema, <code>indice</code> será el mismo índice de la diana usado en el vector <code>targets[]</code> , <code>baldosa</code> será cero y <code>paleta</code> será el número actual de la gradación de color de la diana (0..15)
<code>SPR_borrarSprite(int indice)</code>	Elimina el <i>sprite</i> indicado por parámetro
<code>INT_instalarRSIPrincipal(u32* irq_handler, void* rpsi, int mascara)</code>	Instala la rutina principal de servicio de interrupciones, según parámetros <code>irq_handler</code> (posición de instalación), <code>rpsi</code> (dirección inicial) y <code>mascara</code> (bits del registro REG_IF)
<code>rsi_principal()</code>	Rutina principal de servicio de interrupciones, que invocará a las RSIs pertinentes según el contenido del registro REG_IF
<code>rsi_vblank()</code>	RSI del <i>Vertical Blank</i> : actualiza los registros OAM de los <i>sprites</i> correspondientes a las dianas activas
<code>retardo(unsigned char ds)</code>	Detiene el programa durante el número de décimas de segundo indicado por parámetro
<code>recolectar_dianas()</code>	Elimina las dianas activas progresivamente (a 10 dianas/segundo), actualizando los contadores

Se propone estructurar el funcionamiento del proyecto en tres partes:

programa principal:

- para cada nivel, calcular el número de dianas restantes `Trest`
- generar todas las dianas restantes en 10 segundos, incrementando `Tcurr` y decrementando `Trest` a cada diana generada
- esperar 1, 2, 3, 4 o 5 segundos, según el nivel actual
- recolectar las dianas visibles
- cambiar el nivel actual y reiniciar el proceso

- RSI del *timer 0*:
- para cada diana activa, obtener su factor de retardo actual
 - restar el factor de retardo actual al retardo acumulado
 - si el factor de retardo es menor que 15, incrementar dicho factor y cambiar el color del *sprite* correspondiente

- RSI de *ipc_fifo_receive*:
- cada vez que el usuario pulse sobre la pantalla táctil, el procesador ARM7 enviará un mensaje de 32 bits a su cola IPC_FIFO de envío, con la coordenada x (0..255) en los 16 bits altos y la coordenada y (0..191) en los 16 bits bajos
 - el procesador ARM9 recibirá el mensaje en su cola IPC_FIFO de recepción, junto con la petición de interrupción que indica que dicha cola no está vacía (IRQ_FIFO_NOT_EMPTY)
 - entonces se llamará a la rutina `rsi_ipcreceive()`, que debe extraer el mensaje de la cola, separar las coordenadas x e y del mensaje, y buscar entre todas las dianas activas si las coordenadas de la pulsación están entre la posición (px, py) y la posición (px+8, py+8) de cada diana inspeccionada
 - en caso afirmativo, borrará el *sprite* correspondiente y desactivará la entrada correspondiente en el vector `targets[]`, además de decrementar `Tcurr`
 - en el caso que el índice de la diana desactivada fuese `Tend-1`, se decrementará el valor de `Tend` para reducir el número máximo de entradas a gestionar del vector `targets[]`

Por último, se propone un ejemplo de variación del contenido del vector `targets[]` junto con los valores correspondientes de `Tcurr` y `Tend`, suponiendo un cierto estado de ejemplo (5 dianas activas) más diversos eventos de generar y eliminar dianas:

	0	1	2	3	4	5	6	7	...	<code>Tcurr</code>	<code>Tend</code>
estado ejemplo:	d0	d1	d2	d3	d4					5	5
elimina d2:	d0	d1		d3	d4					4	5
genera diana (d2):	d0	d1	d2	d3	d4					5	5
genera diana (d5):	d0	d1	d2	d3	d4	d5				6	6
elimina d1:	d0		d2	d3	d4	d5				5	6
elimina d5:	d0		d2	d3	d4					4	5
elimina d3:	d0		d2		d4					3	5
genera diana (d1):	d0	d1	d2		d4					4	5

Se pide:

Completar en la hoja de respuestas los trozos de código fuente que faltan en la propuesta de implementación.

Propuesta Implementación Problema 3

```

int main()
{
    short freq;           // variables locales
    int keys;

    inicializaciones();
    do
    {
        swiWaitForVBlank(); // ajustar velocidad encuesta periódica
        scanKeys();
        keys = keysDown(); // captura pulsación actual
        ¿ a ?           // determinar frecuencia de vibración
        if (keys & KEY_X) freq = 5;
        if (keys & KEY_Y) freq = 20;
        if (keys & KEY_A) freq = 50;
        if (freq != 0)
        {
            generar_vibracion(freq); // activar vibración
            retardo(5);             // esperar medio segundo
            ¿ b ?               // parar vibración
            ¿ c ?
            printf("Ultima frecuencia = %d Hz\n", freq);
        }
    } while (1);
    return(0);
}

```

@; generar_vibración: dada una frecuencia por parámetro, calcula el divisor de frecuencia para el timer0 y lo activa; si la frecuencia es cero, desactiva el timer0.

@; Parámetro:

@; R0 = frecuencia requerida, en Hz

generar_vibracion:

push {r0-r3, lr} @;salvar reg. modificados (R3 por swi 9)

ldr r2, =TIMER0_DATA @; R2 apunta a registros de timer0

¿ d ?

beq .Lfin_vibracion

¿ e ?

@; R1 = freq. salida (denominador)
@; R0 = freq. entrada (numerador)
@; llamada a la BIOS (dividir)
@; negar el divisor de frecuencia

```
mov r0, r0, lsl #16  
    ; f ?  
orr r0, #0x00C30000  
    ; poner a 0 los 16 bits altos  
    ; adjuntar valores para TIMER0_CR  
    ; (bit 7 = 1, bit 6 = 1,  
    ; bits 10 = 11)  
.Lfin_vibracion:  
    ; g ?  
    ; fijar TIMER0_DATA y TIMER0_CR  
    ; si R0 = 0, para la vibración  
pop {r0-r3, pc}
```

;; RSI timer0: se activa a la frecuencia programada (5, 20 o
;; 50 Hz). Cambia el estado del bit 1 del registro REG_RUMBLE.
RSI_Timer0:

```
    ; h ?  
  
    ; i ?  
    ; R0 apunta al registro de E/S  
    ; cargar valor actual en R1  
    ; cambia el valor del bit 1  
    ; actualiza registro de E/S  
pop {r0-r1, pc}
```

Propuesta Implementación Problema 13

```

#define MAX_PER = 50          // número máximo de periodos entre la
                           // detección de tensión mínima y máxima
#define T_MAX = 180           // tiempo máximo entre dos latidos de corazón,
                           // en retrocesos verticales (3 segundos)
#define P_UMB = 10            // presión umbral para retirar brazalete

unsigned char period[MAX_PER];    // vector de periodos
unsigned char i_per;              // índice del periodo actual
unsigned char tiempo;             // valor de tiempo actual
unsigned short presion;           // último valor de presión detectado
unsigned short minima, maxima;    // valores de tensión obtenidos

void main()
{
    inicializaciones();
    printf("Pulse START para iniciar la medición\n");
    do
    { do                                // esperar pulsación de inicio
      { scanKeys();
        swiWaitForVBlank();
      } while (!(keysDown() & KEYS_START));

      ¿ a ?                      // iniciar ciclo de subir presión
      clear();
      printf("Obteniendo datos:\n");
      i_per = 0;
      tiempo = 0;
      do                                // bucle detección presión mínima
      {
        swiWaitForVBlank();
      } while (i_per == 0);
      minima = presion;                 // obtener mínima en primer latido

      do                                // bucle detección presión máxima
      { swiWaitForVBlank();
        tiempo++;
      } while (¿ b ?);
      maxima = presion;                 // obtener máxima en último latido

      printf("Mínima = %d mmHg\n", minima);
      printf("Máxima = %d mmHg\n", maxima);
      printf(¿ c ?);

      TENS_CTRL = 0;                   // liberar presión
      while (TENS_DATA > P_UMB)      // esperar rebajar presión umbral
        swiWaitForVBlank();

      printf("Pulse START para una nueva medición\n");
    } while (1);
}

```

```

@; RSI del tensiómetro: se activa cada vez que detecta un latido completo
@; mientras se está aumentando la presión;
@; almacena el valor actual de la variable 'tiempo' en la posición
@; 'i_per' del vector 'period', pone a cero el tiempo e incrementa
@; la posición del vector; también lee el valor actual de presión y lo
@; guarda en la variable global 'presion'.
RSI_tensiometro:
    push {r0-r4, lr}

    ldr r0, =i_per
    ldrb r1, [r0]           @; R1 = valor de 'i_per' (posición vector)
    ldr r2, =tiempo

    ¿ d ?

    add r1, #1
    strb r1, [r0]           @; i_per++;
    ¿ e ?
    ldrh r1, [r0]           @; R1 = valor actual de presión
    ldr r2, =presion
    strh r1, [r2]           @; guardar valor presión en variable global

    pop {r0-r4, pc}

@; calcular Ritmo: obtiene el ritmo cardíaco como la media de los
@; periodos registrados en el vector que se pasa por parámetro (por
@; referencia), teniendo en cuenta que dichos periodos se expresan en
@; número de retrocesos verticales (60 Hz).
@; Parámetros:
@;   R0 = dirección inicial vector de periodos
@;   R1 = número de elementos registrados en el vector
@; Resultado:
@;   R0 = pulsaciones por minuto (byte)
calcular_ritmo:
    push {r1-r4, lr}

    mov r2, #1               @; R2 es índice vector (obviar primer elem.)
    mov r3, #0               @; R3 es el valor acumulado de periodos
.Lritmo_for:
    ¿ f ?

    mov r0, r3
    sub r1, #1
    ¿ g ?
    mov r1, r0
    ¿ h ?
    ¿ i ?

    ldrb r1, [r0+r2]          @; R4 = periodos[i];
    add r3, r1                @; acumular periodo
    add r2, #1                @; i++;
    cmp r2, r1                @; repetir para todos los elementos del vector

    sub r0, r3                @; R0 = sumatorio de periodos (excepto primer)
    sub r1, #1                @; R1 = número de elementos (menos primero);
    mov r2, #0                @; R0 = periodo promedio entre pulsaciones,
    add r0, r1                @;     expresado en número de VBLs
    mov r3, r0                @; T = R0 / 60 (periodo expresado en segundos)
    mov r0, r3                @; F = 60 / T (frecuencia expresada en ppm)
    mov r3, r0                @; R0 = F ppm (60*60 / periodo promedio)

    pop {r1-r4, pc}

```

Propuesta Implementación Problema 14

```

char currentKey = -1;      // código de tecla actual
char tempKey = -1;        // código de tecla temporal (para el barrido)
char num_fila = 0;         // número de fila actual (0..3)

void main()
{
    char prevKey = -1;      // tecla anterior (variable local)

    inicializaciones();
    ¿ a ?                  // inicialmente no se selecciona ninguna fila
do
{
    tareas_independientes();
    if (¿ b ?)            // detección cambio de tecla
    {
        prevKey = currentKey;
        if (currentKey != -1)          // si hay pulsación
        {
            swiWaitForVBlank();
            ¿ c ?
        }
    }
} while (1);              // repetir siempre
}

@; descodificar_tecla: a partir del estado de los bits b3..b0 de
@; las columnas y de la fila que se pasa por parámetro, detectar
@; la columna de mayor peso de dicha fila que presenta pulsación
@; (bit a 0); a partir de esta información (fila, columna),
@; devolver el código numérico correspondiente (fila*4+columna),
@; o -1 si no hay pulsación en la fila.
@; Parámetros:
@; R0 = estado de las columnas (valor de bits b3..b0 de REG_TECL)
@; R1 = número de fila analizada (0..3)
@; Resultado:
@; R0 = código numérico de tecla
descodificar_tecla:
    push {r2-r3, lr}

    ¿ d ?                  @;R2 = código de columna testada (3,2,1,0)
    mov r3, #1              @;R3 es máscara de test (inicialmente, 0001)
.Ldesc_col:
    ¿ e ?                  @;salir del bucle si encuentra bit a 0
                            @;desplazar máscara a la izquierda
                            @;siguiente código columna (de mayor a menor)

```

```

bge .Ldesc_col           @; repetir mientras código de columna >= 0

    mov r0, #-1          @; R0 = código de no pulsación
    b .Ldesc_fin

.Ldesc_fincol:
    ¿ f ?             @; R0 = columna (r2) + fila*4 (r1 lsl #2)

.Ldesc_fin:
    pop {r2-r3, pc}

@; RSI del timer 0: se activa 40 veces por segundo, de modo que se
@; utilizará para activar el barrido de una fila, aunque se
@; interpretarán las columnas activas de la fila anterior, lo cual
@; introducirá un retardo de 25 milisegundos entre activación de
@; fila y consulta de columnas correspondientes. La RSI fijará el
@; código de tecla en la variable global currentKey, después de
@; analizar las 4 filas de cada barrido completo del teclado.

RSI_timer0:
    push {r0-r6, lr}

    ldr r6, =REG_TECL
    ldrh r0, [r6]          @; R0 = valor actual de REG_TECL
    ldr r2, =num_fila
    ldrb r1, [r2]          @; R1 = número de fila actual
    bl descodificar_tecla @; R0 = código de tecla de fila actual
    ldr r3, =tempKey
    ¿ g ?              @; R4 = código temporal de tecla
    cmp r0, r4
    ble .Lkey_cont1
    mov r4, r0              @; actualizar R4 si código de fila actual
                            @; es mayor que código temporal del barrido

.Lkey_cont1:
    add r1, #1            @; pasar a siguiente fila
    cmp r1, #4
    blo .Lkey_cont2
    ldr r5, =currentKey   @; si estamos en la "quinta" fila (R1==4)
    strb r4, [r5]          @; actualizar currentKey con código temporal
    ¿ h ?              @; reset código temporal
    mov r1, #0              @; reset número de fila

.Lkey_cont2:
    ¿ i ?              @; actualizar código de tecla temporal
                            @; actualizar número de fila actual
                            @; R0 tiene el bit de fila actual a cero
                            @; actualiza REG_TECL para siguiente interrup.

.Lkey_fin:
    pop {r0-r6, pc}

```

Propuesta Implementación Problema 20

```

unsigned short pasos = 0; // número de pasos pendientes de enviar
                         // (243 pasos por cm)
char sentido;           // 1 → forward, -1 → backward
unsigned char fase = 0; // fase actual (0..7)
                         // bits de salida b7-b4 por cada fase
char Vphases[] = {0x80, 0xC0, 0x40, 0x60, 0x20, 0x30, 0x10, 0x90};

void main()
{
    char velocidad;          // variables locales de la
    unsigned char avance;    // consigna actual

    inicializaciones();
    desactivar_timer0();     // inicialmente el motor estará parado
    do
    {
        tareas_independientes();
        if (a ?)             // si el motor está parado
        {
            siguiente_movimiento(&velocidad, &avance);
            if (velocidad != 0)
            {
                sentido = (velocidad < 0 ? -1 : 1); // signo de vel.
                velocidad = velocidad * sentido;    // v. absoluto de vel.
                b ?                                     // freq. pasos = pasos/cm * cm/s
                pasos = 243 * avance; // pasos avance = pasos/cm * cm
                if (pasos > 0) c ?
                swiWaitForVBlank();
                printf("v = %d cm/s\n", velocidad);
                printf("\t%c = %d cm\n", 'd'+ 2*sentido, avance);
                // 'f' = 'd'+ 2
                // 'b' = 'd'- 2
            }
        }
    } while (1);
}

```

@; intr_main: rutina principal de gestión de las interrupciones;
@; comprueba si se ha activado la RSI del timer 0 y, en caso
@; afirmativo, invoca a la rutina RSI_timer0(); además, debe
@; notificar la resolución de cualquier IRQ que se haya
@; producido al controlador de interrupciones y a la posición
@; global de memoria INTR_WAIT_FLAGS.

```
intr_main:
    push {r0-r2, lr}
```

¿ d ?

@;R1 = valor actual de REG_IF
@;verificar si se ha activado IRQ_TIMER0
@;en caso afirmativo,
@; llamar la RSI específica

```
str r1, [r0]
```

@;marcar todas las IRQ activadas en REG_IF

¿ e ?

```
str r1, [r2]
```

@;ídem en INTR_WAIT_FLAGS

```
pop {r0-r2, pc}
```

@;RSI del timer 0: se activará a la frecuencia de pasos calculada
@; según la velocidad de rotación requerida del motor; a cada
@; activación debe cambiar de fase, según el sentido actual de
@; avance y la fase anterior; además, debe decrementar el número
@; de pasos pendientes, y parar el timer si dicho número ha
@; llegado a cero.

RSI_timer0:

```
push {r0-r2, lr}
```

```
ldr r0, =fase
```

@;R1 es fase actual

```
ldrb r1, [r0]
```

@;R2 es sentido de avance

```
ldr r2, =sentido
```

@;actualizar fase (inc. o dec.)

```
ldsb r2, [r2]
```

@;módulo 8 para actualización circular

```
add r1, r2
```

@;actualiza variable global 'fase'

¿ f ?

```
strb r1, [r0]
```

ldr r0, =Vphases

@;R2 es estado de los bits de salida

¿ g ?

@;según la nueva fase

```
mov r0, #0x0A000000
```

@;R0 es dirección registro E/S

```
ldr r1, [r0]
```

@;R1 es valor actual del reg. E/S

¿ h ?

```
orr r1, r2
```

@;limpiar bits b7-b4

```
str r1, [r0]
```

@;añadir bits de fase

@;actualiza reg. E/S

¿ i ?

@;decrementa contador de pasos restantes
@;si llega a cero, FZ = 1
@;si FZ = 1, parar las interrupciones

```
pop {r0-r2, pc}
```

Propuesta Implementación Problema 22

```

#define NUM_PLANT      4           // número de plantas a gestionar

unsigned char planta_actual = 0;    // número de planta actual
unsigned char sentido = 1;          // sentido de movimiento actual
                                    //   = 1 → subida,
                                    //   = 2 → bajada
unsigned char nueva_planta = 0;     // indica si el ascensor ha llegado
                                    //   a una nueva planta

void main()
{
    int i;

    inicializaciones();
    do
    {
        tareas_independientes();
        if (nueva_planta)           // comprobar si hay detección de
                                    // llegada a una nueva planta
                                    // desmarcar variable
            nueva_planta = 0;
        swiWaitForVBlank();
        printf("Planta: %d\n", planta_actual);
        printf("Peticiones:\n\t");
        ¿ a ?                      // escribir peticiones pendientes
        {
            if (RA_Botons & (1 << i))
                printf("%d ", i);      // planta solicitada
            else
                printf("_ ");       // planta no solicitada
        }
        printf("\n");
        if (RA_Motor == 0)          // en caso de motor parado
            ¿ b ?                  // si hay peticiones pendientes,
                                    // el motor está parado, y ya se
                                    // ha procesado la detección de
                                    // nueva planta (puerta abierta)
        {
            cerrar_puerta();
            RA_Motor = ¿ c ?       // sentido de movimiento actual
        }
    } while (1);
}

```

```

@;RSI de la interfaz con el ascensor: se activará cada vez que el
@; ascensor llegue a una nueva planta; activará la variable global
@; 'nueva_planta' y actualizará 'planta_actual' consecuentemente;
@; además, si la planta a la que acaba de llegar el ascensor
@; estaba solicitada, parará el motor y desactivará el bit de petición
@; correspondiente.

RSI_Ascensor:
    push {r0-r7, lr}

    ldr r0, =nueva_planta
    mov r1, #1
    strb r1, [r0]           @;activar variable 'nueva_planta'

    ldr r4, =planta_actual
    ldrb r5, [r4]           @;R5 = número de planta actual
    mov r6, #1
    ¿ d ?                 @;R6 = máscara de planta actual
    ldr r7, =sentido
    ldrb r7, [r7]           @;R7 = sentido de movimiento actual
    cmp r7, #1

    ¿ e ?

    strb r5, [r4]           @;si subiendo, incrementar contador de planta
                           @; y mover máscara a planta superior
                           @;si bajando, decrementar contador de planta
                           @; y mover máscara a planta inferior
                           @;actualizar variable 'planta_actual'

    ldr r2, =RA_Botons
    ldrh r3, [r2]
    ¿ f ?
    beq .LRSIA_final       @;R3 = bits de peticiones pendientes
                           @;comprobar si bit de petición activado
                           @;si planta actual no solicitada, salir de
                           @; la rutina de servicio de interrupción

    ldr r0, =RA_Motor
    mov r1, #0
    strb r1, [r0]           @;parar motor (inmediatamente < 5μs)
    ¿ g ?
    strh r3, [r2]           @;bit de petición de planta actual = 0
                           @;actualizar registro RA_Botons

.LRSIA_final:
    pop {r0-r7, pc}

```

```

NUM_PLANT = 4

@; siguiente_movimiento: calcula cual es el siguiente sentido del
@; movimiento del motor, según las peticiones pendientes (se supone que
@; existe al menos una), la planta actual y el sentido actual del
@; movimiento; el tercer parámetro se pasa por referencia para poder
@; ser modificado directamente en memoria desde la propia rutina; además,
@; el nuevo valor del sentido se devolverá también como resultado de la
@; rutina.

@; Parámetros:
@; R0: peticiones pendientes (unsigned short)
@; R1: valor de planta actual (unsigned char)
@; R2: referencia de la variable que memoriza el sentido actual
@; del movimiento (unsigned char *)
@; Resultado:
@; R0: siguiente sentido del movimiento (1 → subida, 2 → bajada)

siguiente_movimiento:
    push {r3-r5, lr}          @;salvar registros a modificar

    ldrb r3, [r2]             @;R3 = valor de sentido actual
    cmp r3, #1                @;comprobar sentido actual del movimiento
    bne .Lsm_bajada

    .Lsm_subida:
        add r4, r1, #1          @;R4 = número de bits inferiores a planta
        mov r5, r0, lsr r4      @;actual (inclusive)
        cmp r5, #0                @;R5 descarta bits de petición inferiores
        moveq r3, #2              @;comprobar peticiones superiores
        beq .Lsm_actualizar     @;en caso negativo, cambiar sentido
        b .Lsm_final

    .Lsm_bajada:
        beq .Lsm_actualizar     @;si hay detección, finalizar rutina sin
        b .Lsm_final            @;cambio de sentido

    .Lsm_actualizar:
        .i ?                   @;actualizar variable sentido, por referencia

    .Lsm_final:
        mov r0, r3              @;y devolver su valor como resultado
        pop {r3-r5, pc}          @;restaurar registros modificados

```

Propuesta Implementación Problema 30

```

#define MAX_LEVEL      2500          // númer. máximo de niveles
#define MAX_INTV       15            // númer. máximo de intervalos

unsigned short Vlevel[MAX_LEVEL];    // vector de niveles (gráfica)
unsigned short i_level = 0;          // índice actual en Vlevel[]
unsigned short Vintv[MAX_INTV];     // vector de intervalos IBI
unsigned char i_intv = 0;            // índice actual en Vintv[]

unsigned short ibi, temp_ibis = 0;    // intervalo entre latidos
                                      // (último y temporal)
unsigned short nivel_1, nivel_2;    // niveles anteriores de señal
unsigned char nuevo_pulso = 0;        // detección de nuevo pulso

void main()
{
    inicializaciones();
    do
    {
        tareas_independientes();
        swiWaitForVBlank();
        ¿ a ?

        if (nuevo_pulso)           // si nuevo pulso pendiente,
        {
            Vintv[i_intv] = ibi;   // almacenar último IBI
            ¿ b ?                 // siguiente posición
            calcular_coherencia(Vintv, MAX_INTV);
            swiWaitForVBlank();     // escribir valores numéricos
            printf(¿ c ?);
            nuevo_pulso = 0;
        }
    } while (1);
}

```

@;RSI del timer0: se activará 500 veces por segundo, es decir, cada
@; 2 milisegundos; realiza un seguimiento del nivel del sensor,
@; almacenando dichos niveles en el vector global Vlevel[], según
@; el índice i_level (última posición de inserción);
@; detecta picos de señal (máximos locales) por encima del
@; umbral 550, que se supone que se corresponden con los pulsos R
@; (sístoles);
@; cuenta el tiempo entre pulsos (en milisegundos), transmitiendo
@; este valor al programa principal mediante las variables
@; globales ibi (interbeat interval) y nuevo_pulso (=1);

```

RSI_timer0:
    push {r0-r7, lr}

    ¿ d ?
    ldrh r1, [r0]          @; R1 = valor temporal de intervalo
    add r1, #2             @; acumula 2 milisegundos más

    ¿ e ?
    ldrh r2, [r2]           @; R2 = nivel actual de la señal

    ldr r3, =i_levl
    ldrh r4, [r3]           @; R4 = i_levl
    ¿ f ?
    cmp r4, #MAX_LEVEL
    ¿ g ?

    ldr r3, =nivel_1
    ldrh r4, [r3]           @; R4 = nivel_1
    ldr r5, =nivel_2
    ldrh r6, [r5]           @; R6 = nivel_2

    ldr r7, =550
    cmp r2, r7              @; ignorar valores inferiores al umbral
    blo .LRSI_fin

    cmp r6, r4
    bhs .LRSI_fin
    cmp r4, r2
    bls .LRSI_fin
    ldr r7, =ibi
    ¿ h ?

.LRSI_fin:
    ¿ i ?
    strh r2, [r3]           @; nivel_2 = nivel_1
    strh r1, [r0]            @; nivel_1 = nivel actual
                            @; actualiza variable temp_ib
    pop {r0-r7, pc}

```

Propuesta Implementación Problema 36

```

RSI_Timer0:                                @; se activa 1 vez por segundo
    push {r0-r2, lr}
    ldr r0, =ra_objetivo
    ldr r1, [r0]
    add r1, #1                                @; incrementa ra_objetivo
    ldr r2, =86400
    cmp r1, r2
    ¿ d ?                                @; ajuste circular
    str r1, [r0]
    pop {r0-r2, pc}

@; REG_TEL: bit 5: 0 -> giro grueso, 1 -> giro fino
@;           bit 6: 0 -> incrementar, 1 -> decrementar
@;           bit 7: pulso -> una variación de giro
@;           grueso:   1 pulso = 0,5°,      120 segundos de RA
@;           fino:     1 pulso = 0,00417°,  1 segundo de RA (aprox.)
@;           giro de 360° -> 720 variaciones gruesas, 86.400 var. finas
@;           frecuencia máxima de motores: 15 pulsos/segundo
@;           vel. máx. variación gruesa: 1 vuelta/48 segundos
@;           vel. síncrona con rotación de Tierra: 1 variación fina/segundo

```

```

@; RSI del timer1: se activará a la frecuencia requerida para generar el
@;           pulso de variación del motor M1;
@;           si se encuentra en búsqueda, la velocidad será 15 pulsos/segundo
@;           y el tipo de giro dependerá de la distancia entre ra_actual y
@;           ra_objetivo (grueso si >=120, fino si <120);
@;           si ra_actual = ra_objetivo, pasará a seguimiento, con giro fino
@;           y velocidad de 1 pulso/segundo;
@;           también ajusta el sentido de giro según el resultado de la
@;           comparación entre ra_actual y ra_objetivo, y se actualiza
@;           ra_actual según el tipo de giro (grueso/fino) y el sentido de
@;           giro (incremento/decremento).
RSI_Timer1:
    push {r0-r9, lr}

```

¿ e ?

```
orr r3, #0x80                                @; pasa pulso 0 a pulso 1
```

```

ldr r4, =seek_ra
ldrb r5, [r4]
ldr r6, =ra_actual
ldr r7, [r6]
ldr r8, =ra_objetivo
ldr r8, [r8]
mov r9, #1
                                         ; R5 = valor de búsqueda

mov r0, r8
mov r1, r7
¿ f ?
                                         ; R0 = circ(ra_objetivo-ra_actual)
                                         ; R1 = abs(R0)
                                         ; comprobar modo búsqueda

cmp r5, #1
bne .LTim1_cont

cmp r1, #120
¿ g ?
                                         ; si abs(R0) >= 120, giro grueso
movhs r9, #120
orrlo r3, #0x20
                                         ; incremento grueso
                                         ; si no, giro fino

.LTim1_cont:
cmp r0, #0
¿ h ?
                                         ; detectar signo de comparación
                                         ; avanzar ra_actual
                                         ; retroceder ra_actual
                                         ; si obj > act, incrementar
                                         ; si obj < act, decrementar
                                         ; actualizar variable ra_actual
                                         ; si obj != act, saltar al final

bne .LTim1_fin

cmp r5, #0
beq .LTim1_fin
                                         ; si modo seguimiento,
                                         ; saltar al final

ldr r0, =divfreq_vmin
ldsh r1, [r0]
mov r0, #1
¿ i ?
                                         ; si ra_objetivo == ra_actual
                                         ; fijar divfreq_vmin

mov r5, #0
strb r5, [r4]
ldr r4, =track
mov r5, #1
strb r5, [r4]
                                         ; fijar seek_ra = 0
                                         ; fijar track = 1

.LTim1_fin:
strb r3, [r2]
                                         ; actualizar REG_TEL

pop {r0-r9, pc}

```

Propuesta Implementación Problema 38

```

TIMER0_DATA = 0x04000100
TIMER0_CR = 0x04000102

@; rsi_sensor(): activates at every rising pulse of the vintage telephone
@; dialer; to check whether the new pulse belongs to a sequence of
@; pulses of the current digit, or it is the start of a new digit,
@; the routine compares the current counter of the timer 0 and
@; decides if the pulse is the start of a new digit when
@; the counter is above 16364, which corresponds to half a second.
@; The processing of the new digit is carried out by an external
@; routine called capture_digit().
@; Whether the capture_digit() routine is called or not, the
@; current number of pulses is increased and the timer counting is
@; reset.

.global rsi_sensor
rsi_sensor:
    push {r0-r4, lr}           @; save modified regs

    ldr r3, =TIMER0_DATA
    ldrh r1, [r3]              @; R1 is the current counter value

    ldr r2, =num_pulses
    ldrb r0, [r2]              @; R0 is the current number of pulses

    ldr r4, =16364
    cmp r1, r4                @; check if counter is above 50 cs
    e?d?

    add r0, #1
    strb r0, [r2]              @; update number of pulses
                                @; if capture_digit() is called, the
                                @; current number of pulses will be 1
                                @; (0+1) accounting the current pulse
                                @; as the start of the new digit

    ldr r2, =REG_TEL
    e?e?                         @; read previous activations of LEDs
                                @; create mask for the LED corresponding
                                @; to the current number of pulses
                                @; activate the LED bit
                                @; update I/O register

    add r3, #2
    mov r0, #0
    strh r0, [r3]
    mov r0, #0xC3              @; R3 points to TIMER0_CR
                                @; start, enable IRQs, select input '11'
                                @; reset counter

    pop {r0-r4, pc}

```

```

@; capture_digit(n_puls): converts the past number of pulses into a
@; digit value that will be stored in the global vector NTEL[], at the
@; current position ind_digit (global variable). This ind_digit variable
@; will be increased and the global variable new_digit will be set to
@; 1; when number of pulses is ten, the stored digit value is 0.
@; Register R0 (parameter n_puls) always returns zero.

capture_digit:
    push {r1-r3, lr}           @; save modified regs

    cmp r0, #10
    movhs r0, #0               @; convert counts equal (or above) ten

    @; R1 is base address of digits vector

    add r3, #1
    strb r3, [r2]             @; increase ind_digit

    ldr r2, =REG_TEL
    mov r3, #0
    strh r3, [r2]             @; clear I/O register (turn off all LEDs)

    ldr r1, =new_digit
    strb r2, [r1]              @; signal for the main loop

    mov r0, #0                 @; return reset value of number of pulses

    pop {r1-r3, pc}

@; rsi_timer0(): activates when 2 seconds have passed since the last
@; pulse; in this case, the last digit of the telephone number must be
@; recorded, the counter of pulses must be reset and the timer must be
@; stopped.
.global rsi_timer0
rsi_timer0:
    push {r0-r1, lr}           @; save modified regs

    ldr r1, =num_pulses
    ldrb r0, [r1]
    bl capture_digit

    @; mov r0, #0                @; no need to set R0 to 0, because
                                @; capture_digit() always returns R0 = 0

    @; h ?

    ldr r1, =TIMER0_CR
    @; i ?                      @; stop the timer

    pop {r0-r1, pc}

```

Solución Problema 3

Pregunta	Valor	Respuesta
a	1,0	freq = 0;
b	1,0	generar_vibracion(0);
c	1,0	swiWaitForVBlank();
d	0,5	cmp r0, #0
e	2,5	<pre>mov r1, r0 ldr r0, =32728 swi 9 rsb r0, r0, #0</pre>
f	0,5	mov r0, r0, lsr #16
g	0,5	str r0, [r2]
h	0,5	push {r0-r1, lr}
i	2,5	<pre>ldr r0, =REG_RUMBLE ldrb r1, [r0] eor r1, #0x02 strb r1, [r0]</pre>

Solución Problema 14

Pregunta	Valor	Respuesta
a	1,0	REG_TECL = -1;
b	1,0	currentKey != prevKey
c	1,0	processKey(currentKey);
d	0,5	mov r2, #3
e	2,5	tst r0, r3 beq .Ldesc_fincol mov r3, r3, lsl #1 sub r2, #1 cmp r2, #0
f	0,5	add r0, r2, r1, lsl #2
g	0,5	ldsbt r4, [r3]
h	0,5	mov r4, #-1
i	2,5	strb r4, [r3] strb r1, [r2] mov r0, #0x10 mvn r0, r0, lsl r1 strh r0, [r6]

Solución Problema 20

Pregunta	Valor	Respuesta
a	1,0	pasos == 0
b	1,0	fijar_frecuencia(243 * velocidad);
c	1,0	activar_timer0();
d	2,5	<pre>ldr r0, =REG_IF ldr r1, [r0] tst r1, #IRQ_TIMER0 blne RSI_timer0</pre>
e	0,5	ldr r2, =INTR_WAIT_FLAGS
f	0,5	and r1, #7
g	0,5	ldrb r2, [r0, r1]
h	0,5	bic r1, #0xF0
i	2,5	<pre>ldr r0, =pasos ldrh r1, [r0] subs r1, #1 bleq desactivar_timer0 strh r1, [r0]</pre>