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

Autor: Santiago Romaní (santiago.romani@urv.cat)

Índice de contenidos

Problema 1: Reloj de tiempo real

Se propone trabajar con la información que ofrece el reloj en tiempo real con el que está equipada la plataforma NDS.

El programa a realizar define un formato del tiempo específico utilizando un vector de 6 bytes, con las siguientes posiciones:

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

El programa a realizar consta de las siguientes tareas, para las cuales se dispone de rutinas ya implementadas:

Rutina	*Descripción*
`inicializaciones()`	Realiza inicializaciones del *hardware*
`tareas_independientes()`	Tareas que no dependen del tiempo real
`swiWaitForVBlank()`	Espera retroceso vertical
`mostrar_tiempo(char *tiempo)`	escribe en pantalla el tiempo real que se pasa por parámetro
`detectar_alarma(char *t, char *a)`	si el tiempo real 't' coinciden con el tiempo de alarma 'a', se activa un proceso de alarma, la ejecución del cual es del orden de 50 milisegundos

Además, hay que realizar la captura del tiempo real por interrupciones. Como el reloj en tiempo real NO genera interrupciones por el paso de los segundos, 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 *chip select* 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; concretamente, después del comando anterior se podrán recibir hasta 7 bytes con la siguiente información:
	`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 *chip select* del reloj en tiempo real

El protocolo de comunicación es simple:

- 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).

Un último detalle a tener en cuenta es la codificación BCD (Binary Coded Decimal); se trata de números decimales de 2 dígitos codificados dentro de un byte, guardando las unidades en los 4 bits de menos peso y las decenas en los 4 bits de más peso. Por ejemplo, el número en BCD 0x21 (0010 0001) representa 2 decenas y 1 unidad, o sea, el valor decimal 21.

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 la información que ofrece el sensor de movimiento MK6, que se puede conectar a la NDS como tarjeta de expansión GBA.

El programa a realizar consta de las siguientes tareas, para las cuales se dispone de rutinas ya implementadas:

Rutina	*Descripción*
`inicializaciones()`	Realiza inicializaciones del *hardware*
`tareas_independientes()`	Tareas que no dependen de la inclinación
`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:

Posición	*Campo*	*Descripción*
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*
`iniciar_MK6()`	Activa el *chip select* del sensor de movimiento

`enviar_MK6(byte comando)`	Envía un comando al sensor de movimiento; concretamente tendremos que enviar 2 comandos:
	`Read_X: 0x00`
	`Read_Y: 0x02`

`short recibir_MK6()`	Recibe un short (16 bits) del sensor de movimiento, con el valor de aceleración del eje indicado con el comando anterior
`finalizar_MK6()`	Desactiva el *chip select* del sensor de movimiento

El protocolo de comunicación es simple:

- 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, por lo tanto, 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 para el eje X:

$$Inclin\_X = Inclin\_X + (Acel\_X - xoff) * xsens;$$

donde Inclin_X es el valor de inclinación (acumulada) y Acel_X es el valor de

aceleración (instantánea) 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 *inc, short acel, short off, short sens)` donde `inc` (inclinación) se pasa por referencia y el resto de 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:

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

Por otro lado, hay que implementar una rutina que se llamará `generar_vibracion(short frec)`, 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 es muy sencillo: se trata de modificar el bit 1 del registro REG_RUMBLE. Cada vez que este bit cambia de estado se produce un movimiento del dispositivo vibrador. Por lo tanto, para generar vibración a una determinada frecuencia hay que cambiar el estado del bit 1 a la frecuencia requerida.

El funcionamiento del programa principal tiene que ser el siguiente:

- inicializaciones
- bucle principal
- 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
- fin de bucle principal

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

0400 0100	TIMER0_DATA	Valor del contador / carga de divisor de frecuencia
0400 0102	TIMER0_CR	Registro de control del Timer 0

Donde TIMER0_DATA se utilizará para cargar el divisor de frecuencia y TIMER0_CR se utilizará para iniciar y parar la generación de interrupciones periódicas, con los siguientes parámetros:

Característica	*Bits*	*Valor*	*Descripción*
Prescaler Selection	1..0	11	frecuencia de entrada aprox. 32728 Khz
Count-up Timing	2	0	No
Timer IRQ Enable	6	1	Sí
Timer Start/Stop	7	1	Start

Para parar 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:

Div_Frecuencia = -(Freq_Entrada / 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 la 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 0x09FFFFFE, el cual proporciona un bit para cada una de las tecla 13 teclas del piano:

Bit	*Campo*	*Nota*	Código de nota
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

Además se dispone de las siguientes rutinas ya implementadas:

Rutina	*Descripción*
`inicializaciones()`	Realiza inicializaciones del *hardware*
`tareas_independientes()`	Tareas que no dependen del piano
`swiWaitForVBlank()`	Espera retroceso vertical
`generar_nota(char codigo, short volumen)`	Inicia la reproducción de una nota de piano según el código de nota y el volumen inicial
`modular_volumen(char codigo, short volumen)`	Cambia el volumen de una nota.

Los códigos de nota son los mostrados en la primera tabla. Los valores de volumen oscilan entre 16 y 0, donde 16 es el volumen máximo, 1 es el mínimo y 0 indica 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.

La tareas que tiene que controlar el programa son:
- 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.

Se debe utilizar una estructura de información para cada nota, 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 este cálculo para 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 frec, short vol)`	Activa la reproducción de una nota por un canal de sonido, a una frecuencia (en Hz) a un determinado volumen (127..0) durante un tiempo indefinido
`RSI_timer0()`	Rutina de servicio de interrupciones del *timer* 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 (0400 04X0) y al registro de *timer* (0400 04X4) del canal especificado, donde X es 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 frec: Timer value = -(33513982/2) / frec

Por otro lado, se dispone de las siguientes rutinas ya implementadas:

Rutina	*Descripción*
`inicializaciones()`	Realiza inicializaciones del *hardware*
`tareas_independientes()`	Tareas independientes del sonido
`swiWaitForVBlank()`	Espera retroceso vertical

`printf(const char * format,...)`	Imprime por pantalla un mensaje de texto con formato

Se debe utilizar una estructura de información para cada nota, con los siguientes campos:

```
typedef struct {
     short freq;    // frecuencia de la nota (Hz)
     short time;    // tiempo de la nota (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
- tareas independientes
- sincronización de pantalla
- escribir el índice de la nota actual (sólo cuando haya un cambio de nota)
- fin de bucle principal

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 consta de las siguientes tareas, para las cuales se dispone de rutinas ya implementadas:

Rutina	*Descripción*
`inicializaciones()`	Realiza inicializaciones del *hardware*
`tareas_independientes()`	Tareas que no dependen de la información del audio
`swiWaitForVBlank()`	Espera retroceso vertical
`mostrar_frecuencias(char *audio)`	Dibuja un ecualizador en una pantalla de la NDS

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 *hardware* 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 *hardware* del micrófono

El protocolo de comunicación es simple:

- iniciar el micrófono, una vez al principio de la captura
- recibir y almacenar los bytes necesarios
- parar el micrófono, al final de la captura

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 captura 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*.

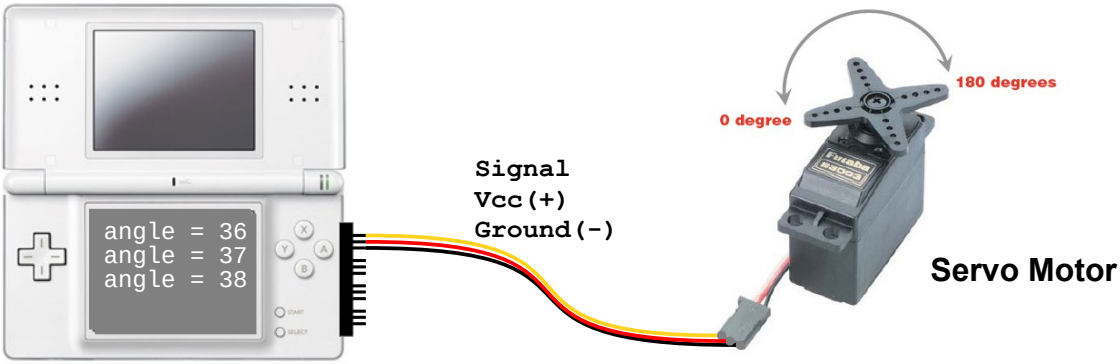
Habrá que crear una rutina de nombre `cambiar_buffers()`, que cambiará el buffer actual de captura.

Se pide:

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

Problema 7: Servomotor

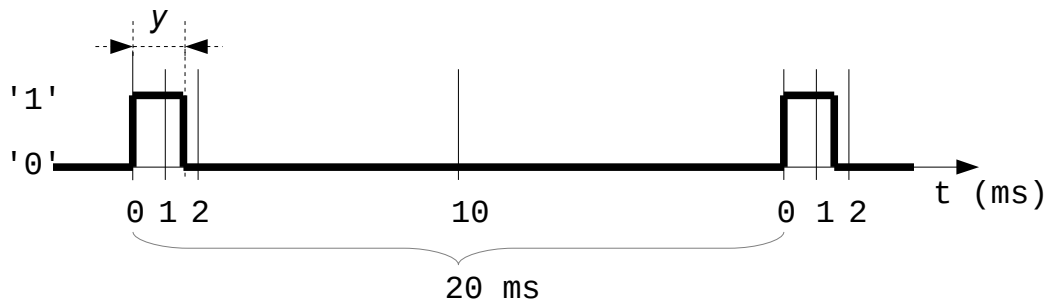
Se propone controlar el ángulo de giro de un servomotor con la NDS. El programa a realizar permitirá al usuario seleccionar un valor del ángulo entre 0º y 180º:



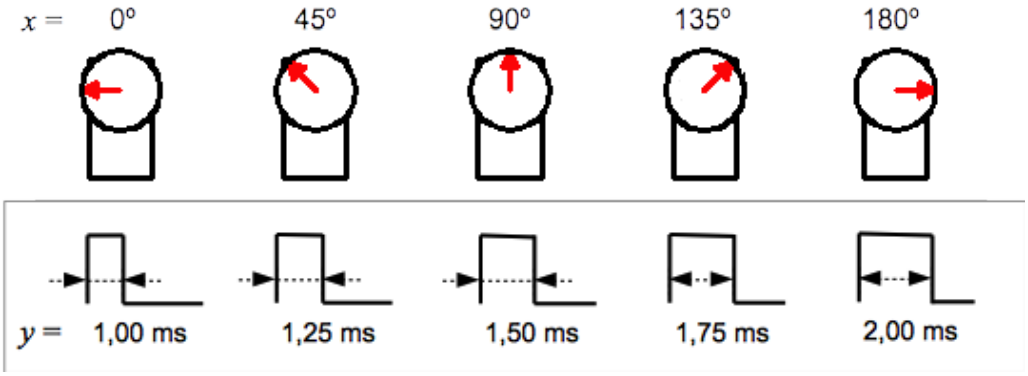
En el esquema se han representado los pines de un adaptador de Entrada/Salida que permite a la NDS controlar hasta 4 servomotores simultáneamente. El controlador de E/S del adaptador se gestiona a través de un registro de E/S denominado REG_SERVO, de 8 bits, de los cuales solo se utilizan los 4 bits 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) i masa (negro) correspondientes. Supongamos que el cable de señal del servomotor del esquema 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 0º y 2 ms para 180º, 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 (0º, 45º, 90º, 135º, 180º), aunque se puede conseguir prácticamente cualquier valor (entero) de grados. En general, la siguiente fórmula permite calcular el número de milisegundos del tiempo a '1' (*y*) a partir del valor del ángulo en grados sexagesimales (*x*):

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

Como en lenguaje máquina no podemos trabajar con valores reales, podemos adaptar la fórmula para expresar el tiempo *y* 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 / 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 / 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.

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

Rutina	*Descripción*
`inicializaciones()`	Inicializa el *hardware* (pantalla, interrupciones, etc.)

`tareas_independientes()`	Tareas que no dependen del acceso al servomotor conectado al bit 3; supondremos que nunca tardarán más de 100 milisegundos en ejecutarse
`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 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 en '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 se tendrán que alternar los periodos para el estado '1' y el estado '0' del pulso, es imprescindible detener el *timer* antes de fijar el un nuevo divisor de frecuencia, y luego volver a activarlo, configurando de nuevo el registro de control. Para la frecuencia de entrada, se sugiere utilizar F/64.

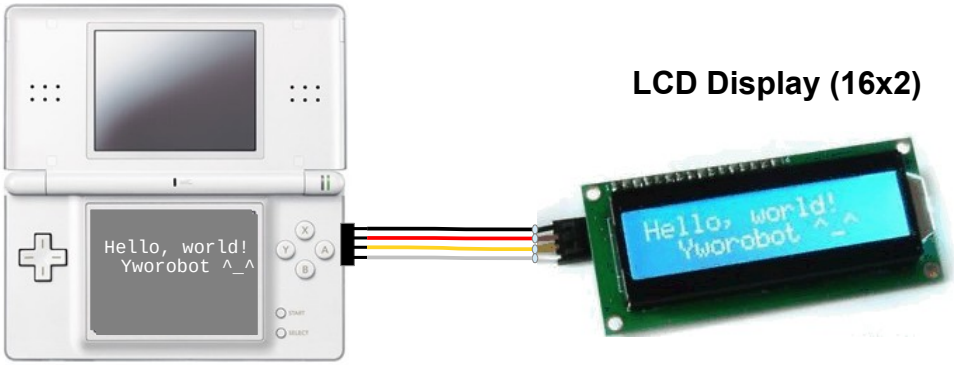
Para realizar las divisiones desde lenguaje ensamblador, se puede utilizar la rutina BIOS `swi 9` (entrada: R0 = numerador, R1 = divisor; salida: R0 = cociente, R1 = resto, R3 = cociente sin signo).

Se pide:

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

Problema 8: Display LCD

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



En el esquema 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:

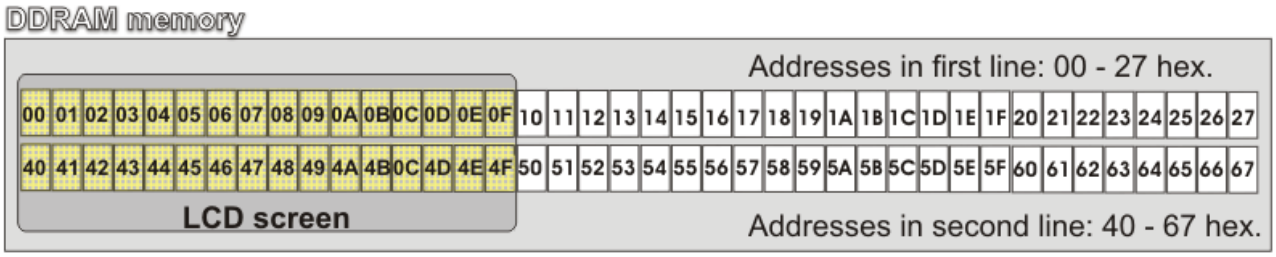
Instruction	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Clear display	0	0	0	0	0	0	0	0	0	1
Cursor home	0	0	0	0	0	0	0	0	1	x
Entry mode set	0	0	0	0	0	0	0	1	I/D	S
Display on/off control	0	0	0	0	0	0	1	D	C	B
Cursor/display shift	0	0	0	0	0	1	S/C	R/L	x	x
Function set	0	0	0	0	1	DL	N	x	BR1	BR0
CGRAM address set	0	0	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": REG_DISPLAY = "000001(S/C)(R/L)xx"
  - S/C: '1' → screen, '0' → cursor
  - R/L: '1' → right, '0' → left
- "DDRAM Address set": REG_DISPLAY = "001(DDRAM address)"
  - DDRAM address (7 bits)
- "DDRAM or CGRAM write": REG_DISPLAY = "10(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.

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

Rutina	*Descripción*
`inicializaciones()`	Inicializa el *hardware* (pantalla, interrupciones, etc.)
`tareas_independientes()`	Tareas que no dependen del acceso al display LCD; nunca tardarán más de 200 milisegundos en ejecutarse
`int wifiReceiveText(char *t)`	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 µs
`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 de la NDS

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 lineas 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, com mínimo, desde los cuales se copiaran los códigos ASCII de cada carácter sobre las 32 primeras columnas de la DDRAM.

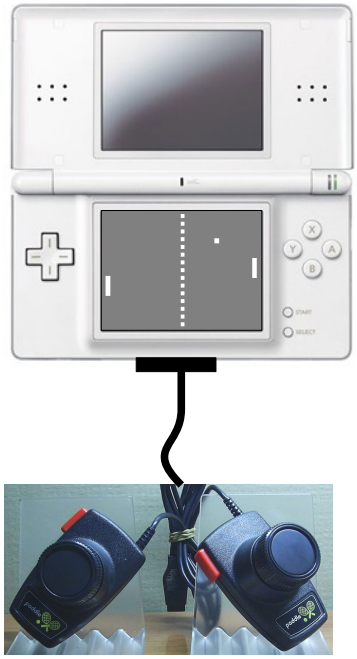
Para sincronizarse con el display, se debe invocar a la rutina `sincro_display()` antes de enviar cualquier comando al display. Esta rutina no retornará hasta que el display no esté preparado, pero se nos asegura que, como máximo, tardará 41 microsegundos en retornar.

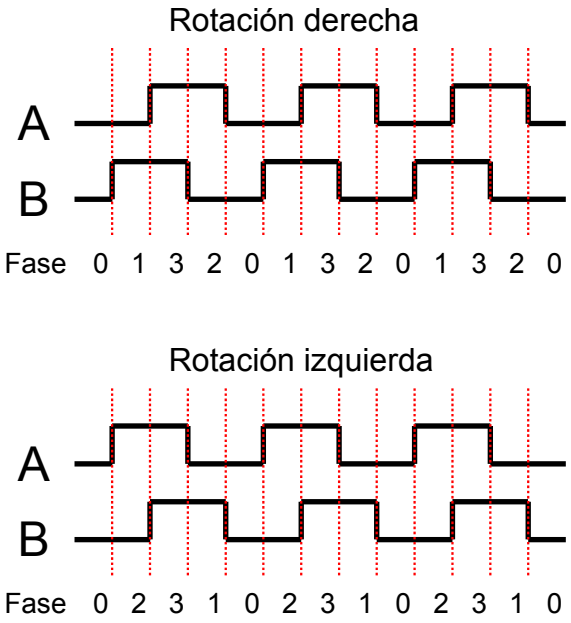
Se pide:

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

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

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 un píxel 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 paddle 1
1	PADDLE1_B	Bit B del paddle 1
2	PADDLE1_A	Bit A del paddle 1
3	PADDLE2_C	Botón del paddle 2
4	PADDLE2_B	Bit B del paddle 2
5	PADDLE2_A	Bit A del paddle 2

Además, se dispone de las siguientes rutinas ya implementadas:

Rutina	*Descripción*
`inicializaciones()`	Realiza inicializaciones del *hardware*
`tareas_independientes()`	Tareas que no dependen del movimiento de los *paddles*
`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 cambio en la fase de cada *paddle*. En caso afirmativo tiene que llamar a una rutina que se llamará `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):

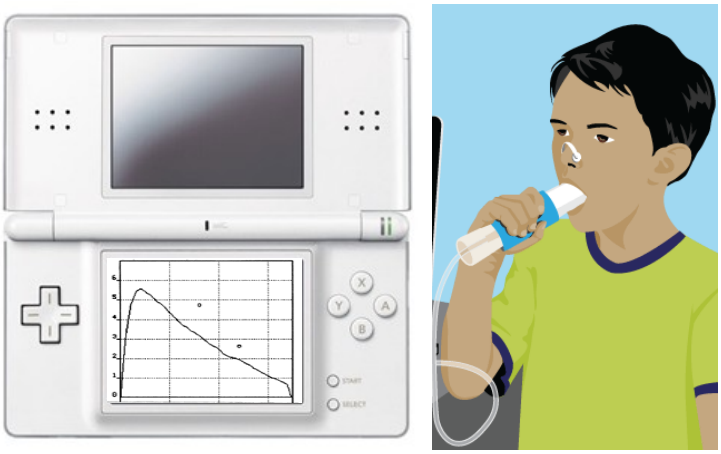
```
char s_derecha[] = {1, 3, 0, 2};
char s_izquierda[] = {2, 0, 3, 1};
```

Se pide:

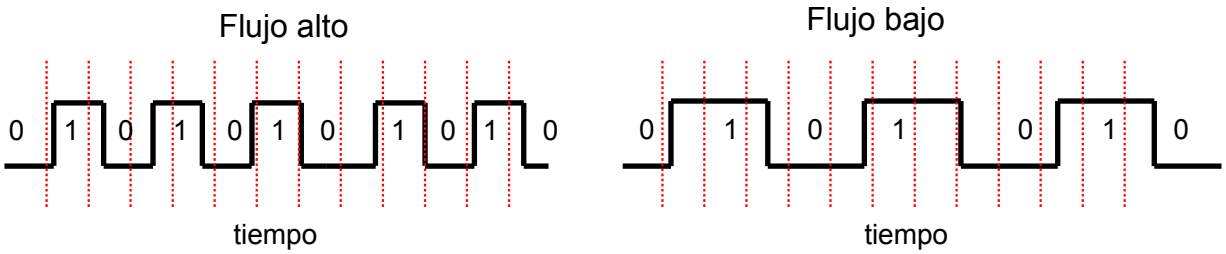
Programa principal en C, RSI del *timer* 0 y rutina `detectar_sentido()` en ensamblador.

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

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 (0/1/0/1/0/1/...) 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 detectará por el bit 0 del registro de E/S 0x0A000180 (reg. de 8 bits). El programa utilizará las interrupciones del *timer* 0 para consultar periódicamente este bit. 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 linea 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);
```

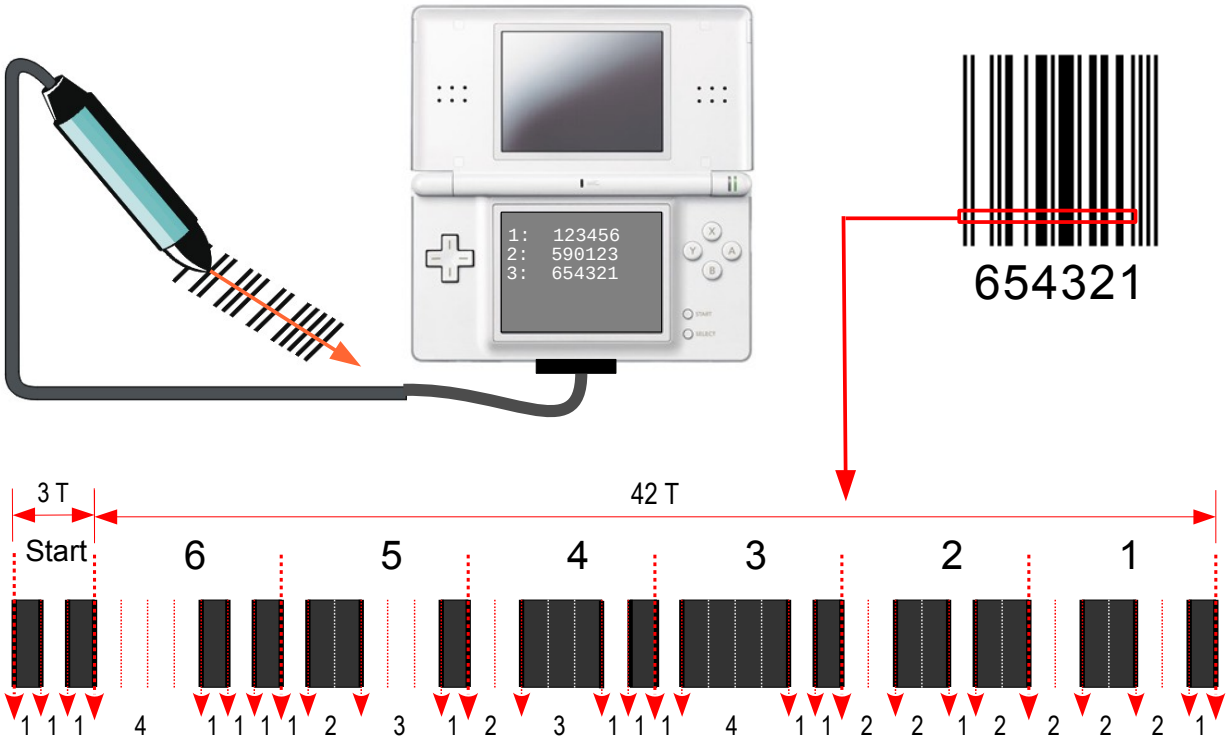
A la llamada de la rutina, los valores de entrada (tiempo, flujo) se deben poner en R0 y R1. Al retorno de la llamada, los mismos registros contendrán las coordenadas en píxeles (px, py). Hay que tener en cuenta que el valor de tiempo se debe desplazar 20 píxeles a la derecha para ajustarse a la gráfica, 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: Lector de códigos de barras (Ex. 1ª Conv. 2012-13)

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 cada vez que el haz de luz del lápiz 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 de 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 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, 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 los códigos de barras
`swiWaitForVBlank()`	Espera el retroceso vertical
`cpuStartTiming(int timer)`	Inicia un cronómetro de precisión usando el *timer* que se pasa por parámetro (0-2) y el siguiente *timer*
`cpuGetTiming()`	Devuelve el contaje 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 (≈ 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 40 μs. 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[]);
```

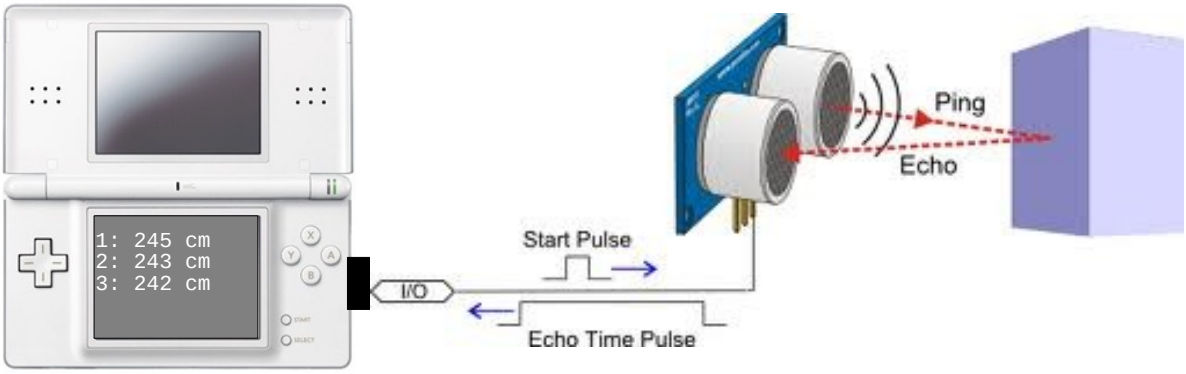
Después se tendrán que decodificar los tiempos relativos obtenidos y escribir los dígitos correspondientes invocando a la rutina `decodificar_codigo()`.

Se pide:

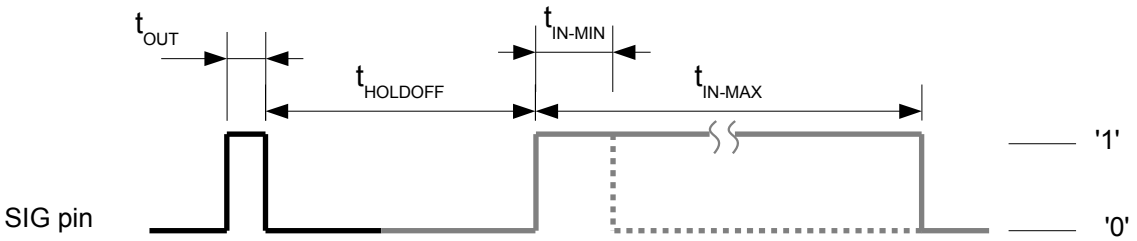
Programa principal en C, RSI del lector y rutina `normalizar_tiempos()` en ensamblador.

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

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



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	Start Pulse	t_{OUT}	2 µs (min), 5 µs typical
——	PING)))	Echo Holdoff	$t_{HOLDOFF}$	750 µs
		Echo Time Pulse Minimum	t_{IN-MIN}	115 µs
		Echo Time Pulse Maximum	t_{IN-MAX}	18.5 ms
		Delay before next measurement		200 µs (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). Para realizar este programa se dispone de las siguientes rutinas, ya implementadas:

Rutina	*Descripción*
`inicializaciones()`	Inicializa el *Hardware* (pantalla, interrupciones, etc.)
`tareas_independientes()`	Tareas que no dependen del cálculo de distancias (siempre tardarán menos de un segundo)
`swiWaitForVBlank()`	Espera hasta el próximo retroceso vertical
`cpuStartTiming(int timer)`	Inicia un cronómetro de precisión usando el *timer* que se pasa por parámetro (0, 1 o 2) y el siguiente *timer*
`cpuGetTiming()`	Devuelve el contaje de tics del cronómetro desde que se inició (entero de 32 bits)
`printf(char *format, ...)`	Escribe por pantalla la información especificada
`startPulse()`	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 (≈ 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	≈	168 tics;
$t_{HOLDOFF}$:	750 µs	≈	25.135 tics;
t_{IN-MIN}:	115 µs	≈	3.854 tics;
t_{IN-MAX}:	18,5 ms	≈	620.009 tics;
`retardo mínimo entre mediciones`:	200 µs	≈	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);
```

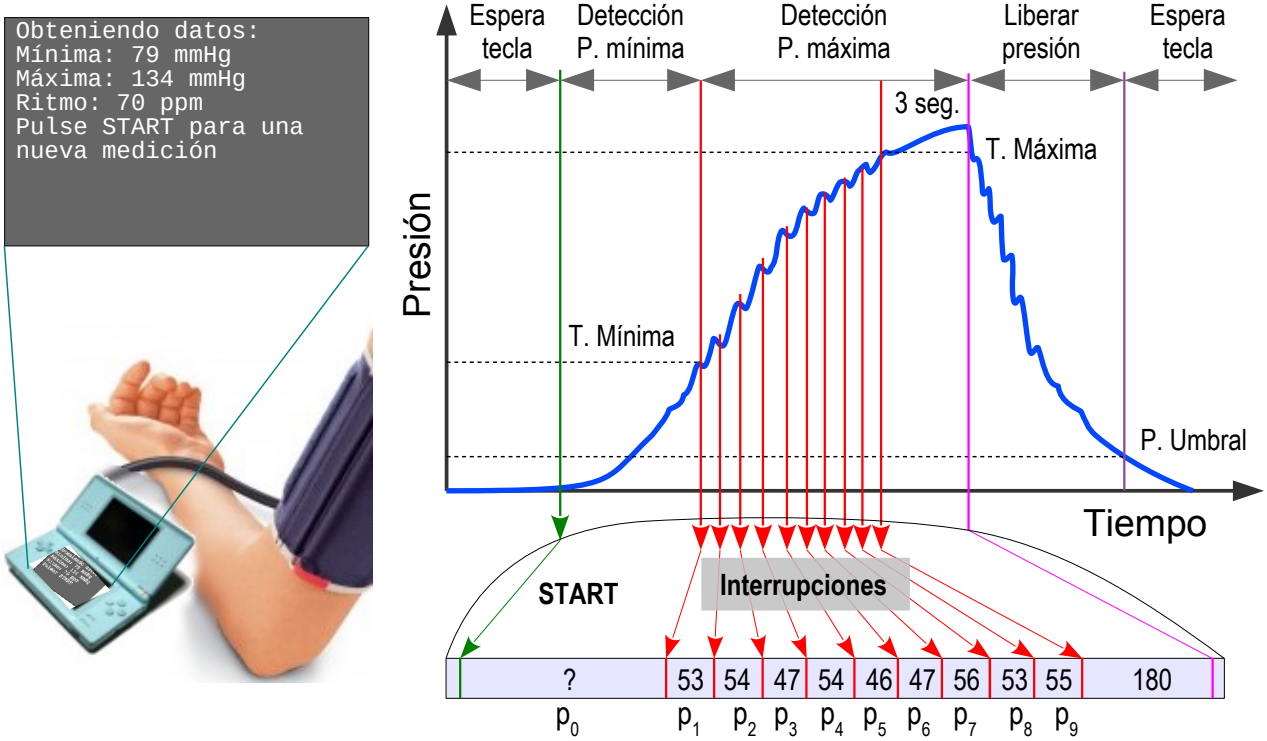
Para realizar las divisiones que sean necesarias se tendrá que utilizar la rutina BIOS `swi 9` (entrada: R0 = numerador, R1 = divisor; salida: R0 = cociente, R1 = resto, R3 = cociente absoluto). Cada división puede tardar entre 5 y 40 microsegundos.

Se pide:

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

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

Tenemos un dispositivo NDS 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 una 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 (sístole + diástole); la tensión mínima será la presión del brazalete al detectar el primer latido completo.
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 periodos (p_0, p_1, p_2, … , p_{n-1} , $n < 50$) para después poder calcular con precisión 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 y 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 completo (sístole + diástole) genera una interrupción específica (IRQ_CART).

Se dispone de las siguientes rutinas ya implementadas:

Rutina	*Descripción*
`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 el retrazado 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 pueden haber pequeñas variaciones de tiempo entre los latidos detectados, se pide que se almacenen dichos periodos en un vector para posteriormente obtener un período promedio.

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 periodo promedio serán 180 y 18 retrocesos verticales, respectivamente.

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

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

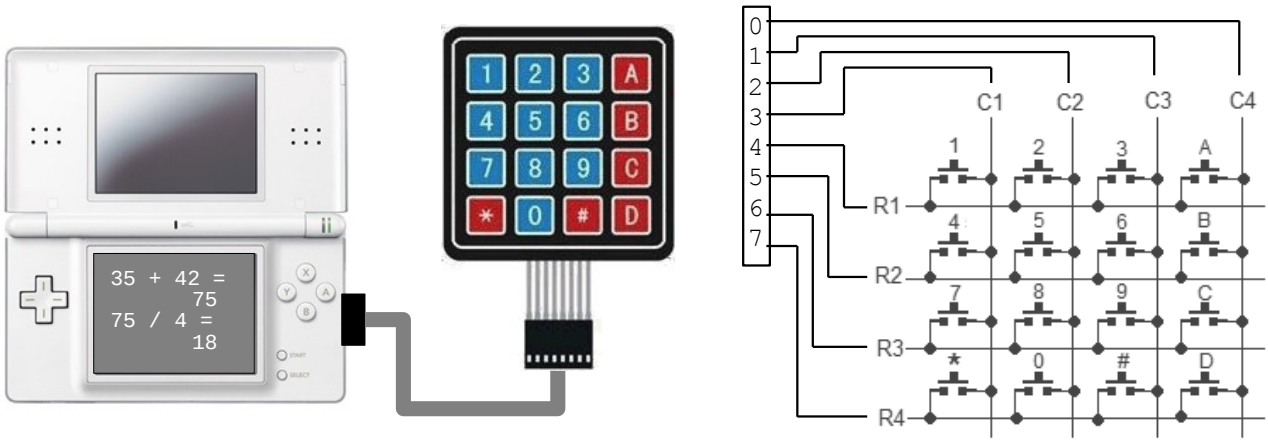
donde el primer parámetro corresponderá al vector de todos los períodos obtenidos (paso por referencia) y el segundo parámetro será el número de elementos registrados en cada obtención de datos, y el resultado que devolverá la rutina será el ritmo cardíaco expresado en pulsaciones por minuto. Para realizar divisiones hay que utilizar 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: Teclado numérico (Ex. 1ª Conv. 2014-15)

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, además de realizar ciertas tareas independientes, 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.

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

Rutina	*Descripción*
`inicializaciones()`	Inicializa el *Hardware* (pantalla, interrupciones, *timer*)
`tareas_independientes()`	Tareas que no dependen del teclado ni del cálculo (siempre tardarán menos de una centésima de segundo)
`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

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 ("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:

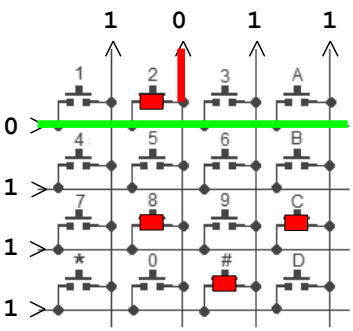
```
char descodificar_tecla(char regteclval, char num_fila);
```

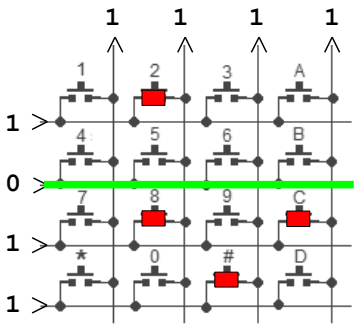
Por su parte, el programa principal debe sincronizarse con la RSI a través de la variable global `currentKey`, por el método de encuesta desde interrupción periódica, es decir, que se deben poder ejecutar las tareas independientes mientras se procesan las teclas. Además, para evitar que una misma pulsación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

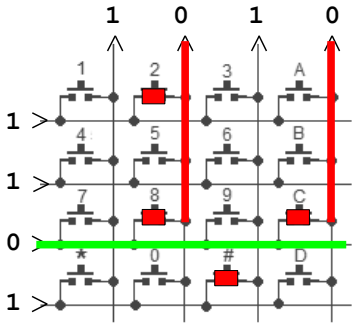
Por último, hay que tener en cuenta que la variable `currentKey` solo se actualizará al final del barrido de todo el teclado, puesto que hay que

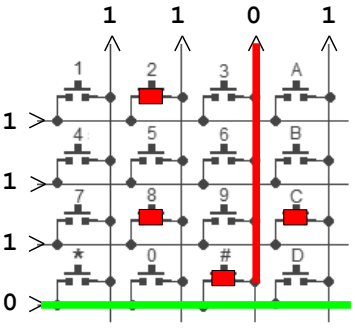
contrastar la prioridad de las teclas de las distintas filas, para lo cual necesitaremos otra variable global de nombre `tempKey` para mantener el código de tecla mayor de todo el barrido, además de la variable global `num_fila` que indicará la fila actual de procesamiento.

A continuación se muestra un ejemplo de barrido del teclado suponiendo que se han pulsado 4 teclas a la vez. Aunque es un caso improbable, sirve para mostrar los valores que se obtendrán en cada fila y cómo se debe actualizar la variable `tempKey`:







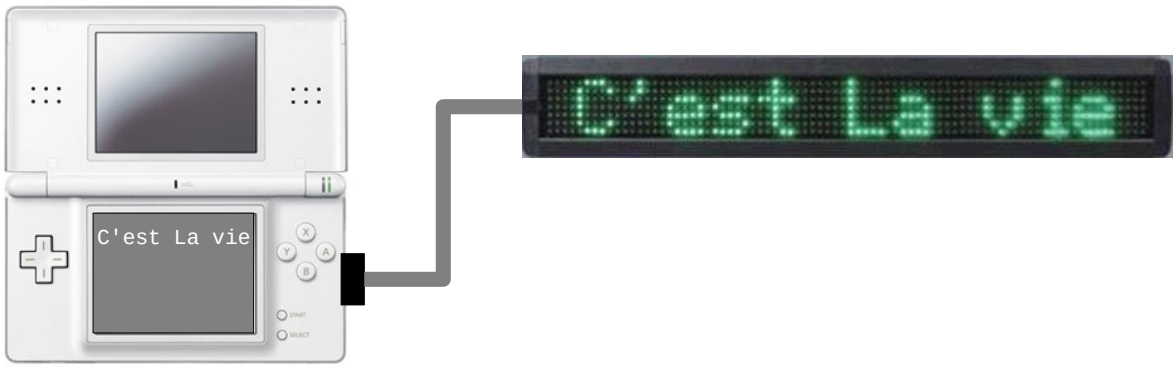


Se pide:

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

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

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 (píxel a píxel):



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 estado (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 *scroll* horizontal.

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

Rutina	*Descripción*
`inicializaciones()`	Inicializa el *Hardware* (pantalla, interrupciones, *timer*)
`tareas_independientes()`	Realiza tareas independientes a la visualización del string, que tardan menos de 1 centésima de segundo.
`swiWaitForVBlank()`	Espera hasta el próximo retroceso vertical
`printf(char *format,...)`	Escribe un mensaje por la pantalla inferior de la NDS

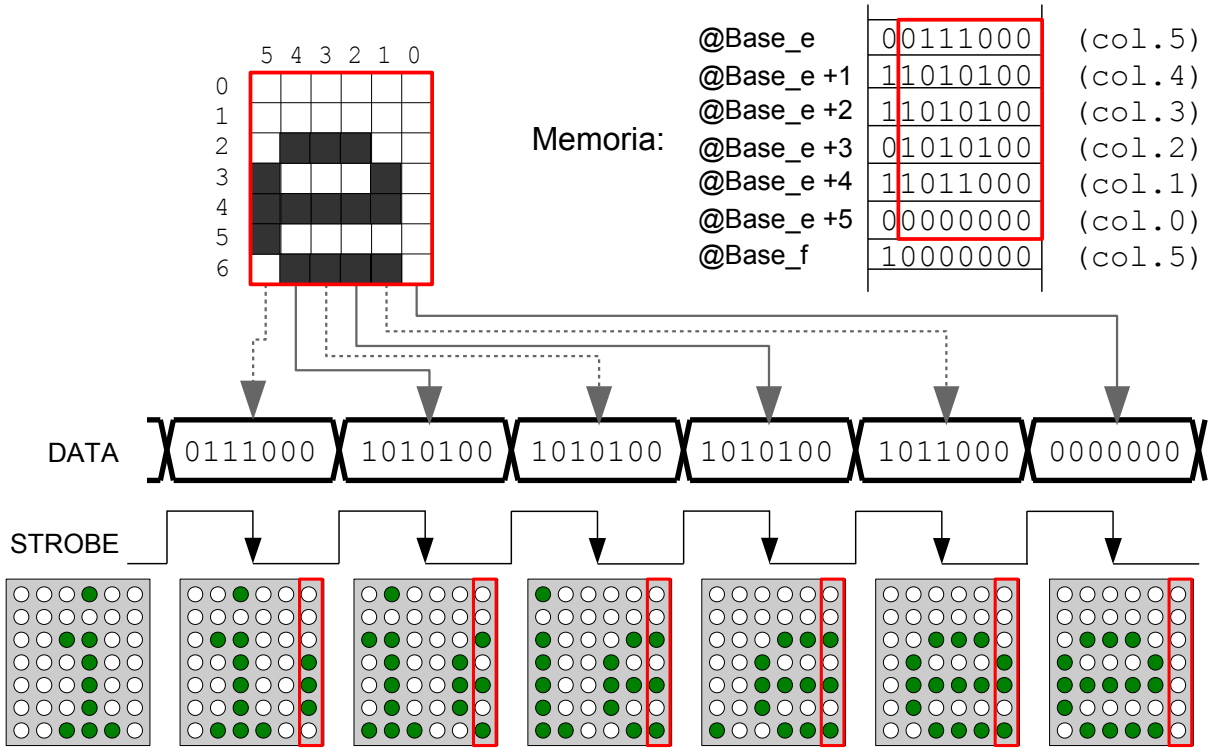
Para poder transferir las columnas de puntos de cada carácter de forma concurrente con el programa principal, se debe utilizar la RSI del *timer* 0, que se programará (por la rutina de inicializaciones) para realizar 36 interrupciones por segundo.

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):



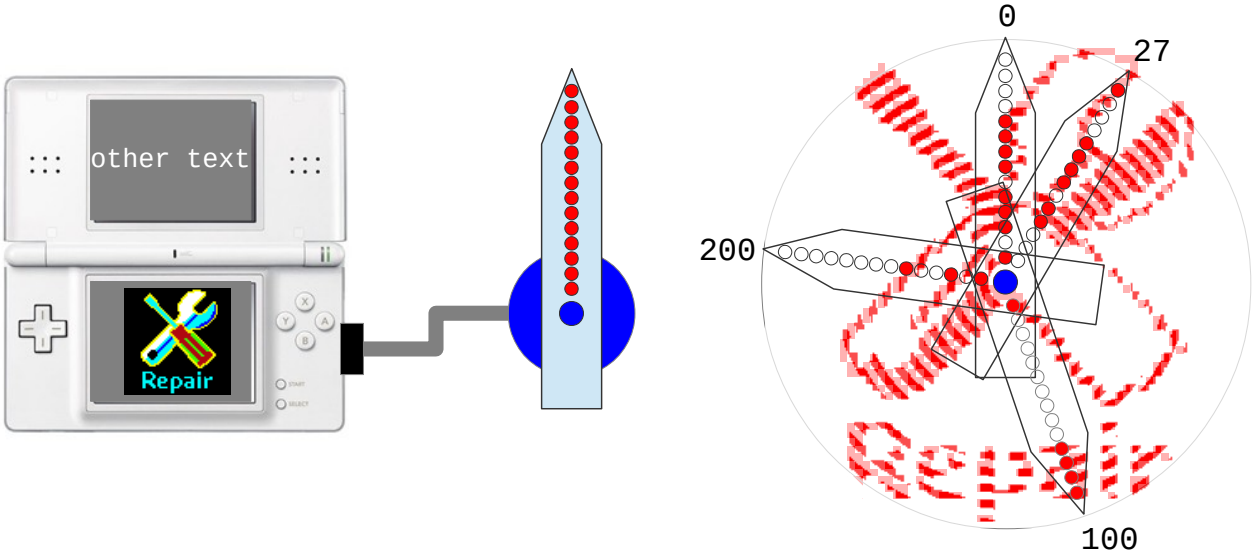
Hay que observar que cada carácter se almacenará a partir de una posición de memoria específica y ocupará 6 bytes consecutivos correspondientes al estado de sus 6 columnas, ocupando dicho estado los 7 bits de menor peso de cada byte, mientras que el bit de más peso del byte puede tener cualquier valor. 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: Propeller display (Ex. 1ª Conv. 2015-16)

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 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 para la imagen de ejemplo que 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 µs 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, además de realizar ciertas tareas independientes, debe recibir imágenes "rectangulares" (64x64 píxeles) desde la 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. Para realizar este programa se dispone de las siguientes rutinas, ya implementadas:

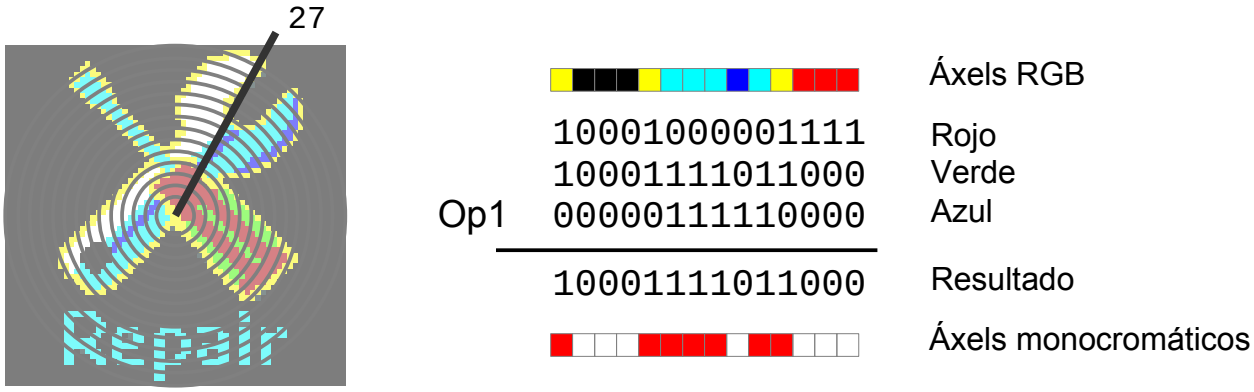
Rutina	*Descripción*
`inicializaciones()`	Inicializa el *hardware* (pantalla, wifi, interrupciones, *timers*, etc.)
`tareas_independientes()`	Tareas que no dependen del display de rotación, aunque sí envían información a la pantalla superior de la NDS (siempre tardarán menos de 5 ms)
`int wifiReceiveImage(unsigned char rimg[])`	Rutina de recepción de una imagen rectangular por la 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; si no hay nueva imagen, devolverá 0
`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 áxels 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 rutina de conversión de los bits de color a bits monocromáticos será la siguiente:

```
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 (± 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 imagen circular 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.

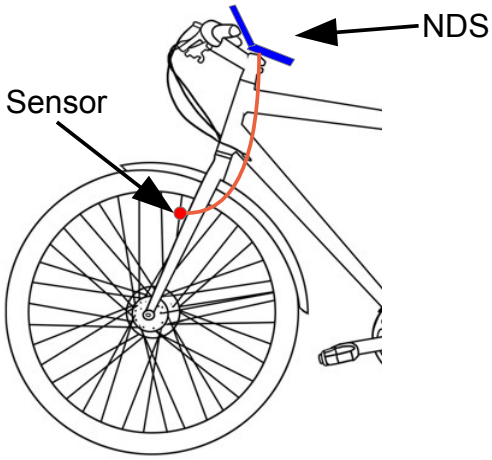
Por último, se puede considerar que, inicialmente, el contenido de los buffers de la imágenes está a cero, de modo que la visualización en el display de rotación de dicho contenido no activará ningún LED, aunque el motor girará a partir del primer instante que se encienda todo el sistema (NDS y dispositivo).

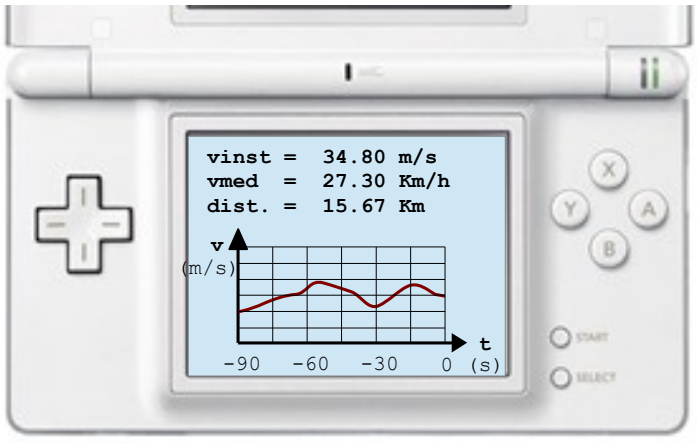
Se pide:

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

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

Se propone realizar un programa para calcular la velocidad instantánea, la velocidad media y la distancia recorrida por una bicicleta, a partir del 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.

Para realizar este programa 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 *hardware* (pantalla, interrupciones, *timers*, etc.) y las variables globales
`tareas_independientes()`	Tareas que no dependen de los cálculos de velocidad, distancia, etc., y que pueden tardar entre una décima de segundo y un segundo
`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 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, al pulsar 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.

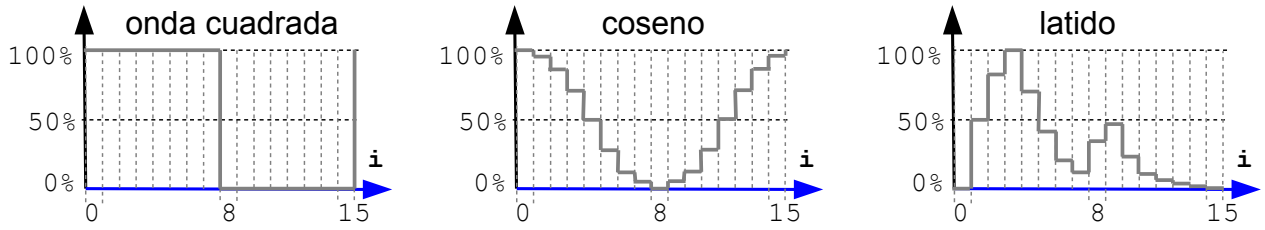
Toda esta información se debe pasar a la función `representarInfo()`, con las unidades indicadas para las variables globales. Internamente, esta función transformará dichas unidades a las habituales (m/s, Km/h, Km). Hay que observar que las variables globales propuestas utilizan fracciones de dichas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

Se pide:

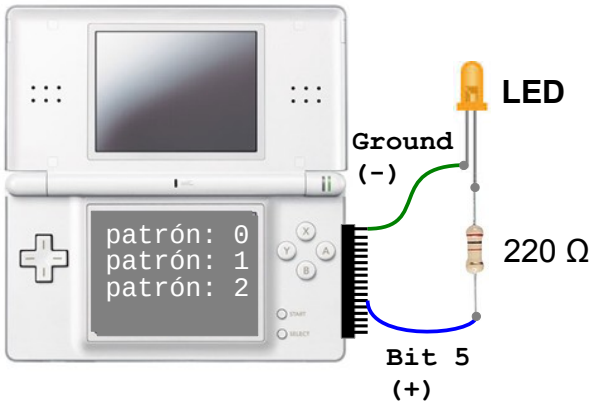
Programa principal y variables globales en C, RSI del *timer* 0 y RSI del sensor en ensamblador.

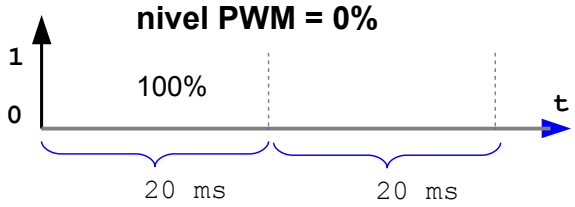
Problema 18: Luz LED regulada por PWM (Ex. 2ª Conv. 2015-16)

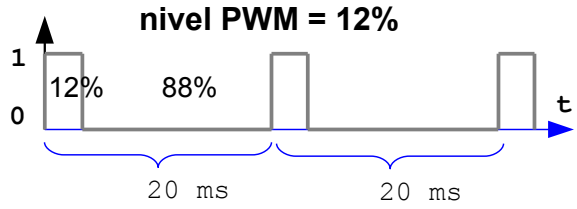
Se propone controlar la intensidad (o brillo) de una luz LED con la NDS, para que emita, repetidamente, una determinada secuencia de niveles del brillo de dicho LED. El programa a realizar permitirá emitir diferentes secuencias, que llamaremos "patrones". A continuación se muestran tres patrones diferentes,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

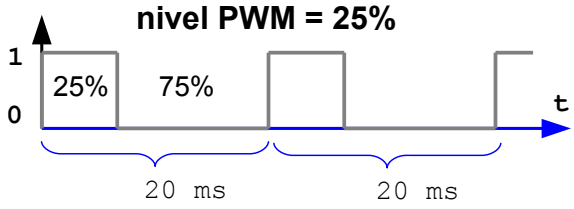


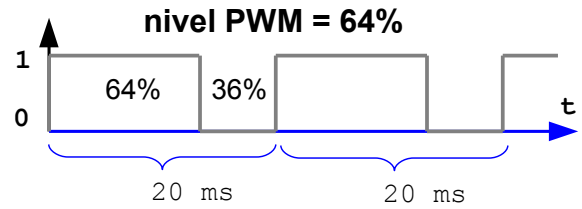
Los niveles de brillo se codificarán como el tanto por ciento de la intensidad máxima que emite el LED. A continuación se muestra un esquema de la estructura del sistema, junto con unos gráficos cuyo significado se describirá más adelante:

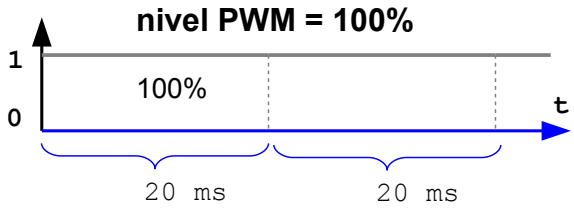












Conectados a la NDS del esquema, se han representado los pins de salida de un dispositivo que permite generar 16 voltajes digitales, es decir, 0 o 5 voltios en cada pin, más un pin extra de masa (*Ground*). El voltaje de los pins se controlará con el estado de un registro de salida de 16 bits, etiquetado 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 pins (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 con 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 estado '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 energía.

El programa a implementar, además de controlar el ancho de pulso correspondiente al valor de brillo actual del patrón seleccionado, tendrá que actualizar dicho valor 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. Para realizar este programa se dispone de las siguientes rutinas, ya implementadas:

Rutina	*Descripción*
`inicializaciones()`	Inicializa el *hardware* (pantalla, interrupciones, *timers*, etc.)
`tareas_independientes()`	Tareas que no dependen del estado del bit 5 del registro de salida, aunque pueden estar modificando algunos de los otros bits del mismo registro (< 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 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] = {  //patrones
     {100,100,100,100,100,100,100,100,0,0,0,0,0,0,0,0},    //cuadrado
     {100,88,75,63,50,38,25,13,0,13,25,38,50,63,75,88},    //triangular
     {0,6,13,19,25,31,37,44,50,56,63,69,75,81,88,94},      //rampa
     {100,96,85,69,50,31,15,4,0,4,15,31,50,69,85,96},      //coseno
     {0,50,80,100,70,40,21,15,37,48,25,13,8,4,2,1}         //latido
};
unsigned char curr_pattern = 0;       // índice del patrón actual
unsigned char curr_value = 0;         // índice del valor actual del pat.
unsigned char brightness = 0;         // valor 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 indice del valor 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 valor 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 lineas:

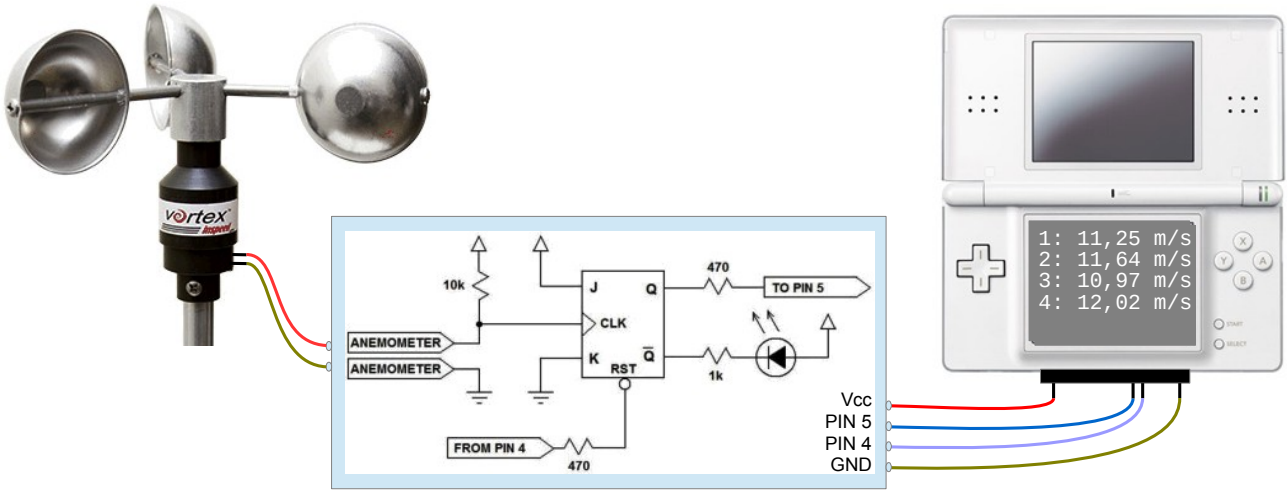
```
NUM_PATTERNS = 5      @; número de patrones de brillo
NUM_VALUES = 16       @; número valores de brillo por patrón
```

Se pide:

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

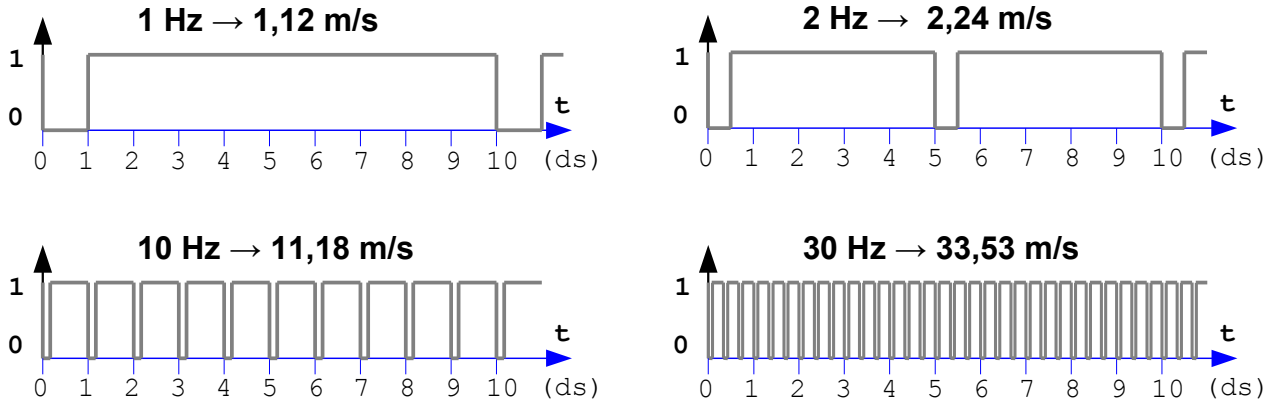
Problema 19: Anemómetro electrónico (Ex. 1ª Conv. 2016-17)

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 0x0A000000. 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 (≈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 m/s ≈ 200 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 el 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, el 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). Para realizar este programa 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 *hardware* (pantalla, etc.)

`tareas_independientes()`	Tareas que no dependen del anemómetro (< 1 segundo)
`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 frec);
```

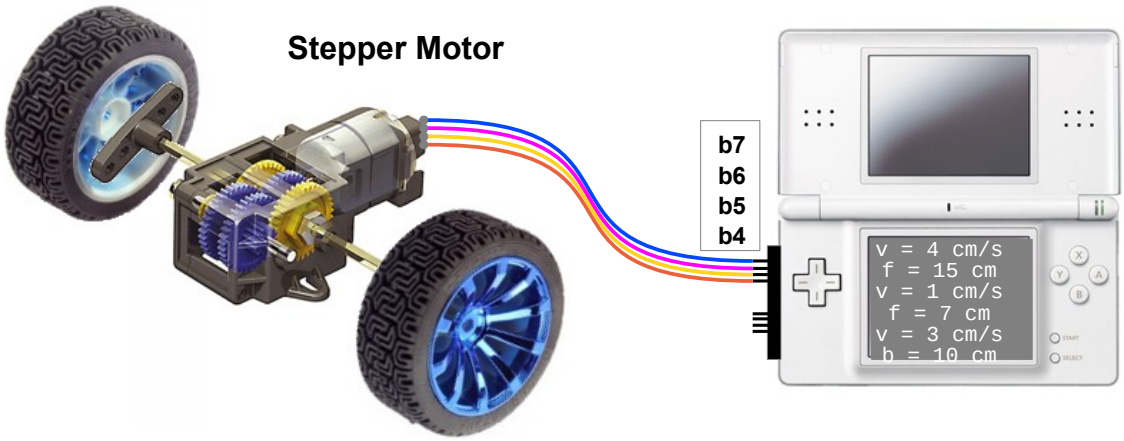
la cual recibe por parámetro la frecuencia de salida del *timer* 0. Esta rutina debe calcular el divisor de frecuencia asociado a la frecuencia de salida requerida, utilizando la segunda frecuencia de entrada más alta disponible (523.656,96875 Hz). Recordar que las direcciones de memoria de los registros 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.

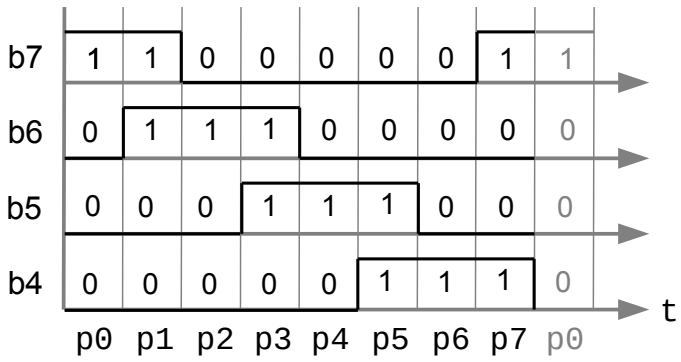
Problema 20: Motor de tracción (Ex. 1ª Conv. 2016-17)

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:

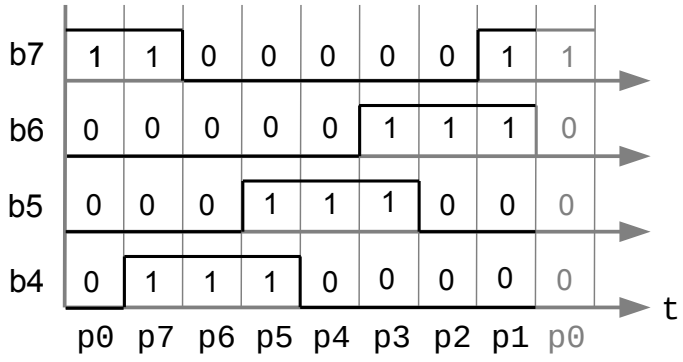


En el esquema anterior se ha representado la conexión de los 4 cables de control del motor paso a paso (*Stepper motor*) de tracción, que están asociados a los bits b7-b4 de un registro de E/S de 32 bits, al cual se accederá a través de la dirección de memoria 0x0A000000. El resto de bits del mismo registro 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 'p7' es la 'p0' y la fase inferior a la 'p0' es la 'p7'. 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 (5,625º/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 (± 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 un 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). Para realizar este programa se dispone de las siguientes rutinas, ya implementadas:

Rutina	*Descripción*
`inicializaciones()`	Inicializa *hardware* (pantalla, interrupciones, timers, etc.)
`tareas_independientes()`	Tareas que no dependen directamente del control del motor de tracción; cada ejecución puede tardar entre 0,1 µs y 1 segundo
`activar_timer0()`	Activa el funcionamiento del *timer* 0
`desactivar_timer0()`	Desactiva el funcionamiento del *timer* 0
`fijar_frecuencia(int frec)`	Calcula y fija el divisor de frecuencia del *timer* 0 para que genere interrupciones a la frecuencia de salida especificada por parámetro (en Hz)
`siguiente_movimiento(char *vel, unsigned char *avn)`	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
`swiWaitForVBlank()`	Espera hasta el próximo retroceso vertical
`printf(char *format,...)`	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();
```

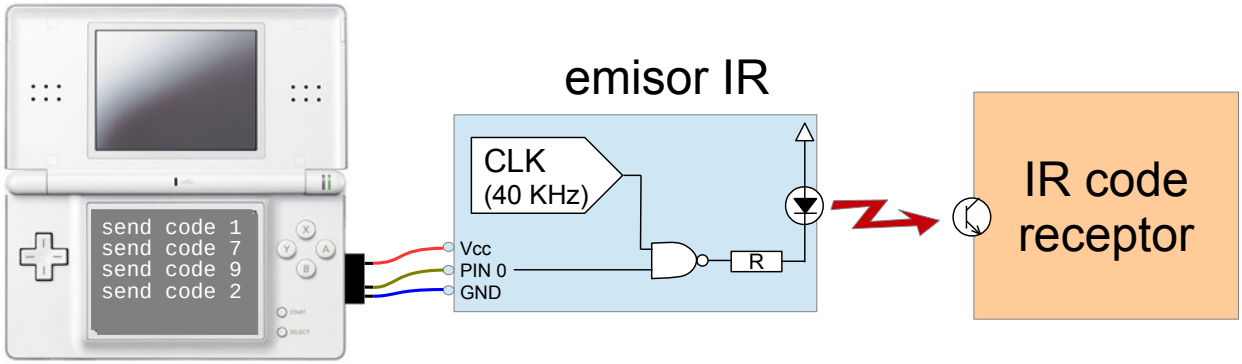
la cual debe detectar si se ha activado el bit IRQ_TIMER0 en el registro REG_IF del controlador de interrupciones y, en caso afirmativo, debe invocar la RSI del *timer* 0. Además, debe notificar la resolución de cualquier IRQ que se haya producido sobre el propio registro REG_IF, así como sobre la posición de memoria INTR_WAIT_FLAGS, para conseguir desbloquear posibles llamadas a funciones de la BIOS que esperan la generación de interrupciones, como la `swiWaitForVBlank()`. La rutina `inicializaciones()` instalará la dirección de la rutina `intr_main()` en la posición de memoria 0x0B003FFC, 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.

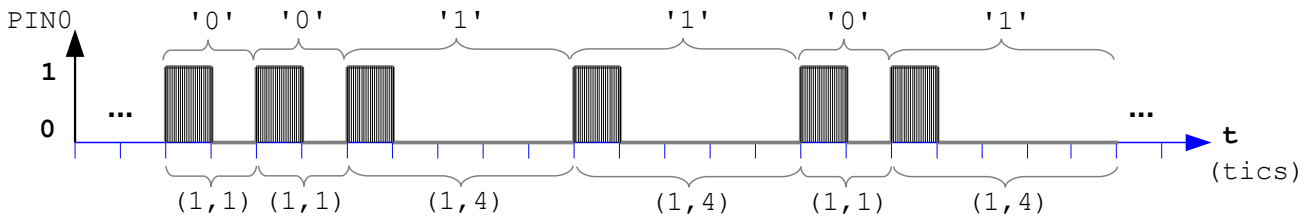
Problema 21: Emisor IR (Ex. 2ª Conv. 2016-17)

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 el 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 µs.



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 a '0' se codificará con el par (1, 1), mientras que un bit a '1' se codificará con el par (1, 4). Es decir, los dos tipos de bit activan la portadora durante un único tic, pero los bits a '0' la desactivan durante un solo tic, mientras que los bits 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. 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 menos peso del registro REG_KEYINPUT, que son todos menos los botones **X** e **Y**. Para cada botón se deberá iniciar la transmisión de una determinada ráfaga de 32 bits (32 pares ON/OFF), precedida de un par ON/OFF de inicio (*Lead In*) i 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).

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

Rutina	*Descripción*
`inicializaciones()`	Inicializa el *hardware* (pantalla, interrupciones, etc.)
`activar_timer0(int freqOut)`	Activa la generación de interrupciones del *timer* 0, a la frecuencia de salida especificada por parámetro
`desactivar_timer0()`	Desactiva la generación de interrupciones del *timer* 0
`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. No es necesario copiar la variables globales indicadas en este enunciado en las hojas de la solución.

Solución Problema 3

```
int main()
{
     short frec;               // variables locales
     int keys;

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

```
@; generar_vibracion: 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:
@;   short frec (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
          cmp r0, #0
          beq .Lfin_vibracion

          mov r1, r0                 @; R1 = frec. Salida (denominador)
          ldr r0, =32728             @; R0 = frec. Entrada (numerador)
          swi 9                      @; llamada a la BIOS (dividir)
          rsb r0, r0, #0             @; negar el divisor de frecuencia
          orr r0, #0x00C30000        @; activar valor para TIMER0_CR

     .Lfin_vibracion:
          str r0, [r2]               @; 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:
           push {r0-r1, lr}

           ldr r0, =REG_RUMBLE        @; R0 apunta al registro de E/S
           ldrb r1, [r0]              @; cargar valor actual en R1
           eor r1, #0x02              @; cambia el valor del bit 1
           strb r1, [r0]              @; actualiza registro de E/S

           pop {r0-r1, pc}
```

Solución 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();
     REG_TECL = -1;        // inicialmente no se selecciona ninguna fila
     do
     {
          tareas_independientes();
          if (currentKey != prevKey)      // detección cambio de tecla
          {
               prevKey = currentKey;
               if (currentKey != -1)      // si hay pulsación
               {
                    swiWaitForVBlank();
                    processKey(currentKey);
               }
          }
     } 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}

     mov r2, #3            @;R2 = código de columna testada (3,2,1,0)
     mov r3, #1            @;R3 es máscara de test (inicialmente, 0001)
.Ldesc_col:
     tst r0, r3
     beq .Ldesc_fincol     @;salir del bucle si ha encontrado bit a 0
     mov r3, r3, lsl #1    @;desplazar máscara a la izquierda
     sub r2, #1            @;siguiente código columna (de mayor a menor)
     cmp r2, #0
     bge .Ldesc_col        @;repetir mientras código de columna >= 0

     mov r0, #-1           @;R0 = código de no pulsación
     b .Ldesc_fin
```

```
.Ldesc_fincol:
     add r0, r2, r1, lsl #2     @;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.
@; 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
     ldsb r4, [r3]         @;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
     mov r4, #-1           @;reset código temporal
     mov r1, #0            @;reset número de fila

.Lkey_cont2:
     strb r4, [r3]         @;actualizar código de tecla temporal
     strb r1, [r2]         @;actualizar número de fila actual
     mov r0, #0x10
     mvn r0, r0, lsl r1    @;R0 tiene el bit de fila actual a cero
     strh r0, [r6]         @;actualiza REG_TECL para siguiente interrup.

.Lkey_fin:
     pop {r0-r6, pc}
```

Solución Problema 16

```
int current_ang = 0;       // valor actual del ángulo
int current_ci = 0;        // identificador imagen circular actual (0..1)
unsigned char rect_img[64*64];
unsigned int circ_img[2][256*3];

void main()
{
     inicializaciones();
     do
     {    swiWaitForVBlank();
          tareas_independientes();
          if (wifiReceiveImage(rect_img)) // si recepción nueva imagen
          {
               convertirImagen(rect_img, circ_img[1 - current_ci]);
               swiWaitForVBlank();
               mostrarImagen(rect_img);
               current_ci = 1 – current_ci;
          }
     } while (1);          // repetir siempre
}



@;RSI del timer 0: se activa 256*25 veces por segundo;
@; se utilizará para transferir una linea de píxeles circulares al
@; display de revolución, es decir, un radio con 32 áxels, según la
@; imagen circular actual 'current_ci' y el ángulo actual 'current_ang',
@; además de incrementar el ángulo y ponerlo a cero cuando llegue al
@; límite (256); también debe resetear el valor del ángulo si se detecta
@; la activación del bit de puesta a cero.
RSI_timer0:
     push {r0-r2, lr}

     ldr r0, =circ_img
     ldr r1, =current_ci
     ldr r1, [r1]          @;R1 = número actual de imagen circular
     cmp r1, #1
     addeq r0, #256*12     @;R0 apunta al inicio de imagen circular act.
     ldr r2, =current_ang
     ldr r1, [r2]          @;R1 = ángulo actual
     bl transferir_radio
     add r1, #1            @;incrementar ángulo actual
     cmp r1, #256          @;si ángulo >= 256,
     movhs r1, #0          @; resetear ángulo actual

     ldr r0, =RDISP_STATUS
     ldrh r0, [r0]         @;R0 = bit de puesta a cero
     tst r0, #1            @;si bit a 1
     movne r1, #0          @; también hay que resetear ángulo actual

     str r1, [r2]          @;actualizar variable global

     pop {r0-r2, pc}
```

```
@;transferir_radio: a partir de la dirección base de una imagen circular
@; y del número de radio (ángulo) que se pasan por parámetro, obtener el
@; estado de los bits de rojo, verde y azul para los 32 áxels, y
@; generar un patrón de 32 bits donde cada bit indicará si el LED
@; correspondiente debe estar encendido o apagado, según el estado del
@; botón SELECT:
@;    soltado (=1): activar LED si el bit verde está a 1 y alguno de los
@;                  otros dos bits está a 1 (pero no los dos a la vez)
@;    pulsado (=0): activar LED si algún bit de color está a 1,
@; La rutina envía los 32 bits resultantes por el registro de datos del
@; dispositivo.
@; Parámetros:
@;    R0 = dirección base de la imagen circular
@;    R1 = número de radio (ángulo : [0..255])
transferir_radio:
     push {r0-r3, lr}

     mov r2, #12
     mla r0, r1, r2, r0    @;R0 += núm.radio (ang.) * 12 bytes/ángulo
     ldr r1, [r0]          @;R1 carga bits de rojo
     ldr r2, [r0, #4]      @;R2 carga bits de verde
     ldr r3, [r0, #8]      @;R3 carga bits de azul
     ldr r0, =REG_KEYINPUT
     ldrh r0, [r0]         @;R0 carga estado botones
     tst r0, #0x04         @;comprobar estado bit de SELECT
     beq .Ltrans_select    @;saltar si bit SELECT = 0 (botón pulsado)
     eor r3, r1            @;R3: bit activo si bit azul o rojo activo,
                           @; pero no los dos a la vez (or exclusiva)
     and r1, r2, r3        @;R1: bit activo si bit verde y bit R3 activo
     b .Ltrans_cont
.Ltrans_select:
     orr r1, r2
     orr r1, r3            @;R1: bit activo si algún bit de color activo
.Ltrans_cont:
     ldr r0, =RDISP_DATA
     str r1, [r0]          @;transferir 32 bits del radio actual

     pop {r0-r3, pc}
```

Solución Problema 20

```
unsigned short pasos = 0;       // número de pasos pendientes a 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 (pasos == 0)           // si el motor está parado
          {
               siguiente_movimiento(&velocidad, &avance);
               if (velocidad != 0)
               {                    // sentido es el signo de vel.
                    sentido = (velocidad < 0 ? -1 : 1);
                                    // obtener valor absoluto de vel.
                    velocidad = velocidad * sentido;
                                    // frec. pasos = pasos/cm * cm/s
                    fijar_frecuencia(243 * velocidad);
                                    // pasos avance = pasos/cm * cm
                    pasos = 243 * avance;
                    if (pasos > 0) activar_timer0();
                    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}

      ldr r0, =REG_IF
      ldr r1, [r0]          @;R1 = valor actual de REG_IF
      tst r1, #IRQ_TIMER0   @;verificar si se ha activado IRQ_TIMER0
      blne RSI_timer0       @;en caso afirmativo, llamar la RSI específica

      str r1, [r0]          @;marcar todas las IRQ activadas en REG_IF
      ldr r2, =INTR_WAIT_FLAGS
      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
      ldrb r1, [r0]             @;R1 es fase actual
      ldr r2, =sentido
      ldsb r2, [r2]             @;R2 es sentido de avance
      add r1, r2                @;actualizar fase (inc. o dec.)
      and r1, #7                @;módulo 8 para actualización circular
      strb r1, [r0]             @;actualiza variable global 'fase'

      ldr r0, =Vphases
      ldrb r2, [r0, r1]         @;R2 es estado de los bits de salida
                                @;según la nueva fase

      mov r0, #0x0A000000       @;R0 es dirección registro E/S
      ldr r1, [r0]              @;R1 es valor actual del reg. E/S
      bic r1, #0xF0             @;limpiar bits b7-b4
      orr r1, r2                @;añadir bits de fase
      str r1, [r0]              @;actualiza reg. E/S

      ldr r0, =pasos
      ldrh r1, [r0]             @;decrementa contador de pasos restantes
      subs r1, #1               @;si llega a cero, FZ = 1
      bleq desactivar_timer0    @;si FZ = 1, parar las interrupciones
      strh r1, [r0]

      pop {r0-r2, pc}
```