Laboratorio 6:

Medida del tiempo

control de temporizadores
entrada por pulsadores y keypads

Programación de sistemas y dispositivos

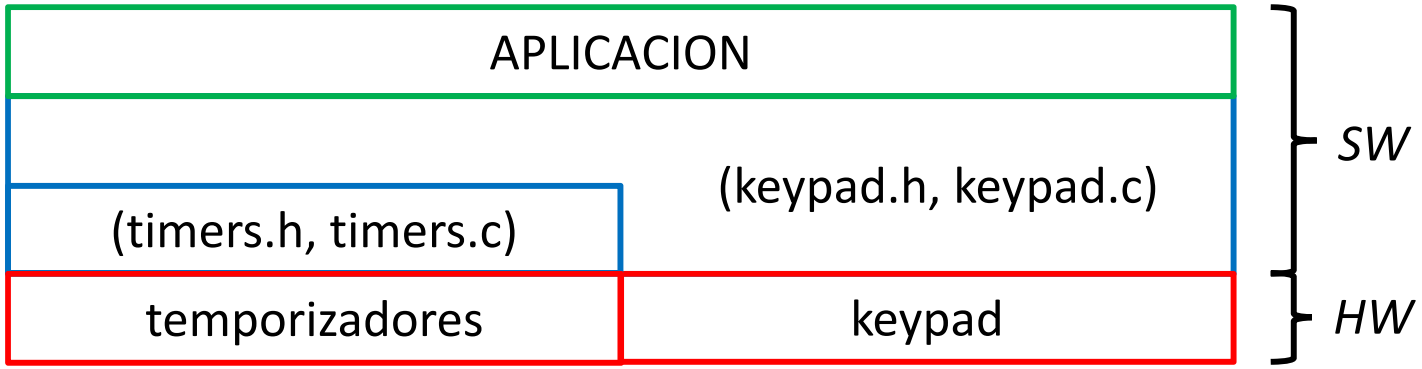
José Manuel Mendías Cuadros
Dpto. Arquitectura de Computadores y Automática
Universidad Complutense de Madrid

Presentación

- Desarrollar una capa de firmware para la medida precisa del tiempo usando un temporizador.
  - Implementaremos 12 funciones :
    - Inicialización: **`timers_init`**
    - Espera HW por un intervalo de tiempo: **`timer3_delay_ms / timer3_delay_s`**
    - Espera SW por un intervalo de tiempo: **`sw_delay_ms / sw_delay_s`**
    - Arranque/parada de un temporizador: **`timer3_start / timer3_stop`**
    - Arranque/consulta de un temporizador: **`timer3_start_timeout / timer3_timeout`**
    - Activación/desactivación de interrupciones por fin de cuenta de temporizadores, así como instalación de la RTI que las atenderá: **`timer0_open_tick / timer0_open_ms / timer0_close`**

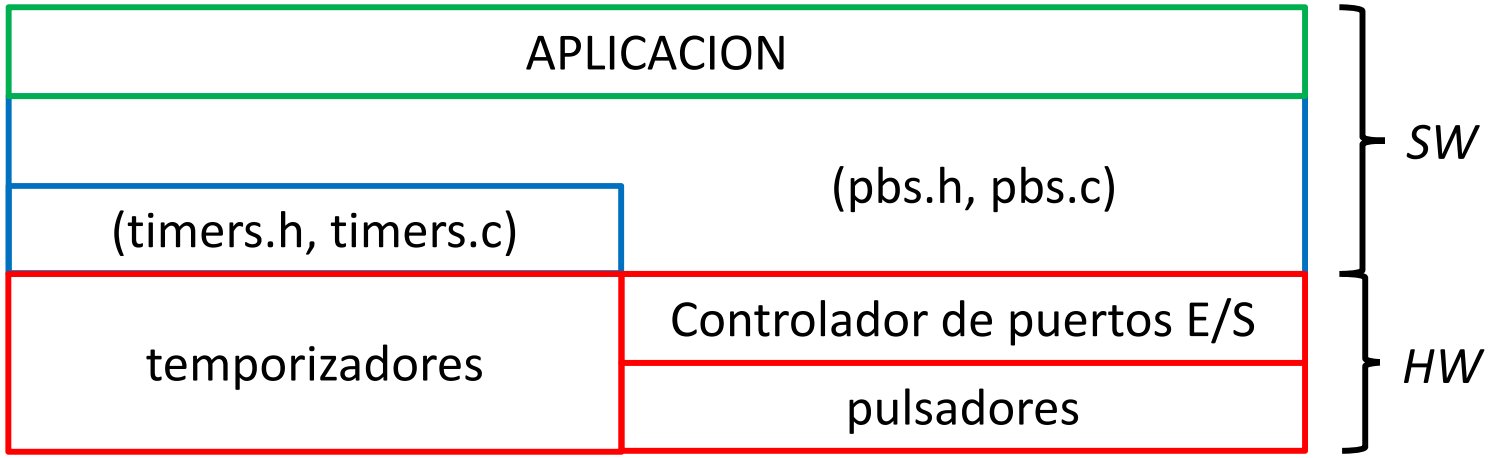
Presentación

- Desarrollar una capa de firmware para leer datos de un keypad
  - Implementaremos 9 funciones sin gestión del tiempo:
    - Inicialización: **`keypad_init`**
    - Lectura del estado del keypad: **`keypad_scan`**
    - Espera por des/pulsación de una tecla: **`keypad_wait_keydown / keypad_wait_keyup`**
    - Espera por des/pulsación del keypad: **`keypad_wait_any_keydown / keypad_wait_any_keyup`**
    - Espera por pulsación y lectura del keypad: **`keypad_getchar`**
    - Activación/desactivación de interrupciones por pulsación del keypad, así como instalación de la RTI que las atenderá: **`keypad_open / keypad_close`**
  - Implementaremos 2 funciones con gestión del tiempo:
    - Espera por pulsación, lectura del keypad y medida del tiempo: **`keypad_getchartime`**
    - Espera con timeout por pulsación y lectura del keypad: **`keypad_timeout_getchar`**



Presentación

- Desarrollar una capa de firmware para leer el estado de pulsadores
  - Implementaremos 7 funciones sin gestión del tiempo:
    - Inicialización: **`pbs_init`**
    - Lectura del estado de un pulsador: **`pb_status`**
    - Espera por des/pulsación de un pulsador: **`pb_wait_keydown / pb_wait_keyup`**
    - Espera por pulsación y lectura de pulsadores: **`pbs_getchar`**
    - Activación/desactivación de interrupciones por pulsación de pulsadores, así como instalación de la RTI que las atenderá: **`pbs_open / pbs_close`**
  - Implementaremos 2 funciones con gestión del tiempo:
    - Espera por pulsación, lectura de pulsadores y medida del tiempo: **`pb_getchartime`**
    - Espera con timeout por pulsación y lectura de pulsadores: **`pb_timeout_getchar`**



Generación de retardos

por software

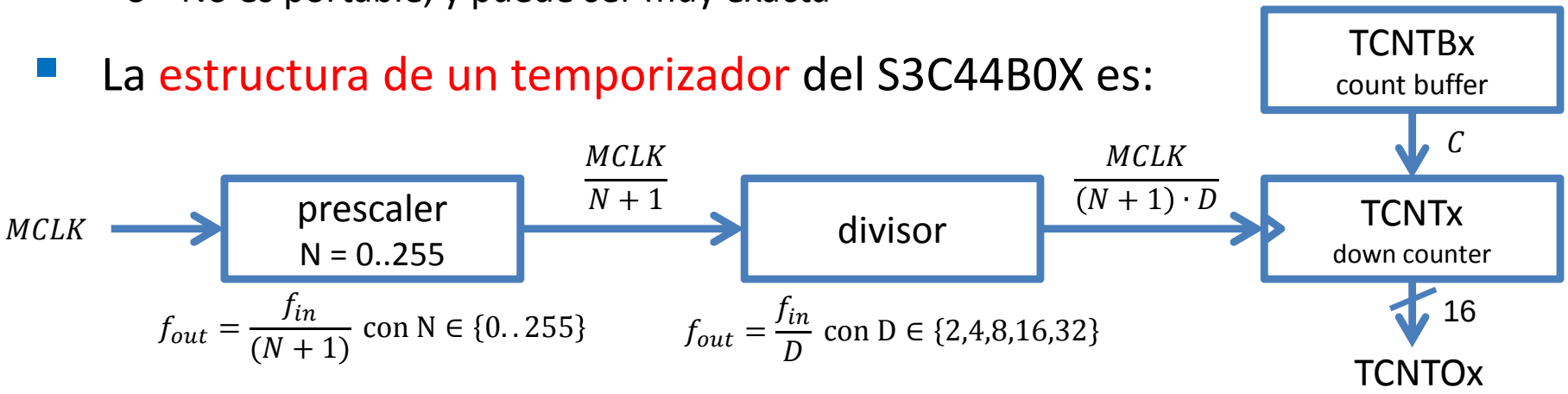
- La generación de retardos por software
  - Se realiza ejecutando una colección de instrucciones de duración conocida.
  - No requiere hardware adicional, pero necesita un ajuste empírico.
  - No es portable y puede no ser exacta:
    - dependen de la frecuencia de reloj del sistema
    - dependen del compilador (i.e. opciones de optimización)
    - depende de la arquitectura (i.e. cache, interrupciones, DMA...)
  - Se usan cuando el retardo puede ser aproximado

- Ejemplos para el microcontrolador S3C44B0X:
  - 64 MHz, sin cache, sin interrupciones, sin DMA y sin optimizar la compilación.

```
asm( "nop" );                    retarda 15,6 ns (1 ciclo)
for( i=48; i; i-- );             retarda aprox. 0,1 ms
for( i=487; i; i-- );            retarda aprox. 1 ms
for( i=4874; i; i-- );           retarda aprox. 10 ms
for( i=48744; i; i-- );          retarda aprox. 0.1 s
for( i=487440; i; i-- );         retarda aprox. 1 s
```

Generación de retardos
por hardware (i)

- La generación de retardos por hardware
  - Se realiza utilizando un temporizador hardware
    - Un contador de anchura fija que cuenta a frecuencia programable
  - La programación de temporizadores no es trivial
    - Requiere llegar a un compromiso entre resolución y retardo máximo alcanzable
  - No es portable, y puede ser muy exacta
- La estructura de un temporizador del S3C44B0X es:



$$f_{counter} = \frac{MCLK}{(N+1)\cdot D} \Rightarrow t_{counter} = \frac{(N+1)\cdot D}{MCLK}$$

$$\begin{cases} resolución\ (retardo\ min.) \equiv t_{counter} = \dfrac{(N+1)\cdot D}{MCLK} \\ retardo\ = C \cdot \dfrac{(N+1)\cdot D}{MCLK}\ con\ 0 < C\ < 2^{16} \\ retardo\ máx. = 2^{16} \cdot \dfrac{(N+1)\cdot D}{MCLK} \end{cases}$$

Generación de retardos
por hardware (ii)

- Para obtener un retardo dado es necesario ajustar N, D y C
  - En general, para conseguir una máxima precisión N y D deben ser lo más pequeños posible tal que C no supere 2^{16}

retardo	N	D	resolución	C
1 ms	0	2	(0+1)·2/(64 MHz) = 31.25 ns	(1 ms)/(31.25 ns) = **32000**
1 µs	0	2	(0+1)·2/(64 MHz) = 31.25 ns	(1 µs)/(31.25 ns) = **32**
10 ms	0	2	(0+1)·2/(64 MHz) = 31.25 ns	(10 ms)/(31.25 ns) = 320000 **no válido** por ser > 2^{16}
10 ms	0	8	(0+1)·8/(64 MHz) = 125 ns	(10 ms)/(125 ns) = **8000**
1 s	63	32	(63+1)·32/(64 MHz) = 32 µs	(1 s)/(32 µs) = **31250**

- Si el retardo requerido es mayor que el alcanzable por el temporizador
  - Se anida en el cuerpo de un bucle for.

```
for( i=0; i<3600; i++ )
  wait_for_1s();
```
} espera 1 hora

Temporizador 3

configuración

- Por ejemplo, para retardar 1 segundo:

$$(N, D, C) = (63{,}32{,}31250) \Rightarrow resolución = \frac{(63+1)\cdot 32}{64\ MHz} = 32\ \mu s \Rightarrow retardo = 31250 \cdot 32\ \mu s = 1\ s$$

- Prescaler: 63
  - TCFG0[15:8] = 63
- Divisor: 32
  - TCFG1[15:12] = 4
- Count buffer: 31250
  - TCNTB3 = 31250
- Modo: one-shot
  - TCON[19] = 0
- DMA: desactivada
  - TCON[27:24] = 0
- No es necesario configurar:
  - TCON[18], porque el temporizador no va a tener salida al exterior
  - TCMPB3, porque es irrelevante la forma concreta de la onda

Temporizador 3

control y acceso a datos

- Para cargar manualmente el contador TCNT3:
  - TCON[17] = 1
- Para arrancar manualmente la cuenta:
  - TCON[16] = 1
- Para conocer el estado de la cuenta:
  - Leer TCNTO3
- Para parar manualmente la cuenta:
  - TCON[16] = 0

```
void wait_for_1s( void )
{
  TCFG0 = (TCFG0 & ~(0xff << 8)) | (63 << 8);  ····· T2-T3 prescaler: N=63
  TCFG1 = (TCFG1 & ~(0xf << 12)) | (4 << 12);  ····· T3 divisor (1/32): D=32
  TCNTB3 = 31250;  ····· T3 count: C = 31250
  TCON = (TCON & ~(0xf << 16)) | (1 << 17);  ····· one shot, carga TCNT3, stop T3
  TCON = (TCON & ~(0xf << 16)) | (1 << 16);  ····· one shot, no carga TCNT3, start T3
  while( !TCNTO3 );  ····· espera a que TCNTO3 se actualice
  while( TCNTO3 );  ····· espera a que TCNTO3 sea 0
}
```

Medida de tiempos

por hardware (i)

- Para medir el tiempo transcurrido entre 2 eventos:
  - Solo puede hacerse por hardware
  - Al detectar el 1er. evento se arranca el temporizador (a una resolución dada).
  - Al detectar el 2do. evento se para el temporizador.
  - El valor de la cuenta del temporizador (multiplicado por su resolución) indicará el tiempo transcurrido.

- Por ejemplo, para medir el tiempo de ejecución de una porción de código:

```
...
timer3_start();
...porción de código a medir...
n = timer3_stop();
...
```

Medida de tiempos

por hardware (ii)

- Por ejemplo, para medir el tiempo con resolución de 0,1 ms:

$$(N, D, C) = (199,32,65535) \Rightarrow resolución = \frac{(199+1)\cdot 32}{64\ MHz} = 100\ \mu s \Rightarrow medida\ máx. = 2^{16}\cdot 100\ \mu s = 6.55\ s$$

```
void timer3_start( void )
{
  TCFG0 = (TCFG0 & ~(0xff << 8)) | (199 << 8);   ····· T2-T3 prescaler: N=199
  TCFG1 = (TCFG1 & ~(0xf << 12)) | (4 << 12);    ····· T3 divisor (1/32): D=32
  TCNTB3 = 0xffff;                               ····· T3 count: C = máximo
  TCON = (TCON & ~(0xf << 16)) | (1 << 17);      ····· one shot, carga TCNT3, stop T3
  TCON = (TCON & ~(0xf << 16)) | (1 << 16);      ····· one shot, no carga TCNT3, start T3
  while( !TCNTO3 );                              ····· espera a que TCNTO3 se actualice
}
```

```
uint16 timer3_stop( void )
{
  TCON &= ~(1 << 16);                            ····· detiene el timer
  return 0xffff - TCNTO3;                        ····· calcula los ciclos de cuenta transcurridos
}
```

Timeouts
por hardware (i)

- La espera activa hasta la ocurrencia de un evento es una de las fuentes más frecuentes de bloqueo en un sistema empotrado

```
while( !(UFSTAT0 & 0xf) );
```
espera indefinidamente la llegada de información por la UART0...

- En una aplicación robusta, toda espera deben finalizar transcurrido un cierto tiempo (timeout)
  - Se puede hacer por software cuando no se requiere demasiada precisión y solo se quiere evitar el bloqueo:

```
for( i=TIMEOUT; !(UFSTAT0 & 0xf) && i; i-- );
```

  - Se puede hacer con precisión por hardware usando un temporizador:

```
timer3_start_timeout( TIMEOUT );
while( !(UFSTAT0 & 0xf) && !timer3_timeout() );
```

Timeouts
por hardware (ii)

- Por ejemplo, para disponer de timeouts con resolución de 0,1 ms:

$$(N, D, C) = (199{,}32, TIMEOUT) \Rightarrow resolución = \frac{(199+1)\cdot 32}{64\ MHz} = 100\ \mu s \Rightarrow timeout\ máx. = 2^{16}\cdot 100\ \mu s = 6.55\ s$$

```
void timer3_start_timeout( uint16 n )
{
  TCFG0 = (TCFG0 & ~(0xff << 8)) | (199 << 8);   ····· T2-T3 prescaler: N=199
  TCFG1 = (TCFG1 & ~(0xf << 12)) | (4 << 12);    ····· T3 divisor (1/32): D=32
  TCNTB3 = n;                                    ····· T3 count: C = timeout
  TCON = (TCON & ~(0xf << 16)) | (1 << 17);      ····· one shot, carga TCNT3, stop T3
  TCON = (TCON & ~(0xf << 16)) | (1 << 16);      ····· one shot, no carga TCNT3, start T3
  while( !TCNTO3 );                              ····· espera a que TCNTO3 se actualice
}
```

```
uint16 timer3_timeout( void );
{
  return !TCNTO3;
}
```

Medida de tiempos

técnica mixta

- Si los tiempos a medir son superiores al máximo alcanzable por el temporizador
  - Es necesario contar por software el número de veces que el temporizador ha completado la cuenta programada.
- La función de arranque del temporizador
  - Programa TCNTBx a un valor fijo y pone el temporizador en modo autorrecarga
  - Inicializa a 0 la variable global, n, que lleva la cuenta del número de veces que el temporizador ha finalizado
  - Instala una RTI que incrementa n
  - Desenmascara las interrupciones del temporizador
- La función de parada del temporizador
  - Para el temporizador, enmascara interrupciones y desinstala la RTI
  - El tiempo transcurrido será: TCNTOx + n × TCNTBx (multiplicado la resolución)
- Un enfoque análogo puede aplicarse a timeouts
  - Con la diferencia de que inicialmente n se inicializa al timeout y la RTI decrementa n
  - El timeout se alcanza cuando n vale 0, en cuyo caso debe pararse el temporizador

Generación de retardos

técnica mixta (i)

- Para evitar dedicar un temporizador cada vez que se requiere hacer un retardo o establecer un timeout es posible adoptar una solución mixta.
  - Usar un temporizador para ajustar automáticamente el número de iteraciones que debe hacer un bucle software para que tarde un tiempo determinado.
    - El temporizador solo se usa 1 vez durante el ajuste.
    - Los retardos se generan por software.

- Para ello se requiere:
  - Una variable global que almacene el número de iteraciones requerido para que un bucle vacío tarde exactamente un tiempo dado (por ejemplo, 1s)
  - Una rutina de inicialización que:
    - Usando un temporizador calcule el tiempo que tarda en ejecutarse un bucle vacío un número de iteraciones fijo y conocido.
    - Conocido dicho tiempo, hace una regla de 3 para determinar el número de iteraciones que tiene que hacer el bucle tardar el tiempo dado.
    - Almacena en la variable global el número calculado.
  - Una rutina de retardo software que
    - Ejecute un bucle vacío el número de veces indicado por la variable global.

Generación de retardos

técnica mixta (ii)

```
uint32 loop_ms = 0;
uint32 loop_s = 0;

static void sw_delay_init( void )
{
  uint32 i;

  timer3_start();
  for( i=1000000; i; i-- );
  loop_s = ((uint64)1000000*10000)/timer3_stop();
  loop_ms = loop_s / 1000;
};

void sw_delay_ms( uint16 n )
{
  uint32 i;

  for( i=loop_ms*n; i; i-- );
}

void sw_delay_s( uint16 n )
{
  uint32 i;

  for( i=loop_s*n; i; i-- );
}
```

almacena el número de iteraciones para retardar 1 ms

almacena el número de iteraciones para retardar 1 s

la resolución del temporizador es 100µs

mide la duración de 1 millón de iteraciones

Regla de tres:
si 10^6 iteraciones tardan n× 100µs
en 1s se harán: 10^6/ (n× 0,0001) iteraciones

en 1 ms se harán 1000 veces menos iteraciones que en 1 s

todos los bucles de espera tienen una estructura similar

Generación de retardos

cambio de frecuencia de reloj

- Los ajustes de N, D y C para un retardo dado asumen MCLK = 64 MHz
  - Si la frecuencia de reloj cambia, todas las rutinas anteriores son inválidas.

- Por ejemplo, la rutina wait_for_1s retarda la ejecución:
  - 128s si el procesador pasa a modo SLOW (MCLK = 500 KHz)
  - 8s si se desahabilita el PLL (MCLK = 8 MHz)

- Por ello, si la aplicación requiere funcionar a distintas frecuencias
  - El firmware tendrá que gestionar la correcta programación de los temporizadores a las distintas frecuencias de funcionamiento posibles

```
extern uint32 mclk;

void wait_for_1s( void )
{
  ...
  TCNTB3 = (uint16) (mclk/2048);
  ...
}
```

el sistema mantiene actualizado el valor de ciclos/s del MCLK

(N+1)·D = (63+1)·32

la rutina en lugar de constante calcula los valores de inicialización necesarios (pueden estar tabulados en el caso de necesitar cambios de prescaler/divisor)

Driver de temporizadores

timers.h

```
#ifndef __TIMERS_H__
#define __TIMERS_H__

#include <common_types.h>

#define TIMER_ONE_SHOT (0)
#define TIMER_INTERVAL (1)

void timers_init( void );
void timer3_delay_ms( uint16 n );
void timer3_delay_s( uint16 n );
void sw_delay_ms( uint16 n );
void sw_delay_s( uint16 n );
void timer3_start( void );
uint16 timer3_stop( void );
void timer3_start_timeout( uint16 n );
uint16 timer3_timeout( void );
void timer0_open_tick( void (*isr)(void), uint16 tps );
void timer0_open_ms( void (*isr)(void), uint16 ms, uint8 mode );
void timer0_close( void );

#endif
```

modos de funcionamiento de los timers (con o sin recarga automática)

Driver de temporizadores

timers.c

```
extern void isr_TIMER0_dummy( void );

static uint32 loop_ms = 0; ..... almacena el número de iteraciones para retardar 1 ms
static uint32 loop_s = 0;..... almacena el número de iteraciones para retardar 1 s

static void sw_delay_init( void );

void timers_init( void )
{
  TCFG0  = ...;
  TCFG1  = ...;                  } pone a 0 los registros de configuración
  TCNTB0 = ...;
  ...                            } pone a 0 los count buffer de todos los temporizadores
  TCMPB0 = ...;
  ...                            } pone a 0 los compare buffer de todos los temporizadores
  TCON   = ...; ..... carga y para todos los TCNTx
  TCON   = ...; ..... no carga y para todos los TCNTx
  sw_delay_init();
}

void timer3_delay_s( uint16 n )
{
  for( ; n; n-- )
  { ... }; ..... efectúa un retardo HW de 1 s con el timer 3
}
```

Driver de temporizadores

timers.c

```
void timer0_open_ms( void (*isr)(void), uint16 ms, uint8 mode )
{
  pISR_TIMER0 = ...;  ····· instala la RTI argumento en la tabla virtual de vectores de IRQ
  I_ISPC      = ...;  ····· borra flag de interrupción pendiente por interrupciones del timer 0
  INTMSK     &= ...;  ····· desenmascara globalmente interrupciones e interrupciones del timer 0
  TCFG0       = ...;
  TCFG1       = ...;  } programa el T0 con resolución de 100 µs
  TCNTB0      = 10*ms; ····· 1 ms = 10 intervalos de 100 µs
  TCON        = ...;  ····· mode, carga TCNT0, stop T0
  TCON        = ...;  ····· mode, no carga TCNT0, start T0
}

void timer0_close( void )
{
  TCNTB0      = ...;  ····· pone a cero el count buffer del timer 0
  TCMPB0      = ...;  ····· pone a cero el compare buffer del timer 0
  TCON        = ...;  ····· carga TCNT0, stop T0
  TCON        = ...;  ····· no carga TCNT0, stop T0
  INTMSK     |= ...;  ····· enmascara interrupciones por fin de timer 0
  pISR_TIMER0 = ...;  ····· instala isr_TIMER0_dummy en la tabla virtual de vectores de interrupción
}
```

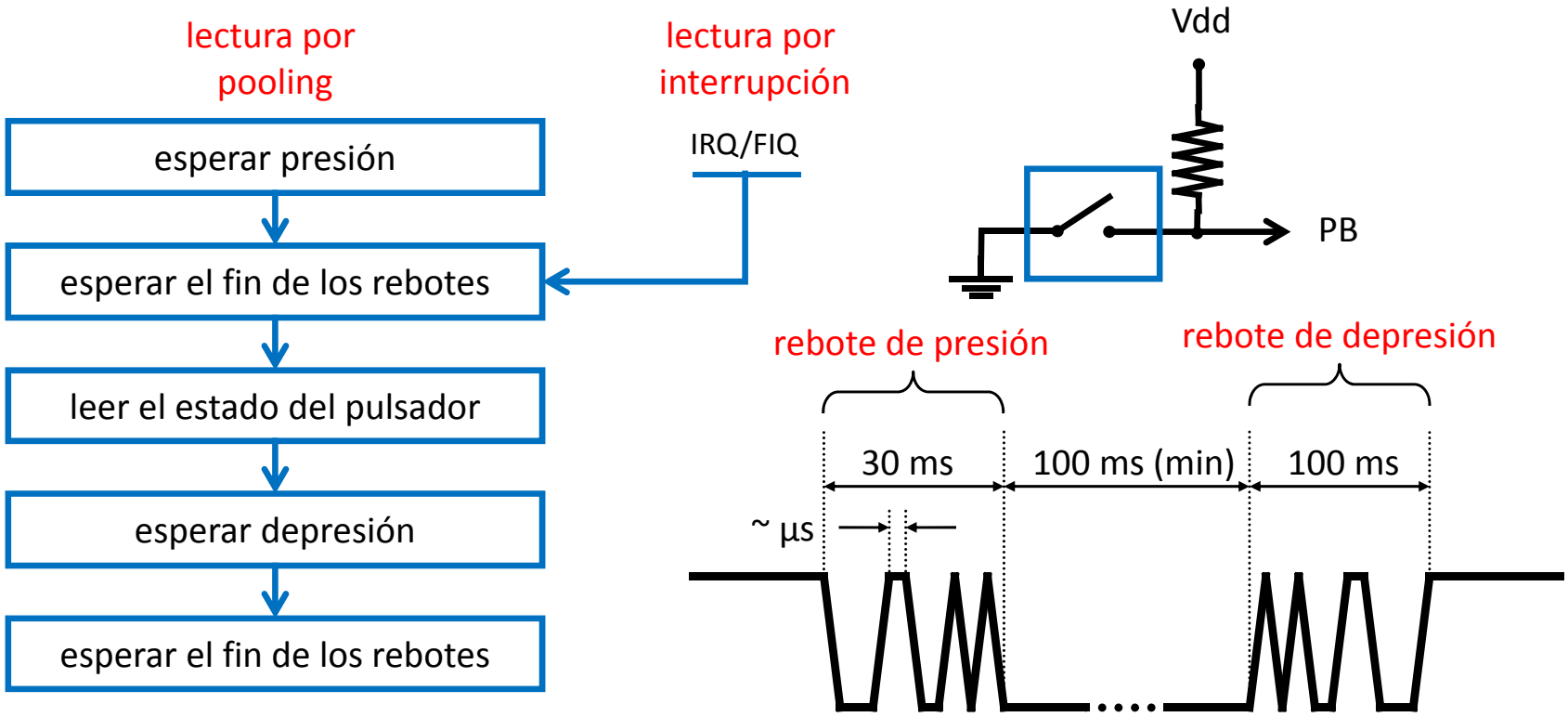
Driver de temporizadores

timers.c

```
void timer0_open_tick( void (*isr)(void), uint16 tps )
{
  pISR_TIMER0 = ...;   ····· instala la RTI argumento en la tabla virtual de vectores de IRQ
  I_ISPC      = ...;   ····· borra flag de interrupción pendiente por interrupciones del timer 0
  INTMSK     &= ...;   ····· desenmascara globalmente interrupciones e interrupciones del timer 0
  if( tps > 0 && tps <= 10 ) {
    TCFG0 = ...;
    TCFG1 = ...;                                   } programa el T0 con resolución de 25 μs (40 KHz)
    TCNTB0 = (40000U / tps); ····· permite obtener el num. de ticks/s indicado
  } else if( tps > 10 && tps <= 100 ) {
    TCFG0 = ...;
    TCFG1 = ...;                                   } programa el T0 con resolución de 2,5 μs (400 KHz)
    TCNTB0 = (400000U / (uint32) tps);
  } else if( tps > 100 && tps <= 1000 ) {
    TCFG0 = ...;
    TCFG1 = ...;                                   } programa el T0 con resolución de 0,25 μs (4 MHz)
    TCNTB0 = (4000000U / (uint32) tps);
  } else if ( tps > 1000 ) {
    TCFG0 = ...;
    TCFG1 = ...;                                   } programa el T0 con resolución de 31,25 ns (32 MHz)
    TCNTB0 = (32000000U / (uint32) tps);
  }
  TCON = ...;  ····· interval, carga TCNT0, stop T0
  TCON = ...;  ····· interval, no carga TCNT0, start T0
}
```

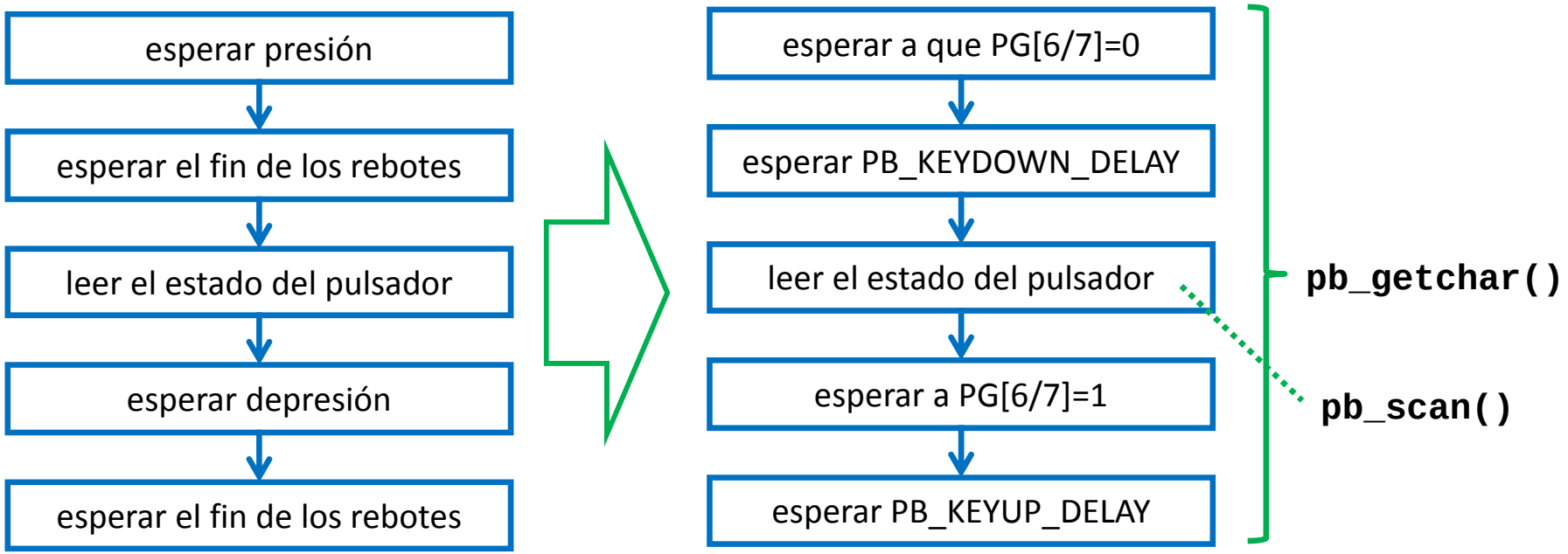
Pulsadores

- La lectura del estado de un interruptor mecánico presenta el problema de existencia de rebotes:
  - Cuando el estado del interruptor cambia, la señal presenta un vaivén transitorio.
  - Este vaivén debe ignorarse y nunca ser interpretado como una serie de pulsaciones.
    - Además, si un pulsador es fuente externa de interrupciones, los rebotes ocasionan que una sola pulsación genere varias interrupciones que es necesario filtrar.



Pulsadores

- En la placa S3CEV40 los pulsadores están conectados al puerto G
  - El derecho al PG[7] y el izquierdo al PG[6]
  - Ambos son de lógica inversa (la pulsación genera un 0).
- Hemos configurado el controlador de E/S para que:
  - Estén conectados a las entradas EINT7 y EINT6 del controlador de interrupciones.
  - Se generen interrupciones (siempre que no estén enmascaradas) a flancos de bajada.
    - En cualquier caso PG[6/7] pueden leerse como si fueran puertos de entrada.



Driver de pulsadores

pbs.h

```
#ifndef __PBS_H__
...
#define PB_RIGHT   (1 << 7)
#define PB_LEFT    (1 << 6)
#define PB_FAILURE (0xff)
#define PB_TIMEOUT (0xfe)
#define PB_DOWN    (1)
#define PB_UP      (0)

void pbs_init( void );
uint8 pb_status( uint8 scancode );
void pb_wait_keydown( uint8 scancode );
void pb_wait_keyup( uint8 scancode );
void pb_wait_any_keydown( void );
void pb_wait_any_keyup( void );
uint8 pb_scan( void );
uint8 pb_getchar( void );
uint8 pb_getchartime( uint16 *ms );
uint8 pb_timeout_getchar( uint16 ms);
void pbs_open( void (*isr)(void) );
void pbs_close( void );

#endif
```

Declara macros para identificar a cada pulsador

Declara macros para identificar errores durante la lectura de pulsadores

Declara macros para identificar el estado de un pulsador

Driver de pulsadores

pbs.c

```
void pbs_init( void )
{
  timers_init(); ..... únicamente inicializa temporizadores,
                       la configuración de puertos la hace system_init()
}

void pbs_open( void (*isr)(void) )
{
  pISR_PB    = ...; ..... instala la RTI argumento en la tabla virtual de vectores de IRQ
  EXTINTPND = ...; ..... borra flag de interrupción pendiente por interrupciones externas
  I_ISPC     = ...; ..... borra flag de interrupción pendiente por interrupciones por presión de pulsador
  INTMSK    &= ...; ..... desenmascara globalmente interrupciones e interrupciones por presión de pulsador
}

void pbs_close( void )
{
  INTMSK   |= ...; ..... enmascara interrupciones por presión de pulsador
  pISR_PB   = ...; ..... instala isr_PB_dummy en la tabla virtual de vectores de interrupción
}
```

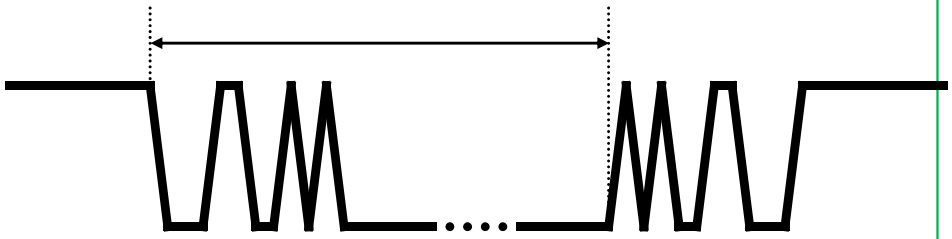
Driver de pulsadores

pbs.c

```
uint8 pb_scan( void )
{
  if( ... )
    return PB_LEFT;
  else if( ... )
    return PB_RIGHT;
  else
    return PB_FAILURE;
}
```

Lee secuencialmente los pulsadores para determinar el código a devolver

si ninguno está pulsado devuelve fallo



```
uint8 pb_getchartime( uint16 *ms )
{
  uint8 scancode;

  while( ... );
  timer3_start();
  sw_delay_ms( PB_KEYDOWN_DELAY );
  scancode = pb_scan();
  while( ... );
  *ms = timer3_stop() / 10;
  sw_delay_ms( PB_KEYUP_DELAY );
  return scancode;
}
```

espera la presión de cualquier pulsador

arranca el timer 3 (0,1 ms de resolución)

espera SW (el timer 3 está ocupado) fin de rebotes

obtiene el código del pulsador presionado

espera la depresión del pulsador

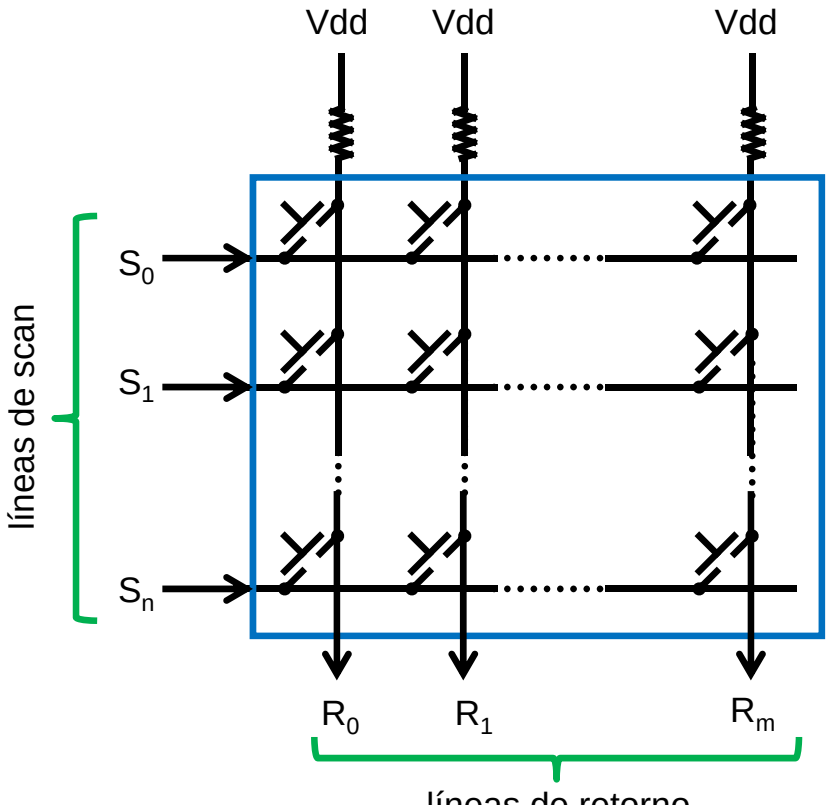
detiene el timer 3 y calcula los ms

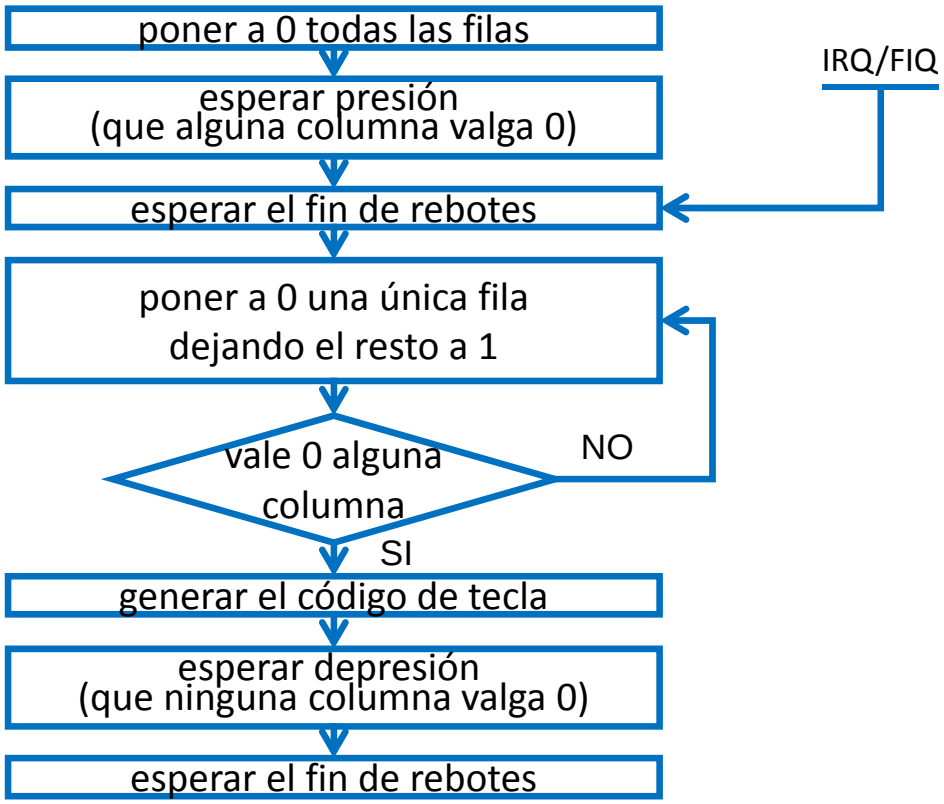
espera SW (el timer 3 está ocupado) fin de rebotes

devuelve el código del pulsador presionado

Keypad

- Un keypad es una colección de pulsadores dispuestos en filas y columnas
  - Los pulsadores de la misma fila/columna comparten uno de sus terminales.
  - Cuando se pulsa una tecla, se conecta una fila y una columna .
- Para saber la tecla pulsada es necesario realizar un proceso de scan
  - Enviando códigos por las líneas de scan y leyendo los código de retorno generados.

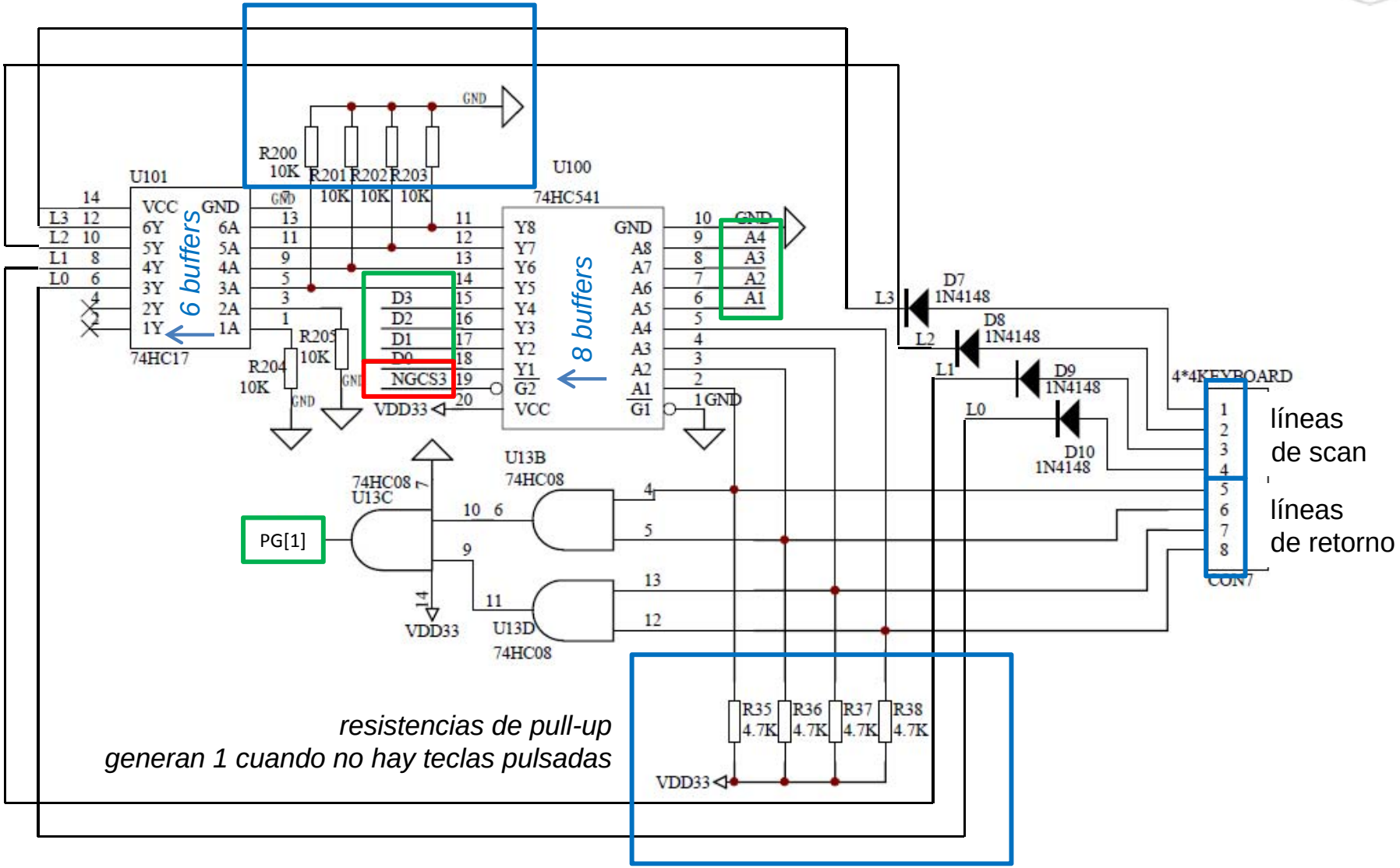




Keypad

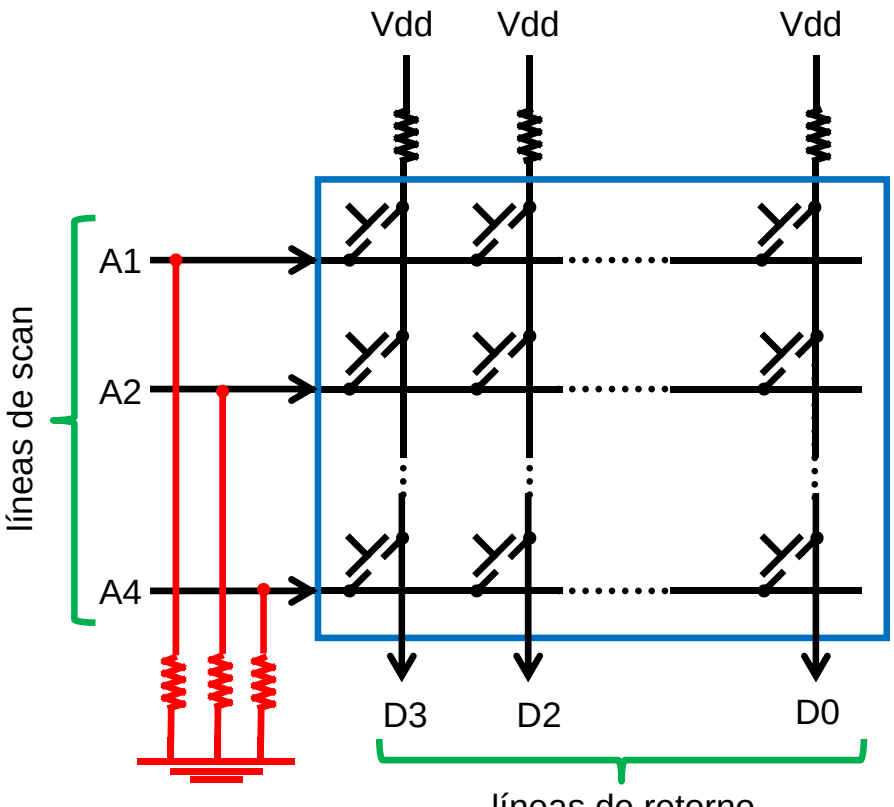
- En la placa S3CEV40 las líneas de scan y retorno están mapeadas en el banco 3 de memoria de la siguiente manera:
  - El código de scan se envía al keypad través del bus de direcciones siempre y cuando la dirección se corresponda al banco 3: (A27, A26, A25) = (011)
    - A1 se corresponde a la primera fila (de arriba a abajo) del keypad
    - A2 se corresponde a la segunda fila del keypad
    - A3 se corresponde a la tercera fila del keypad
    - A4 se corresponde a la cuarta fila del keypad
  - Sea cual sea la dirección enviada, el código de retorno se recibe del keypad a través del bus de datos.
    - D4 se corresponde a la primera columna (de izquierda a derecha) del keypad
    - D3 se corresponde a la segunda columna del keypad
    - D2 se corresponde a la tercera columna del keypad
    - D1 se corresponde a la cuarta columna del keypa
  - La y-lógica de todos los bits del código de retorno está conectada al PG[1]
    - Si el código de scan es 0, cuando se pulse una tecla PG[1] pasará de 1 a 0
  - Hemos configurado el controlador de E/S para que:
    - Esté PG[1] esté conectado a la entrada EINT1 del controlador de interrupciones.
    - Se generen interrupciones (siempre que no estén enmascaradas) a flancos de bajada.

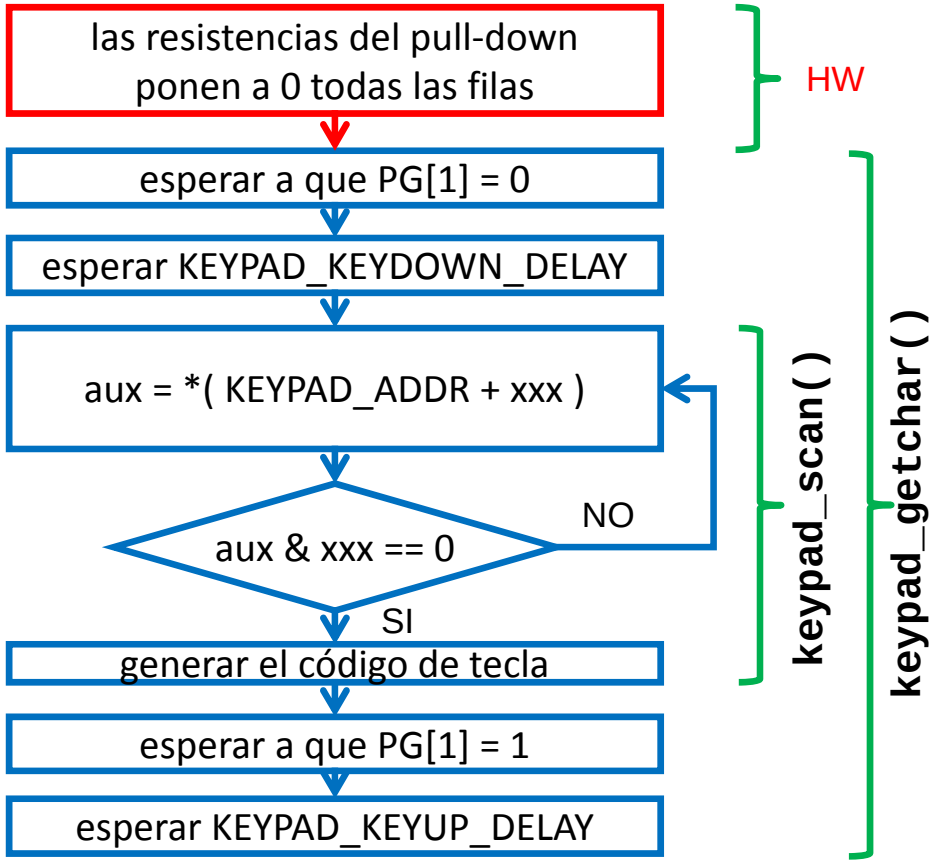
Keypad



Keypad

- El proceso de scan en esta placa supone hacer lecturas de memoria:
  - La dirección que se lea determinará el código de scan enviado al keypad
    - todas deberán corresponder al banco 3 (KEYPAD_ADDR = 0x06000000)
  - El dato leído será el código de retorno devuelto por el keypad





Driver de keypad

keypad.h

```
#ifndef __KEYPAD_H__
#define __KEYPAD_H__
...
#define KEYPAD_KEY0 (0x0)
#define KEYPAD_KEY1 (0x1)
...
#define KEYPAD_FAILURE (0xff)
#define KEYPAD_TIMEOUT (0xfe)
#define KEY_DOWN   (1)
#define KEY_UP     (0)

void keypad_init( void );
uint8 keypad_scan( void );
uint8 keypad_status( uint8 scancode );
void keypad_wait_keydown( uint8 scancode );
void keypad_wait_keyup( uint8 scancode );
void keypad_wait_any_keydown( void );
void keypad_wait_any_keyup( void );
uint8 keypad_getchar( void );
uint8 keypad_getchartime( uint16 *ms );
uint8 keypad_timeout_getchar( uint16 ms );
void keypad_open( void (*isr)(void) );
void keypad_close( void );

#endif
```

Declara macros para identificar a cada tecla

Declara macros para identificar errores durante la lectura del keypad

Declara macros para identificar el estado de una tecla

Driver de keypad

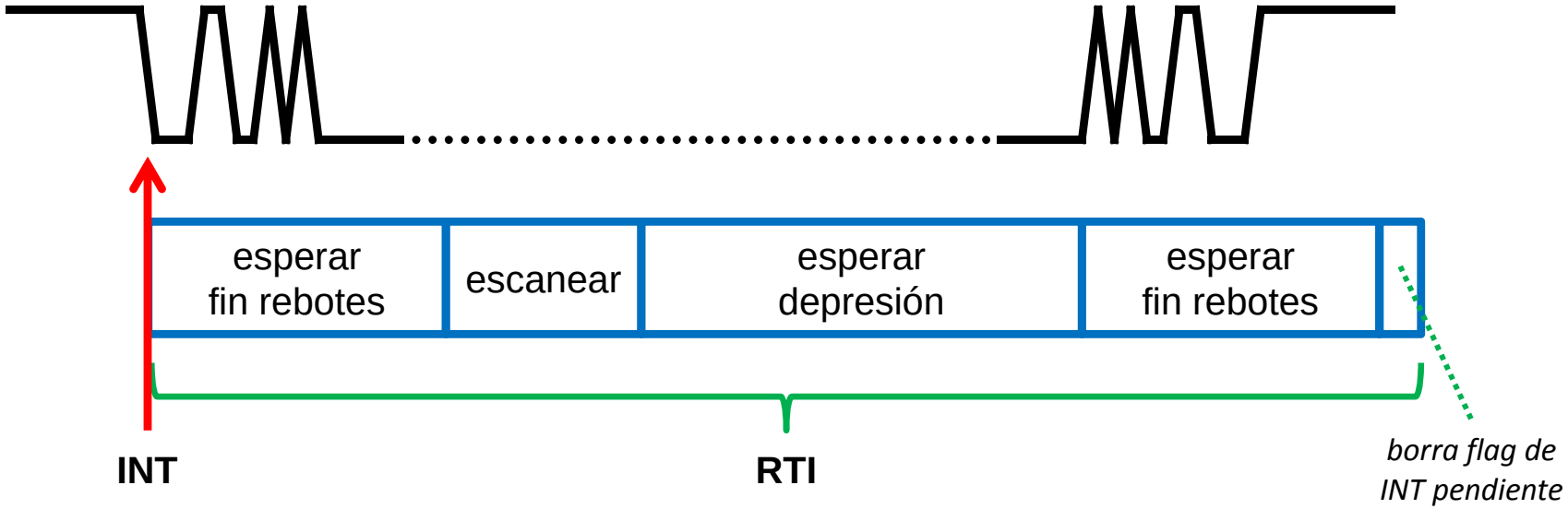
keypad.c

```
uint8 keypad_scan( void )
{
  uint8 aux;

  aux = *( KEYPAD_ADDR + 0x1c );  ····· máscara de scan: 0b00011110
  if( (aux & 0x0f) != 0x0f )  ····· comprueba si la tecla pulsada está en la fila 1
  {
    if( (aux & 0x8) == 0 )      return KEYPAD_KEY0;  ····· comprueba si está en la columna 1
    else if( (aux & 0x4) == 0 ) return KEYPAD_KEY1;  ····· comprueba si está en la columna 2
    else if( (aux & 0x2) == 0 ) return KEYPAD_KEY2;  ····· comprueba si está en la columna 3
    else if( (aux & 0x1) == 0 ) return KEYPAD_KEY3;  ····· comprueba si está en la columna 4
  }
  ...
  return KEYPAD_FAILURE;
}

void keypad_wait_keydown( uint8 scancode )
{
  while(1)
  {
    while( ... );  ····· Espera presión de
    sw_delay_ms( KEYPAD_KEYDOWN_DELAY );
    if( scancode == keypad_scan() )
      return;      } Si la tecla pulsada es la indicada, retorna
    while( ... );  ····· Si no lo es, espera depresión y vuelve a empezar
    sw_delay_ms( KEYPAD_KEYUP_DELAY );
  }
}
```

Pulsadores y keypads

evitando la espera activa (i)

- Para evitar la espera activa por pulsación, pueden usarse interrupciones
  - Desenmascarando las interrupciones por EINT6/7 (pulsadores) y/o EINT1 (keypad).
  - Instalando una RTI que realice el correspondiente filtrado de los rebotes.

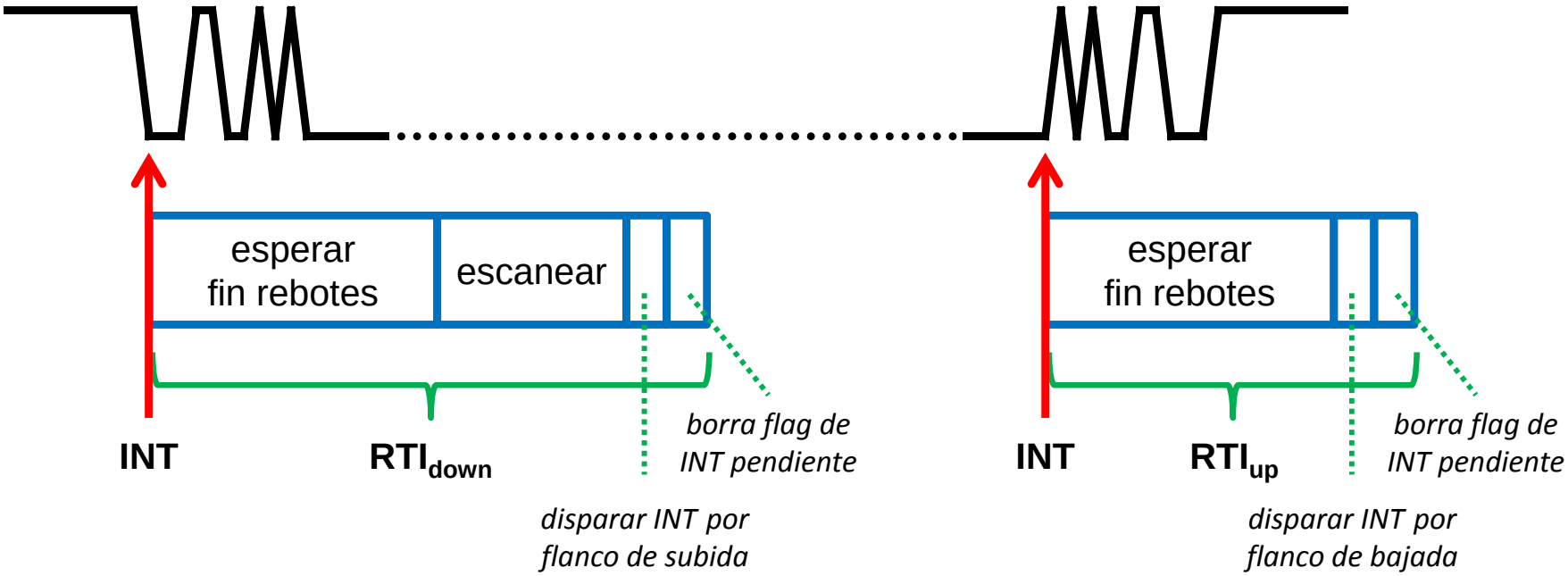


  - El flag de interrupción pendiente debe borrarse al final de la RTI:
    - Si se hace antes, los rebotes volverán a activarlo y la RTI se ejecutará 2 veces. La segunda ejecución dejará al sistema bloqueando a la espera de una depresión que ya pasó.
- Sin embargo, esta solución tiene 2 inconvenientes:
  - Si el pulsador no se suelta, el sistema queda bloqueado
  - Las esperas activas en la RTI degradan el tiempo de respuesta
    - por defecto, las interrupciones están deshabilitadas cuando se ejecuta una RTI.

Pulsadores y keypads

evitando la espera activa (ii)

- Para evitar el eventual bloqueo del sistema si no se despulsa:
  - Tanto la presión como la depresión deberán disparar interrupciones.
  - Las RTI únicamente esperarán el fin de los rebotes y cambiarán en el controlador de puertos de E/S la polaridad del flanco que dispara la siguiente interrupción.



  - El escaneo del keypad puede provocar rebotes "fantasma" adicionales en PG[1]
    - Si tras escanear una fila sin teclas pulsadas se escanea una que sí las tiene: pasa de 1 a 0.
    - Si tras una fila con teclas pulsadas se escanea una que no las tiene: pasa de 0 a 1
    - Por ello, el flag de interrupción pendiente se borra al final de la RTI

Pulsadores y keypads

evitando la espera activa (iii)

```
void keypad_init( void )
{
  EXTINT = (EXTINT & ~(0xf<<4)) | (2<<4);
  keypad_open( keypad_down_isr );
};

void keypad_down_isr( void )
{
  sw_delay_ms( KEYPAD_KEYDOWN_DELAY );
  ...se escanea el teclado y se almacena el código...
  EXTINT = (EXTINT & ~(0xf<<4)) | (4<<4);
  keypad_open( keypad_up_isr );
  I_ISPC   = BIT_KEYPAD;
}

void keypad_up_isr( void )
{
  sw_delay_ms( KEYPAD_KEYUP_DELAY );
  EXTINT = (EXTINT & ~(0xf<<4)) | (2<<4);
  keypad_open( keypad_down_isr );
  I_ISPC   = BIT_KEYPAD;
}
```

las interrupciones por EINT1 se generan a **flanco de bajada** de la señal

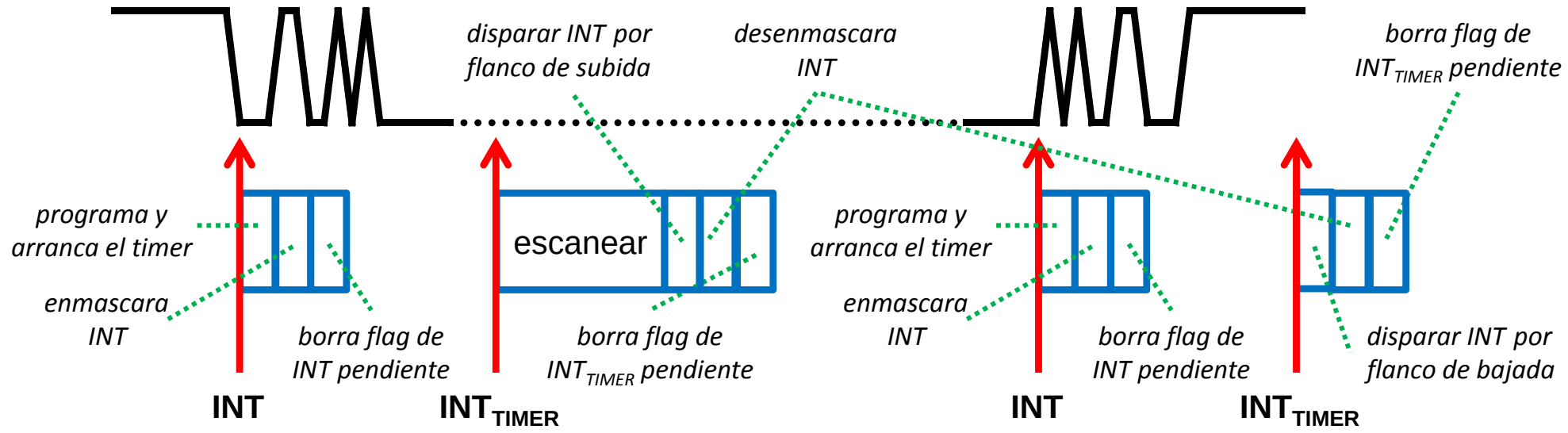
las interrupciones por EINT1 se generan a **flanco de subida** de la señal

las interrupciones por EINT1 se generan a **flanco de bajada** de la señal

Pulsadores y keypads

evitando la espera activa (iv)

- Para evitar también la espera activa por el fin de los rebotes:
  - Usar un temporizador que interrumpa pasado el tiempo de rebote.
  - La RTI de presión/depresión:
    - programa en temporizador el tiempo que debe contar y lo arranca.
    - enmascara las interrupciones por presión /depresión para evitar que los rebotes provoquen su ejecución múltiple
  - La RTI del temporizador:
    - cambia la polaridad del flanco que dispara la interrupción por pulsador
    - desenmascara las interrupciones por presión /depresión



Pulsadores y keypads

evitando la espera activa (v)

```
void keypad_init( void )
{
  EXTINT = (EXTINT & ~(0xf<<4)) | (2<<4);
  keypad_open( keypad_down_isr );
};

void keypad_down_isr( void )
{
  timer0_open_ms( timer0_down_isr, KEYPAD_KEYDOWN_DELAY, TIMER_ONE_SHOT );
  INTMSK |= BIT_KEYPAD;
  I_ISPC  = BIT_KEYPAD;
}

void timer0_down_isr( void )
{
 ...se escanea el teclado y se almacena el código..;
  EXTINT = (EXTINT & ~(0xf<<4)) | (4<<4);
  keypad_open( keypad_up_isr );
  I_ISPC = BIT_TIMER0;
}
```

las interrupciones por EINT1 se generan a **flanco de bajada** de la señal

las interrupciones por EINT1 se generan a **flanco de subida** de la señal

esta función internamente desenmascara interrupciones del keypad

Pulsadores y keypads

evitando la espera activa (vi)

```
void keypad_up_isr( void )
{
  timer0_open_ms( timer0_up_isr, KEYPAD_KEYUP_DELAY, TIMER_ONE_SHOT );
  INTMSK |= BIT_KEYPAD;
  I_ISPC  = BIT_KEYPAD;
}

void timer0_up_isr( void )
{
  EXTINT = (EXTINT & ~(0xf<<4)) | (2<<4);
  keypad_open( keypad_down_isr );
  I_ISPC = BIT_TIMER0;
}
```

las interrupciones por EINT1 se generan a **flanco de bajada** de la señal

esta función internamente desenmascara interrupciones del keypad

Tareas

1. Crear el proyecto **`lab6`** a partir de una copia de uno anterior.
2. Descargar de la Web en el directorio **`lab6`** el fichero **`lab6.c`**
3. Refrescar el proyecto **`lab6`**.
4. Descargar de la Web en el directorio **`BSP/include`** los ficheros:
   - **`timers.h`**, **`pbs.h`** y **`keypad.h`**
5. Codificar en **`BSP/source`** los ficheros:
   - **`timers.c`**, **`pbs.c`** y **`keypad.c`**
6. Refrescar el proyecto **`BSP`**
7. Compilar primero el proyecto **`BSP`** y después el proyecto **`lab6`**.
8. Crear una configuración de depuración **`lab6`** a partir de una anterior.
9. Arrancar Termite.
10. Conectar la placa y encenderla.
11. Arrancar OpenOCD.
12. Arrancar la configuración de depuración **`lab6`**