Sergio Mancha Cerezuela i Ferran Granell Jurado - G12

Sistemes Encastats. Pràctica C1. Lectura del teclat.

L'objectiu d'aquest estudi de laboratori ha estat aprendre a llegir un teclat per escanejat, així com ser capaós de llegir-lo també configurant i utilitzant interrupcions. Finalment, implementar una aplicació que ens permeti validar tots els coneixements adquirits.

1. Configuració i lectura del teclat.

En aquesta secció, configurem els ports GPIO del microcontrolador per a l'ús del teclat, així com configurar aquells ports que utilitzarem també com a *triggers* d'interrupcions. A més a més, implementem un mètode per a la lectura del teclat, que ens servirà d'aquesta manera per a verificar tant la correcte configuració com el bon funcionament del teclat.

El primer mètode implementat ha estat *void initKeyboard(void)*, que no rep ni retorna cap variable, però és l'encarregat de configurar els ports *GPIO D*, associats a la constant de *labBoard12.h* **KEY_PORT**, per a utilitzar el teclat disponible en la placa.

```
void initKeyboard(void){
  // Configuració MODER
  KEY_PORT->MODER = ((KEY_PORT->MODER) | (BIT (2*KEY_ROW1_PAD) | BIT (2*KEY_ROW2_PAD)
                    | BIT(2*KEY_ROW3_PAD) | BIT (2*KEY_ROW4_PAD)))
                    &~(BIT (2*KEY_ROW1_PAD+1) | BIT (2*KEY_ROW2_PAD+1)
                    | BIT(2*KEY_ROW3_PAD+1) | BIT(2*KEY_ROW4_PAD+1)
                    | BIT(2*KEY_COL1_PAD) | BIT(2*KEY_COL2_PAD)
                    | BIT(2*KEY_COL3_PAD)| BIT(2*KEY_COL4_PAD)
                    | BIT(2*KEY_COL1_PAD+1) | BIT(2*KEY_COL3_PAD+1)
                    | BIT(2*KEY_COL2_PAD+1) | BIT(2*KEY_COL4_PAD+1));

  // Configuració del PUPDR
  KEY_PORT->PUPDR = ((KEY_PORT->PUPDR) | (BIT (2*KEY_COL1_PAD) | BIT (2*KEY_COL2_PAD)
                    | BIT(2*KEY_COL3_PAD) | BIT (2*KEY_COL4_PAD)))
                    & ~(BIT(2*KEY_COL1_PAD+1) | BIT(2*KEY_COL2_PAD+1)
                    | BIT(2*KEY_COL3_PAD+1) | BIT(2*KEY_COL4_PAD+1));

  // Configuració del OTYPER
  KEY_PORT->OTYPER |= BIT(KEY_ROW1_PAD) | BIT(KEY_ROW2_PAD)
                    | BIT(KEY_ROW3_PAD) | BIT(KEY_ROW4_PAD);
}
```

El teclat disponible en la placa *discovery* de laboratòri té les tecles distribuïdes de manera matricial, com es correspon a la *figura 2*. Les files estan associades a les linies *PD_0* ... *PD_3*, i les columnes a *PD_6* ... *PD_9*. Per a l'ús del teclat, les columnes estan configurades com a entrades amb *pull-up*, i les files com a sortides *open drain*. Per a configurar-les com a entrades les unes i sortides les altres, hem modificat el registre *GPIO_MODER*, amb mapa associat a la *figura 1*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15		MODER14		MODER13		MODER12		MODER11		MODER10		MODER9		MODER8	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7		MODER6		MODER5		MODER4		MODER3		MODER2		MODER1		MODER0	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figura 1: Diagrama corresponent al mapa del registre *GPIO_MODER*, a partir del que es troba al *datasheet* del fabricant.

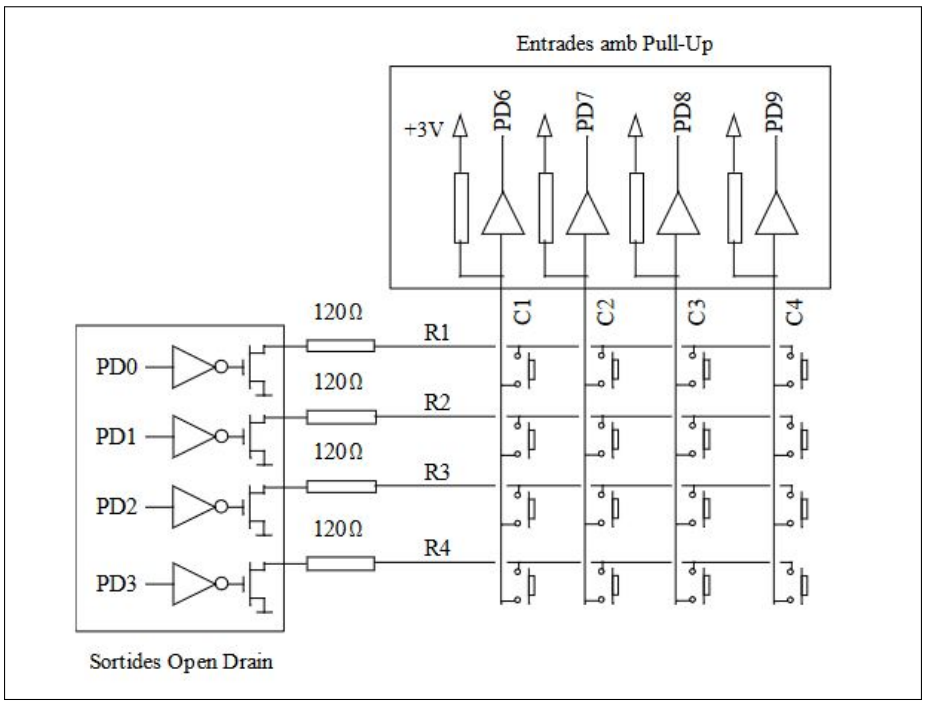


Figura 2: Diagrama corresponent a la distribució del teclat disponible a la placa del laboratòri, on d'indiquen les sortides tipus *open drain* i entrades amb *pull-up*.

El registre *GPIO_MODER* conté 16 parells de bits, *MODERX[1:0]*, associats, cadascún, a cada linia d'un port *GPIOX*, on en funció de la configuració d'aquest parell podem aconseguir que una linia sigui tipus sortida, *MODERX[1:0] = "01"*, o entrada, *MODERX[1:0] = "00"*, d'entre altres. En aquest cas, fent ús de les constants **KEY_ROWX_PAD** i **KEY_COLX_PAD**, que ens indiquen la linia associada a cada fila i columna i que es troben declarades al fitxer de capçelera *labBoard12.h*, i amb l'ús de la macro **BIT** hem aplicat una màscara bit a bit tipus *or* per a encendre el LSB bit del parell associat a les files, i amb una màscara bit a bit tipus *and* apagar el MSB, configurant així les files com a sortides.

Modificació de GPIO_MODER a partir de KEY_X_PAD

MODER2	MODER1	MODER0	
XX	XX	XX	
00	01	00	← BIT(2h) — 2 x 1h — **KEY_ROW2_PAD** 1h —
	or		
XX	X1	XX	

MODER2	MODER1	MODER0	
XX	X1	XX	
11	01	11	← BIT(3h) – 2 x 1h + 1 **KEY_ROW2_PAD** 1h —
	and		
XX	01	XX	

De la mateixa manera, per a les columnes aplicant una màscara bit a bit tipus *and*, en aquest cas per apagar ambdós bits configurant-les així com a entrades de tipus general.

A continuació, hem configurat les columnes com a entrades amb *pull-up*, modificant el registre *GPIO_PUPDR*, amb mapa associat corresponent a la *figura 3*. En aquest cas, tenim la mateixa distribució de mapa de registre, on 16 blocs controlen la configuració respecte la resistència de *pull-up* o *pull-down* de cada lina *GPIOX*. Les

configuracions trascendents d'aquest registre per a la nostre aplicació són habilitar la resistència *pull-up*, *PUPDRX[1:0] = "01"*, desactivar tant la resistència de *pull-up* i *pull-down*, *PUPDRX[1:0] = "00"*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15		PUPDR14		PUPDR13		PUPDR12		PUPDR11		PUPDR10		PUPDR9		PUPDR8	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPDR7		PUPDR6		PUPDR5		PUPDR4		PUPDR3		PUPDR2		PUPDR1		PUPDR0	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figura 3: Diagrama corresponent al mapa del registre *GPIO_PUPDR*, a partir del que es troba al *datasheet* del fabricant.

En el nostre cas, per a configurar les columnes amb resistència *pull-up*, hem aplicat una màscara bit a bit tipus *or* al registre, fent servir la constant **KEY_COLX_PAD** i la macro BIT, d'igual manera que en el cas del registre anterior però ara per a encendre el LSB, i amb una màscara tipus *and* apagar el MSB. De la mateixa manera, ens hem assegurat, tot i que el fabricant ja especifica al *datasheet* del microcontrolador que el registre després del reset torna a un estat *0x0000 0000* per al cas del port GPIO que estem configurant, que les files estaran configurades com a *no pull-up no pull-down* configurant els parells de bits a *0b00*, aplicant simplement amb el mateix procediment a partir de les constants **KEY_ROWX_PAD** una màscara tipus *and* sobre el registre.

Finalment, hem configurat les files com a *open drain*, modificant el registre *GPIO_OTYPER*, amb mapa de registre corresponent a la *figura 4*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RESERVAT															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figura 4: Diagrama corresponent al mapa del registre *GPIO_OTYPER*, a partir del que es troba al *datasheet* del fabricant.

El mapa d'aquest registre és diferent als altres, ja que en aquest cas únicament podem fer dues configuracions, o bé configurar una sortida com a *push-pull*, *OTYPERX = '0'*, o bé com a *open-drain*, *OTYPERX = '1'*. Per això, només es necessita un sol bit. Fent servir la macro BIT i les constants **KEY_ROWX_PAD**, hem aplicat una màscara tipus *or* bit a bit per a encendre els bits associats a les files del teclat, i per tant configurant-les com a sortides *open-drain*.

A continuació, hem implementat un mètode de lectura del teclat, *int32_t readKeyboard (void)*, que no rep com a argument cap paràmetre però retorna un enter de 32 bits corresponent al codi de la tecla premuda.

Com ja hem dit, la disposició del teclat que utilitzem al laboratori és matricial, on cada columna està connectada a una entrada amb resistència *pull-up*, i cada fila es correspon amb una sortida tipus *open-drain* del microcontrolador, corresponent al diagrama de la *figura 2*. L'algorisme de cerca de la tecla premuda es basa en que una vegada s'acciona una de les tecles, la sortida *open drain* associada a la seva fila es connectara amb l'entrada amb resistència *push-pull* de la columna corresponent. Si posem aquesta sortida a '0' lògic, la malla es completarà i a l'entrada mesurarem un '1', altrament si posem un '0' a la sortida *open drain*, a la columna corresponent trobarem un '0'. El diagrama de flux corresponent a la implementació d'aquest mètode és el de la *figura 5*.

```
int32_t ROWBITS[4] = {KEY_ROW1_BIT, KEY_ROW2_BIT, KEY_ROW3_BIT, KEY_ROW4_BIT},
COLUMNBITS[4] = {KEY_COL1_BIT, KEY_COL2_BIT, KEY_COL3_BIT, KEY_COL4_BIT}
```

```
int32_t readKeyboard (void) {
  int32_t row, col;
  int8_t foundFlag = 0;

  // Posem a nivell alt totes les files
  KEY_PORT->BSRR.H.set = KEY_ROW1_BIT | KEY_ROW2_BIT | KEY_ROW3_BIT | KEY_ROW4_BIT;

  row = 0;
  while ((row < 4) && (!foundFlag)) {
    KEY_PORT->BSRR.H.clear = ROWBITS[row]; // Posem a nivell baix una fila
    DELAY_US(100); // Esperem 100us per evitar error deguts al <<bouncing>>

    col = 0;
    while ((col < 4) && (!foundFlag)) {
      if (!(KEY_PORT->IDR & COLUMNBITS[col])) // Verifiquem si es tracta la tecla que és
        foundFlag = 1; // Indiquem que s'ha trobat

      col++;
    }
    KEY_PORT->BSRR.H.set = ROWBITS[row]; // Posem la fila que hem verificat a nivell alt
    row++;
  }

  if (!foundFlag) // Retornem 32 si no s'ha detectat cap tecla
    return (32);

  row--;
  col--;
  return(row*4 + col); // Retornem el codi de la tecla premuda
}

// THE END
```

En primer lloc, posem totes les files a nivell alt, mitjançant el registre *GPIO_BSRR* i les constants associades a cada linia del port *GPIOD* associada a cada fila, **KEY_ROW1_BIT** ... **KEY_ROW4_BIT** declarades al fitxer de capçelera *labBoard12.h*. A continuació, per anar verificant cada fila hem creat una estructura amb bucles tipus *while* anidats. El primer bucle es repeteix sempre que la variable *row*, que es l'encarregada de controlar quina filera evaluem [0...3], no és més gran que 3, i que *foundFlag*, la variable que controla si hem trobat o no la hipotètica tecla que s'hagi premut, no sigui certa. Per a cada iteració, posem en sortida baixa la fila que verificarem mitjançant el registre *GPIO_BSRR* i la constant associada a cada fila emmagatzemada en un vector de tipus *int32_t* que conté, en ordre creixent, les constants associades a cada fila, *int32_t ROWBITS[4]*. Posteriorment, esperem 100µs, cridant a la macro *DELAY_US* i passant com a paràmetre 100, per tal d'assegurar-nos que no es produeix cap error derivat dels efectes del *bouncing*.

A continuació, entrem en un segon *loop* que s'encarregarà de verificar columna a columna si la tecla que busquem es troba en aquesta combinació de fila i columna. Per realitzar aquesta verificació hem de llegir el valor d'entrada que reben les columnes, corresponent als camps del registre *GPIO_IDR*, el mapa de registre es correspon al diagrama de la *figura 6*. Aquest registre està conformat per 16 bits d'ús, corresponents cadascún a la lectura de cada linia d'un port *GPIOX*. En aquest cas, busquem verificar si la lectura d'una de les linies associades a les columnes del teclat es correspon amb un nivell baix. Una condició que tenim assegurada és que com únicament utilitzem les linies PD6...PD9 com a entrades, la resta de bits associats a les altres linies dins els registres es trobaran en nivell baix. A més a més, com en cada iteració ens assegurem de mantenir totes files en nivell alt, a excepció de la que estem verificant, sabem que únicament esperarem un nivell baix en aquella columna, o columnes, en que s'hagi premut una tecla, ja que altrament la resistència de *pull-up* posaria el valor a nivell alt.

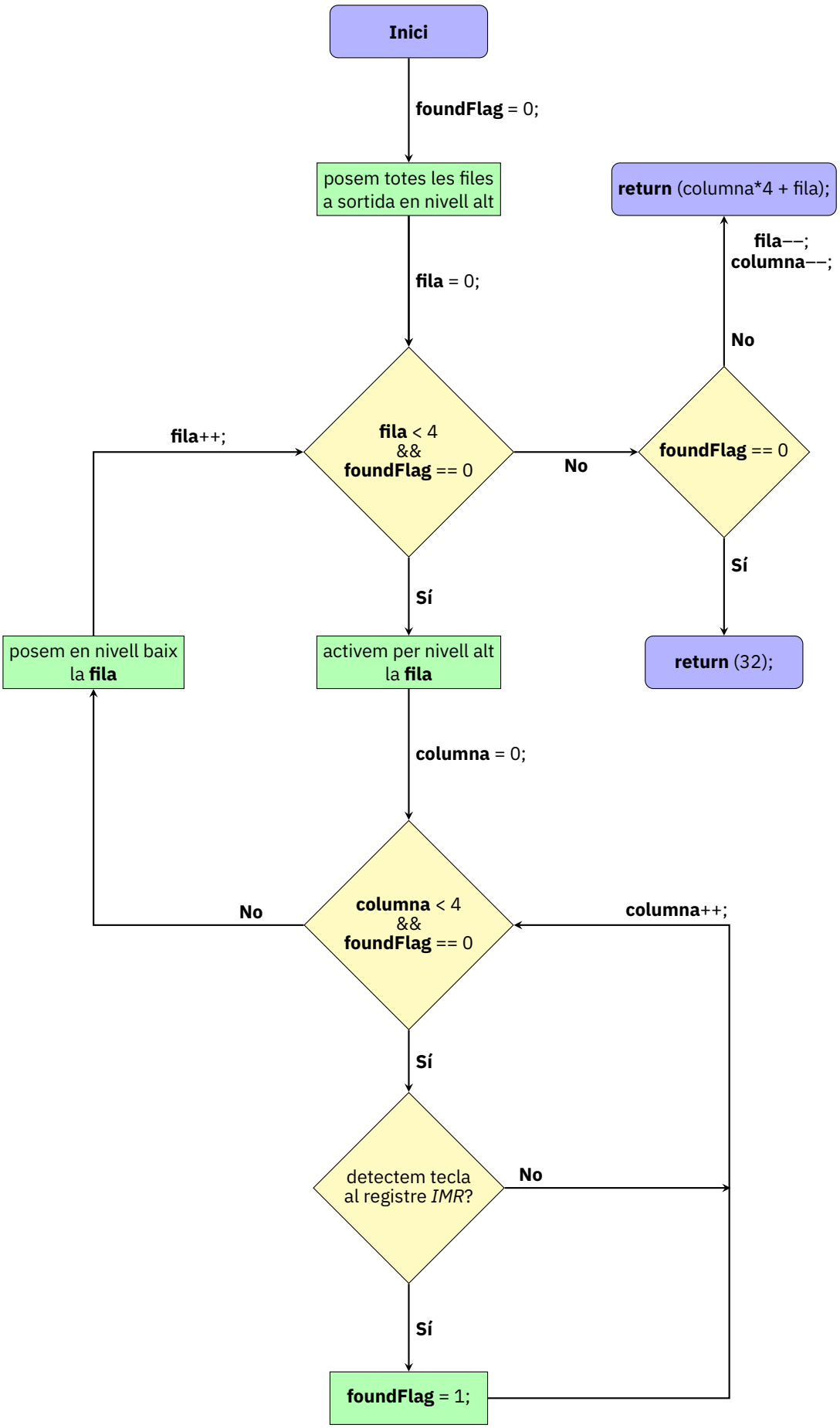


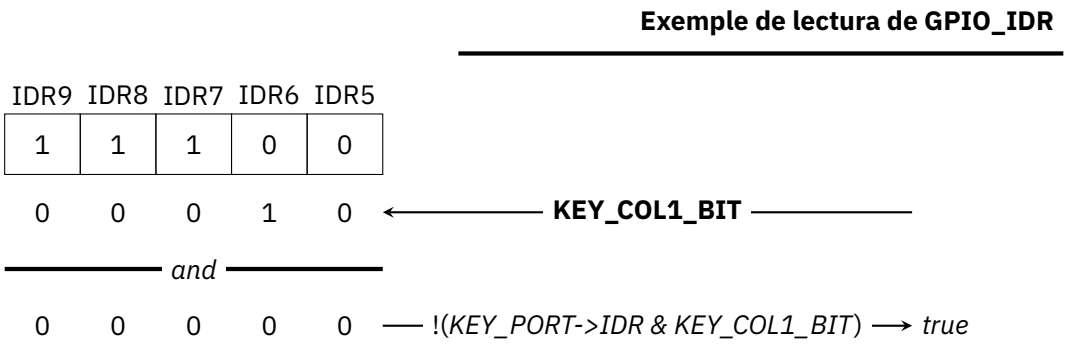
Figura 5: Diagrama de flux corresponent a la lògica implementada per a l'exploració del teclat, i la detecció d'una tecla premuda.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RESERVAT															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Figura 6: Diagrama corresponent al mapa del registre *GPIO_IDR*, a partir del que es troba al *datasheet* del fabricant.

Aplicant una màscara bit a bit tipus *and* entre el registre *GPIO_IDR* i un bit en la posició de la columna associada en aquest registre, ens hauria de retornar una mateixa cadena de bits en nivell alt en totes les posicions, menys en la que estem verificant en cas de que



En cas que es detecti la tecla, s'activa la variable que actua com a flag, *foundFlag*, i al finalitzar la iteració surtim dels bucles, per acabar retornant el codi associat a la tecla premuda. En cas contrari, s'itera sobre totes les columnes, i en acabar en cas que no s'hagi activat el flag tornariem a posar en nivell baix la fila que haviem estat verificant, i incrementant en una unitat *row* per a verificar la següent fila en la propera iteració.

Com a conveni de codis per a les tecles, hem seguit el proposat al document de la pràctica, que es correspon amb la *taula 1*.

TECLA	CODI
1	0
2	1
3	2
A	3
CAP	32

TECLA	CODI
4	4
5	5
6	6
B	7
7	8

TECLA	CODI
9	10
C	11
*	12
0	13
#	14

TECLA	CODI
D	15

Taula 1: Taula que associa les tecles del teclat amb el codi utilitzar a descodificar-les.

Finalment, per a verificar el correcte funcionament d'ambdues funcions hem implementat el següent programa a *main.c*. Inicialitzem una variable de tipus enter 32 bits, que ens servirà per anar capturant les crides al mètode *int32_t readKeyboard (void)*, i un vector estàtic de llargada 16 de variables tipus *char*, *char TECLES[16]*, que emmagatzema els caràcters de les tecles en la posició corresponent a la descodificació plantejada en la *taula 1*, així simplement podrem decodificar la lectura accedint al vector amb el valor de *readKeyboard* com a índex. Tot seguit, hem fet crida dels mètodes *void baseInit(void)*, declarat a l'arxiu de capçelera

base.h i que serveix per a inicialitzar el microcontrolador, *void LCD_Init(void)*, declarat a l'arxiu de capçelera *lcd.h* i que inicialitza la pantalla de la placa, i *void initKeyboard(void)*, el mètode que hem implementat per a la inicialització del teclat. A continuació, entrem en un bucle *infinit* que s'encarrega de cridar en cada iteració al mètode implementat *int32_t readKeyboard(void)*, i emmagatzemar en la variable *keyboardLECTURA* el valor retornat, tot això amb una latència de 10ms que forçem cridant la macro *SLEEP_MS*, passant com a argument 10. En el cas de que alguna de les lectures es correspongui amb un codi diferent a 32, és a dir que es correspongui amb el valor d'alguna tecla que ha estat premuda, aleshores una estructura condiciona s'encarrega de permetre netejar la pantalla, cridant *void LCD_ClearDisplay(void)*, i mostrar la tecla per pantalla invocant a *LCD_SendChar(TECLES[keyboardLECTURA])*.

```
#include "Base.h"
#include "lcd.h"
#include "keyboard.h"

int main(void) {
  int32_t keyboardLECTURA = 0;
  char TECLES[16] = "123A456B789C*0#D";

  baseInit(); // Inicialització bàsica
  LCD_Init(); // Inicialització del LCD
  initKeyboard(); // Inicialització del teclat

  while(1) {
    keyboardLECTURA = readKeyboard(); // Lectura del teclat

    if (keyboardLECTURA != 32){
      LCD_ClearDisplay(); // Neteja de la pantalla
      LCD_SendChar(TECLES[keyboardLECTURA]); // Mostrat del valor llegit del teclat
    }

    SLEEP_MS(10); // Pausa per a establir una latència entre lectures del teclat
  }
  return (0);
} // THE END
```

Als respectius *annexos B.1, B.2 i B.3* es poden veure els valors obtinguts en la lectura dels registres configurats mitjançant *void initKeyboard(void)*, que juntament amb la correcta lectura de les tecles observada al laboratòri ens permeten verificar el correcte funcionament del codi.

2. Configuració d'interrupcions per a la lectura del teclat.

En aquesta secció, ens encarreguem de configurar un mètode d'interrupcions a través del teclat per a llegir les tecles premudes.

El primer mètode que hem implementat ha estat *void intConfigKeyboard (void)*, que no rep ni retorna cap variable però s'encarrega de configurar les interrupcions al teclat.

```
volatile int32_t keyboardLECTURA = 0;
char TECLES[16] = "123A456B789C*0#D";

int32_t ROWBITS[4] = {KEY_ROW1_BIT, KEY_ROW2_BIT, KEY_ROW3_BIT, KEY_ROW4_BIT},
        COLUMNBITS[4] = {KEY_COL1_BIT, KEY_COL2_BIT, KEY_COL3_BIT, KEY_COL4_BIT},
        PR_KEYBOARD[4] = {EXTI_PR_PR6, EXTI_PR_PR7, EXTI_PR_PR8, EXTI_PR_PR9};

void intConfigKeyboard (void) {
  // Posem les files a nivell baix
  KEY_PORT->BSRR.H.clear = KEY_ROW1_BIT | KEY_ROW2_BIT | KEY_ROW3_BIT | KEY_ROW4_BIT;
  RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN; // Activació del clock

  // Configurem els multiplexors per les interrupcions al port D
  SYSCFG->EXTICR[1] |= SYSCFG_EXTICR2_EXTI6_PD | SYSCFG_EXTICR2_EXTI7_PD;
```

```
    SYSCFG->EXTICR[2] |= SYSCFG_EXTICR3_EXTI8_PD | SYSCFG_EXTICR3_EXTI9_PD;

    // Habilitació de les interrupcions, desemmarcarant-les
    EXTI->IMR |= EXTI_IMR_MR6 | EXTI_IMR_MR7
            | EXTI_IMR_MR8 | EXTI_IMR_MR9;

    // Interrupcions per flanc de baixada activades
    EXTI->RTSR &= ~(EXTI_RTSR_TR6 | EXTI_RTSR_TR7
            | EXTI_RTSR_TR8 | EXTI_RTSR_TR9);
    EXTI->FTSR |= EXTI_FTSR_TR6 | EXTI_FTSR_TR7
            | EXTI_FTSR_TR8 | EXTI_FTSR_TR9;

    // Netejem PR per esborrar qualsevol interrupció pendent
    EXTI->PR = EXTI_PR_PR6 | EXTI_PR_PR7 | EXTI_PR_PR8 | EXTI_PR_PR9;

    nvicEnableVector(EXTI9_5_IRQn, CORTEX_PRIORITY_MASK(STM32_EXT_EXTI5_9_IRQ_PRIORITY));

    while(1) {
      keyboardLECTURA = 32;

      while(keyboardLECTURA == 32);
      LCD_ClearDisplay();
      LCD_SendChar(TECLES[keyboardLECTURA]);
    }
}
```

Per a configurar les interrupcions sobre el teclat, hem utilitzat les linies del port *GPIOD* associades a les columnes del teclar, que hem configurat prèviament en la inicialització del teclat com a entrades.

En primer lloc, hem posat les linies associades amb les files del teclat a nivell baix. A continuació, per a configurar les interrupcions primer hem de seleccionar quines linies podran realitzar interrupcions, configurant el *System Configuration Controller*. Com aquest perifèric penja del bus perifèric *APB2* hem d'activar el seu rellotge, a través del registre *RCC_APB2ENR*. Per això, hem activat el bit associat al rellotge d'aquest perifèric aplicant una màscara tipus *or* bit a bit entre el registre i la constant **RCC_APB2ENR_SYSCFGEN**, que es correspon amb un bit encès en la mateixa posició en que es troba el bit del rellothe del *System Configuration Controller* en el registre, i ve definida a l'arxiu de capçelera *stm32f4xx.h*.

El sistema de sel·lecció de linies que permeten interrupcions funciona a través de la configuració d'uns multiplexors que agrupen les linies del tipus *PX0*...*PX15*, per tant d'un subtotal de 16 blocs de sel·lecció. Per tant, només podem habilitar el *trigger* d'interupcions en un sol tipus de linia *PXY*. Per això, es subdivideixen fins a 4 registres dins el perifèric anomentats *External Interrupt Controller X*, *SYSCFG_EXTICR_X*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RESERVAT															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTI7[3:0]				EXTI6[3:0]				EXTI5[3:0]				EXTI4[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RESERVAT															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTI11[3:0]				EXTI10[3:0]				EXTI9[3:0]				EXTI8[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figura 7: Diagrama corresponent al mapa dels registres *SYSCFG_EXTICR1* i *SYSCFG_EXTICR2*, en ordre descendent respectivament, a partir dels que es troben al *datasheet* del fabricant.

Cadascún d'aquests registres agrupa fins a 4 d'aquests multiplexors, en estricte ordre creixent, per això en el nostre cas com volem configurar interrupcions per a les linies *PD6...PD9* voldrem configurar els multiplexors *EXTI6...EXTI9*, que es troben entre els registres *SYSCFG_EXTICR_1* i *SYSCFG_EXTICR_2* amb els mapes de registre corresponents a la *figura 7*. Cada bloc ha de ser configurat amb la combinació de bits corresponent a la linia del port en concret que volem utilitzar, en el nostre cas *GPIOD*. Per això, hem aplicat sobre els respectius registres una màscara *or* bit a bit amb les constants de *stm32f4xx.h* **SYSCFG_EXTICR2_EXTI6_PD**, **SYSCFG_EXTICR2_EXTI7_PD**, **SYSCFG_EXTICR2_EXTI8_PD** i **SYSCFG_EXTICR2_EXTI9_PD**, que es corresponen amb la combinació de bits corresponent al port *GPIO D* en la posició dins el registre de cada respectiva seqüència de bits de cada multiplexor.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RESERVAT									MR22	MR21	MR20	MR19	MR18	MR17	MR16
									rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR5	MR4	MR3	MR2	MR1	MR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figura 8: Diagrama corresponent al mapa de registre *EXTI_IMR*, a partir del que es troba al *datasheet* del fabricant.

Una vegada sel·leccionades les linies que podran generar interrupcions, les hem habilitat mitjançant el *Interrupt Mask Register*, *EXTI_IMR*, del *External Interrupt Controller*, mapa de registre corresponent a la *figura 8*. Per a això, simplement hem de posar a nivell alt els bits corresponents a les posicions associades a les linies *PX6...PX9*, amb una màscara tipus *or* bit a bit entre el registre i les constants **EXTI_IMR_MR6**, **EXTI_IMR_MR7**, **EXTI_IMR_MR8** i **EXTI_IMR_MR9**, declarades al fitxer *stm32f4xx.h* i que contenen un bit a nivell alt en la posició de cada linia dins el registre.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RESERVAT									TR22	TR21	TR20	TR19	TR18	TR17	TR16
									rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figura 9: Diagrama corresponent al mapa de registre *EXTI_FTSR*, a partir del que es troba al *datasheet* del fabricant.

A continuació, hem configurat el mode d'interrupció per a que s'activin per flanc de baixada. Per això, hem configurat els registres *Falling Trigger Selection Register*, *EXTI_FTSR* i diagrama del seu mapa en la *figura 9*, i *Rising Trigger Selection Register*, *EXTI_RTSR*. Per a aquesta configuració, hem aplicat una màscara tipus *and* bit a bit entre el registre de *rising trigger* i les constants negades **EXTI_RTSR_TR6**, **EXTI_RTSR_TR7**, **EXTI_RTSR_TR8** i **EXTI_RTSR_TR9**, declarades al fitxer *stm32f4xx.h* i que contenen un bit actiu en la posició associada a la linia a configurar dins el registre, per a apagar els bits del registre associats a les interrupcions per *PD6...PD9*, i en conseqüència deshabilitar les interrupcions per flanc de pujada. Per contra, hem aplicat una màscara tipus *or* bit a bit entre el registre de *falling trigger* i les constants **EXTI_FTSR_TR6**, **EXTI_FTSR_TR7**, **EXTI_FTSR_TR8** i **EXTI_FTSR_TR9**, també declarades al fitxer de capçelera *stm32f4xx.h* i que contenen un bit en nivell alt en la posició associada a la linia dins el registre, per a activar l'accionament per flanc de baixada en les linies d'interrupció *PD6...PD7*.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RESERVAT									PR22	PR21	PR20	PR19	PR18	PR17	PR16
									rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PR15	PR14	PR13	PR12	PR11	PR10	PR9	PR8	PR7	PR6	PR5	PR4	PR3	PR2	PR1	PR0
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

Figura 10: Diagrama corresponent al mapa de registre *EXTI_PR*, a partir del que es troba al *datasheet* del fabricant.

Posteriorment, hem esborrat el *pending register*, *EXTI_PR* amb diagrama de memòria associat a la *figura 10*, per a eliminar qualsevol interrupció pendent i així assegurar-nos que no es genera cap interrupció amb les que configurem ara.

Finalment, per a programar els registres interns del *nested vectored interrupt controller* hem cridat a *nvicEnableVector(Linia, CORTEX_PRIORITY_MASK(Prioritat))*, passant com a paràmetre de linia la constant **EXTI9_5_IRQn**, que es correspon amb les linies GPIO de la 5 a la 9, la constant **STM32_EXT_EXTI5_9_IRQ_PRIORITY** com a prioritat.

Una vegada configurades les interrupcions, hem implementat un bucle tipus *while* de repetició infinita per a controlar el mostrat per pantalla de les tecles premudes capturades a partir de l'accionament d'una interrupció. Per això, hem definit una variable global entera de 32 bits *int32_t keyboardLECTURA*, que serà modificada per el RSI que implementarem posteriorment en accionar-se una interrupció a través del teclat. Predeterminadament, hem fixat el valor de la variable a 32, que es correspon amb el codi de no haver premut cap tecla. El bucle es quedarà esperant fins que el RSI no modifiqui la variable amb la nova sel·lecció, i una vegada això aleshores simplement netejem la pantalla cridant el mètode *LCD_ClearDisplay()*, declarat a *lcd.h*. Per a descodificar la tecla i poder mostrar-la com un caràcter, ens hem fet ajut d'un vector de *chars* que hem inicialtizat de manera global que conté els 16 símbols ordenats pel seu codi com a ordre d'indexació, així simplement utilitzant la pròpia variable *keyboardLECTURA* podem descodificar la lectura del teclat, i passar-ho com a argument al mètode *void LCD_SendChar(char)* per a visualitzar la tecla en pantalla.

Per acabar, hem implementat la macro de la funció de RSI, *CH_IRQ_HANDLER (EXTI9_5_IRQHandler)*. El diagrama de la *figura 11* ve a il·lustrar la lògica implementada.

En primer lloc, com el vector que estem utilitzant per a configurar el *nested vectored interrupt* contempla interrupcions accinades des de linies *PX5...PX9*, però únicament volem implementar un RSI per a interrupcions accionades en les columnes del teclat, això és els multiplexors configurats per a les linies *PD6...PD9*. Per a això, amb una estructura condicional, ens assegurem que la interrupció s'hagi produït únicament pel teclat accedint al *pending register*, *EXTI_PR*, i aplicant sobre ell una màscara tipus *and* bit a bit conformada per les constants **EXTI_PR_PR6**, **EXTI_PR_PR7**, **EXTI_PR_PR8** i **EXTI_PR_PR9**, que contenen cadascuna un bit en nivell alt en la posició de cada corresponent linia al *pending register* i que es troben declarades al *stm32f4xx.h*. D'aquesta manera, en cas de que el resultat d'aplicar aquesta màscara sigui diferent a 0 implicarà que, efectivament, s'ha produït una interrupció on ens interessa.

A continuació, ens assegurem que no es pugui produïr cap nova interrupció fins que no acabem de gestionar l'actual emmascarant-les posant a nivell baix les posicions associades a les linies de les columnes al *Interrupt Mask Register*, *EXTI_IMR*. Per això, apliquem una màscara tipus *and* bit a bit entre el registre i una cadena conformada per bits en nivell baix en les posicions que volem emmascarar al registre, i alts altrament. Per a aconseguir aquesta cadena, simplement neguem amb l'operador *not* bit a bit la seqüència conformada pels bits a nivell alt resultants de la unió de les constants **EXTI_IMR_MR6**, **EXTI_IMR_MR6**, **EXTI_IMR_MR6** i **EXTI_IMR_MR6**, que contenen un bit en nivell alt en la posició associada a la seva respectiva linia dins el registre d'emmascarat d'interrupcions i que es troben declarades al *stm32f4xx.h*.

L'objectiu d'aquesta funció RSI és capturar la tecla premuda al teclat a partir d'una interrupció. A diferència del mètode implementat anteriorment per a aquesta mateixa acció, en aquest cas ja comptem amb informació a priori, la columna que ha accionat la interrupció.

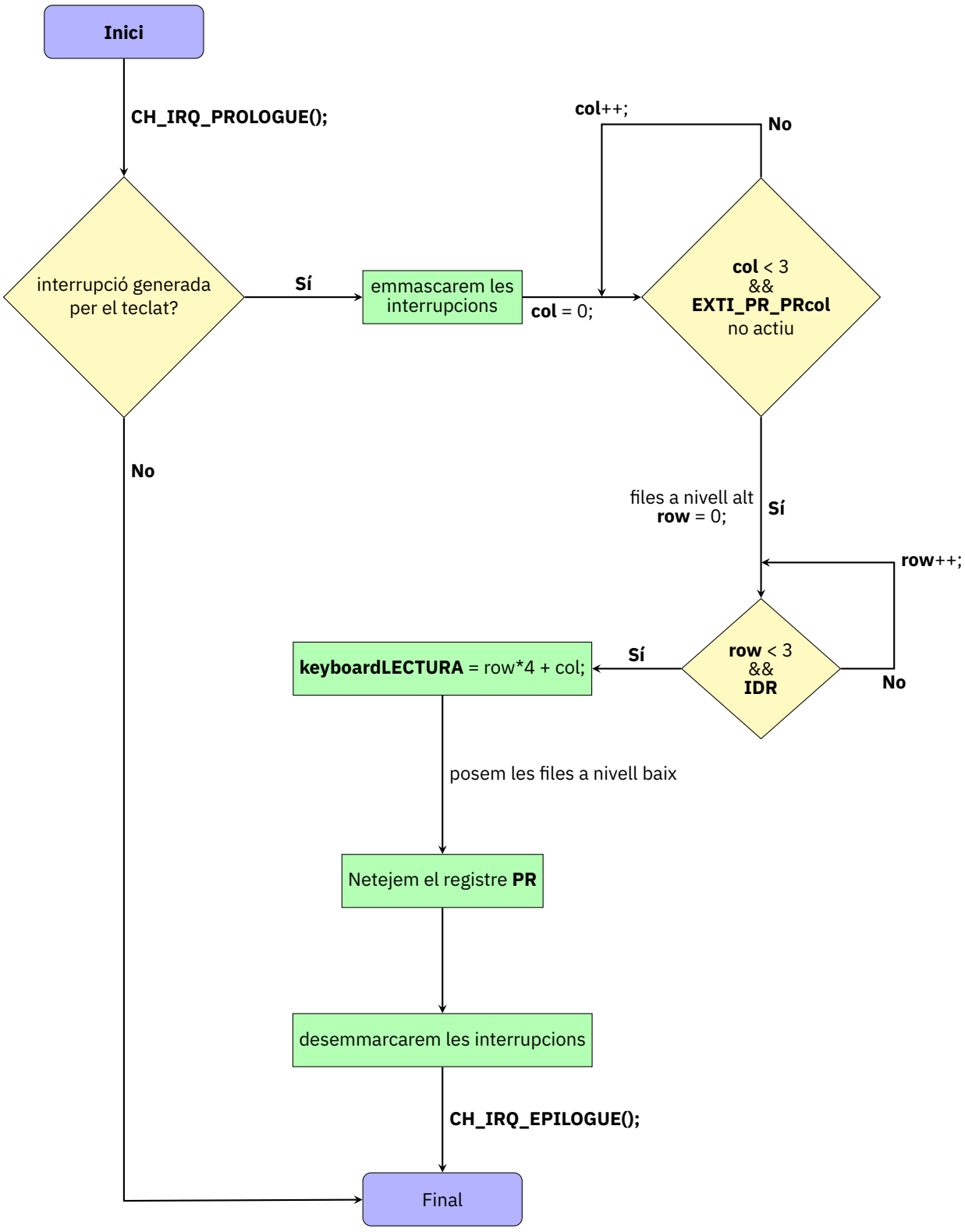


Figura 11: Diagrama de flux corresponent a la lògica implementada per a l'exploració del teclat, i la detecció d'una tecla premuda.

Per a trobar l'índex associat a aquesta columna, entenent que aquestes estan ordenades de la 0 a la 3 en ordre *PD6...PD9*, hem declarat un vector d'enters de 32 bits *int32_t PR_KEYBOARD[4]* que contindrà, en estricte ordre creixen d'indexació, les constants **EXTI_PR_PR6...EXTI_PR_PR9** i implementat un bucle tipus *while*. Aquest *loop* s'encarrega d'incrementar la variable entera de control *col* en una unitat cada vegada que no es detecti una interrupció activada en la posició del *pending register* corresponent a *PR_KEYBOARD[col]*, mitjançant una màscara *and* bit a bit entre el pròpi registre i la constant associada a la posició de la columna en cada iteració. Això sí, sempre que la variable de control sigui mes petita que 3, ja que en qualsevol cas com hem assegurat que s'haurà produït una interrupció al teclat en última instància aquesta haurà estat activada, si no per cap de les altres, per l'última columna.

Una vegada capturada la columna, busquem detectar la fila per a poder determinar quina ha estat la tecla accionada. El procediment que hem utilitzat ha estat el mateix que en el cas del primer mètode implementat, *int32_t readKeyboard (void)*. En primer lloc ens assegurem de deixar les files a nivell alt, mitjançant el registre *GPIO_BSRR*. A continuació, un bucle tipus *while* s'encarregarà d'incrementar en una unitat la variable entera de control *int32_t row* cada vegada que no es detectiu un nivell baix en la posició de la columna premuda dins el regitre *Input Data Register*, *GPIO_IDR*, on en el cas més restrictiu si es correspongués a la darrera fila. Per això, a més a més en cada iteració ens assegurarem, abans d'incrementar la variable, de tornar a posar a nivell alt la fila que acabem de verificar, i deixar en nivell baix la nova fila per a la verificació en la consegüent iteració, en ambdós casos mitjançant el registre *GPIO_BSRR* i el bit del vector amb les constants associades al bit de cada columna en el registre *int32_t COLUMNBITS[4]*. D'aquesta manera, al finalitzar el *loop*, i tornant a deixar en nivell alt la darrera fila que haviem verifiacat darrerament, tindriem les posicions associades tant a la columna i fila, que es traudeixen en la tecla mitjançant $4 \cdot row + col$. Aquest valor el sobreescriurem sobre la variable global *int32_t keyboardLECTURA*, que és la que s'utilitza al mètode implementat per a la configuració i ús de la interrupció.

Darrerament, per a concloure l'RSI simplement netejem el *pending register* per a elminar la interrupció que s'havia generat, i les desemmascarem per a habilitar l'accionament de noves interrupcions.

```
CH_IRQ_HANDLER (EXTI9_5_IRQHandler) {
  CH_IRQ_PROLOGUE();
  // Inici del RSI

  int32_t row, col; // Variables de control

  if ((EXTI->PR&(EXTI_PR_PR6|EXTI_PR_PR7|EXTI_PR_PR8|EXTI_PR_PR9)) != 0) {
    // Emmascararem les interrupcions
    EXTI->IMR &= ~(EXTI_IMR_MR6 | EXTI_IMR_MR7 | EXTI_IMR_MR8 | EXTI_IMR_MR9);

    ////////////////////////////////
    // CÀLCUL DE LA TECLA PREMUDA

    col= 0;
    while((col < 3) && ((EXTI->PR & PR_KEYBOARD[col]) == 0)){
      col++;
    }

    row = 0;
    KEY_PORT->BSRR.H.set = KEY_ROW1_BIT | KEY_ROW2_BIT | KEY_ROW3_BIT | KEY_ROW4_BIT;

    KEY_PORT->BSRR.H.clear = ROWBITS[row];
    DELAY_US(100); // Esperem per evitar problemes amb el <<bouncing>>
    while((row < 3) && ((KEY_PORT->IDR & COLUMNBITS[col]) != 0)){
      KEY_PORT->BSRR.H.set = ROWBITS[row];
      row++;

      KEY_PORT->BSRR.H.clear = ROWBITS[row];
      DELAY_US(100); // Esperem per evitar problemes amb el <<bouncing>>
    } KEY_PORT->BSRR.H.set = ROWBITS[row];

    keyboardLECTURA = row*4 + col;

    // Files a 0
    KEY_PORT->BSRR.H.clear = KEY_ROW1_BIT | KEY_ROW2_BIT | KEY_ROW3_BIT | KEY_ROW4_BIT;

    ////////////////////////////////

    EXTI->PR = EXTI_PR_PR6 | EXTI_PR_PR7 | EXTI_PR_PR8 | EXTI_PR_PR9;

    // Desemmascarem les interrupcions
    EXTI->IMR |= EXTI_IMR_MR6 | EXTI_IMR_MR7 | EXTI_IMR_MR8 | EXTI_IMR_MR9;
  }
```

```
  CH_IRQ_EPILOGUE();

  // Final de l'RSI
}
```

Per a verificar les funcions implementades, hem implementat el següent programa al *main.c*. En primer lloc, incluim les descripcions dels arxius de capçelera *Base.h*, *lcd.h* i *keyboard.h*, per a poder fer ús dels seus mètodes. Dins del programa principal, i abans de res, inicialitzem la placa cridant el mètode *void baseInit(void)*, implementat a *Base.c*, inicialitzem el *display*, invocant el mètode *void LCD_Init(void)* implementada a *lcd.c*, i inicialitzem el teclat, cridant la funció implementada *void initKeyboard(void)* a *keyboard.c*. Posteriorment, per a fer ús de la lectura per interrupcions invoquem el mètode implementat per a la configuració i ús d'interrupcions al teclat, *void intConfigKeyboard(void)*.

```
#include "Base.h"
#include "lcd.h"
#include "keyboard.h"

int main(void) {
  baseInit(); // Inicialització bàsica
  LCD_Init(); // Inicialització del LCD
  initKeyboard(); // Inicialització del teclat

  intConfigKeyboard();
  return (0);
}

// THE END
```

Després de compilar i carregar el programa al microcontrolador, hem pogut verificar el correcte funcionament del mètode d'interrupcions per teclat per a la detecció de les tecles premudes. En primer lloc, pausant l'execució del programa i refrescant la lectura dels registres al *EmbSys Registers*, hem observat les configuracions dels registres *EXTI_IMR*, *EXTI_RTSR*, *EXTI_FTSR* i *EXTI_PR*, corresponent als *annexos B.4, B.5, B.6 i B.7* en el mateix ordre. Com aquests es corresponen amb el que esperàvem, concluim que aquesta primera part s'ha implementat correctament.

Hem verificat el funcionament del restant del programari mitjançant l'accionament del teclat. Hem pogut comprovar la correcte lectura en el display de les tecles, tot i que produïnt-se en ocasions efectes derivats, pel soroll del teclat, del «*bouncing*» l'accionament d'interrupcions després de deixar premer alguna de les tecles. Bo i això, com els resultats eren els que esperàvem, donem per verificat els mètodes implementats al complet.

3. Lectura de múltiples tecles simultàniament.

En aquesta secció, hem implementat una extensió del mètode *int32_t readKeyboard (void)* per a realitzar una lectura simultània de diverses tecles.

El mètode implementat ha estat *int16_t readMultiKey (void)*, que no rep cap variable com a paràmetre però retorna una cadena de 16 bits que es correspondrà amb les tecles premudes simultàniament al teclat.

```
int16_t readMultiKey (void) {
  int16_t row = 0, col = 0, keysREAD = 0;

  // Posem a nivell alt les files.
  KEY_PORT->BSRR.H.set = KEY_ROW1_BIT | KEY_ROW2_BIT | KEY_ROW3_BIT | KEY_ROW4_BIT;

  for (row = 0; row < 4; row++) {
    KEY_PORT->BSRR.H.clear = ROWBITS[row];
    DELAY_US(100);
```

```
    for(col = 0; col < 4; col++) {
    //Retornem el codi per a que aparegui com està proposat al document de pràctiques
      if (!(KEY_PORT->IDR & COLUMNBITS[col]))
        keysREAD |= 1<<(row*4+col);

    } KEY_PORT->BSRR.H.set = ROWBITS[row];

  } return (keysREAD);
}
```

El codi implementat és una simple modificació a partir del mètode ja implementat per a la lectura d'una única tecla, amb la salvetat de que en aquest cas en comptes de fer servir bucles tipus *while* hem optat per tipus *for*, aconseguint així un codi més compacte, degut a que per a detectar totes les tecles premudes en un instant al teclat hem d'iterar necessàriament per totes les files, i per tant també les columnes al registre *GPIO_IDR*, abans d'acabar el mètode. En aquesta implementació, també, en comptes de retornar la tecla en forma d'un codi, activem el bit corresponent a la posició de la tecla fent servir l'ordre establert a la *taula 1*. Això ho aconseguim simplement inicialitzant a l'inici del mètode una cadena de 16 bits a $0x0000$, en forma d'un enter *int16_t*, i aplicant en cada detecció de la tecla una màscar tipus *or* bit a bit sobre ella amb un bit en la posició $4 \cdot \text{row} + \text{col}$, que en essència es correspon al codi que retornàvem atnerioment ja que el pròpi codi fonamentalment es tracta de la posició en la matriu del teclat.

Per a la verificació de la funció, hem implementat el següent programari al mètode principal de *main.c*. En aquest, hem inclòs els fitxers de capçelera *Base.h*, *lcd.h* i *keyboard.h*, per a poder fer ús dels seus mètodes. Inicialitzem el *display*, invocant el mètode *void LCD_Init(void)* implementada a *lcd.c*, i inicialitzem el teclat, cridant la funció implementada *void initKeyboard(void)* a *keyboard.c*.

```
#include "Base.h"
#include "lcd.h"
#include "keyboard.h"

int main(void) {
  int16_t multiREADING = 0, i;
  char TECLES[16] = "123A456B789C*0#D";

  baseInit(); // Inicialització bàsica
  LCD_Init(); // Inicialització del LCD
  initKeyboard(); // Inicialització del teclat

    while(1) {
        multiREADING = readMultiKey(); // lectura del teclat

        // Actualització dels valors en pantalla
        LCD_ClearDisplay();
        for(i = 0; i<16; i++) {
            if ((multiREADING & 1))
                LCD_SendChar(TECLES[i]);

            multiREADING = multiREADING >> 1;
        }
    }

  return (0);
}

// THE END
```

Fent una crida en un bucle tipus *while true* del mètode implementat, *int16_t readMultiKey (void)*, i emmagatzemante el valor de retorn en cada iteració en *int16_t multiREADING*, aconseguim únicament haver de decodificar la cadena de bits per a representar les tecles premudes en pantalla. Aquesta descodifiació l'obtenim a partir d'un bucle tipus *for*, amb el qual, prèviament havent netejat la pantalla, anem verificant si el LSB es troba en nivell alt, fent un desplaçament en cada iteració cap a la dreta per anar col·locant cadascún dels bits en la posició del LSB. En cas de cert, LSBi = '1', prenent el valor d'i com el codi de la tecla a mostrar.

Hem pogut observar experimentalment els efectes produïts al premer tecles que uneixen diverses de les sortides *open drain* amb més d'una columna, on el mètode llegeix tecles que realment no han estat premudes. Bo i això, com els resultats observats han estats els esperats, donem per validat el codi implementat.

4. Implementació d'una calculadora.

En aquesta darrera secció, implementem tots els mètodes que han estat necessàris per a desenvolupar una aplicació tipus calculadora, amb la capaçitat d'operar tant amb operands negatius com amb decimals per a les operacions fonamentals.

El primer mètode que hem implementat ha estat una modificació del *char *itoa(int32_t num, char *str, int32_t radix)*, per a en aquest cas treballar amb valors tipus *float*. El mètode que hem implementat l'hem anomenat *char *ftoa(float num, char *str)*, que rep com a paràmetres un punter a tipus *char*, un string en definitiva, que serà la cadena de caràcters de sortida i un número tipus *float* que serà el convertir, i que retorna també el número en una variable punter a *char*.

```
char *ftoa(float num, char *str) {
  int32_t sign = 0;  // Signe
  int32_t pos = 0;   // Variable de posició al string
  int32_t radix = 10; // Radix fixat a decimal

  // Variable d'ús general
  int32_t i;

  // Busquem el signe
  if (num < 0.0) {
    sign = 1;
    num *= -1;
  }

  int32_t integer_part = (int32_t)num,
          decimal_part = ((int32_t)(num * PRECISIO_DECIMAL) % PRECISIO_DECIMAL);

  // Construcció la part decimal del string
  do {
  i = decimal_part % radix;
  if (i < 10)
    str[pos] = i + '0';
  else
    str[pos] = i - 10 + 'A';

  pos++;
  decimal_part /= radix;
  } while (decimal_part > 0);

  str[pos] = '.';
  pos++;

  // Construcció la part entera del string
  do {
  i = integer_part % radix;
  if (i < 10)
    str[pos] = i + '0';
  else
    str[pos] = i - 10 + 'A';

  pos++;
  integer_part /= radix;
  } while (integer_part > 0);

  // Afegim el signe
```

```
  if (sign)
    str[pos] = '-';
  else
    pos--;

  // Afegim el NULL al final
  str[pos+1]=0;

  // Capgirem el string
  i=0;
  do {
  sign=str[i];
  str[i++]=str[pos];
  str[pos--]=sign;

  } while(i < pos);

  return (str);
}
```

La funció implementada únicament considerarà la representació decimal de la xifra, pel que hem decidit prescindir del paràmetre *radix* present en le mètode original. Hem capturat el signe del flotant, i posteriorment dividit en dues variables enteres la part pròpiament entera del número i la decimal, aplicant *type casting*.

type var2 = (type)var1;

Per a la part entera, aprofitant que simplement es correspondria a un truncament, aplicant *type casting* per a un enter de 32 bits aconseguim aïllar-la. Per la part decimal, però, hem implementat un sistema que ens permet precissar la resolució que volem. Això és, agafant el flotant d'entrada, multiplicant-lo per 10^n, on n és la *precisió decimal* que volem, i prenent el mòdul respecte de 10^n. Aplicant ara *type casting* per a un enter de 32 bits, aconseguim convertir la part decimal amb la precisió que voliem com a un enter. Posteriorment, reutilitzant el codi implementat a *itoa*, simplement començem a construir el string a partir de la part decimal, afegim el símbol '.' com a indicador de la coma, i finalment completem amb la part entera i el signe. Finalment, es reverteix el string i es retorna el seu valor.

El segon mètode implementat ha estat *void padNumDecoding(int32_t key_num, int8_t * value), que no retorna cap valor però requereix com a paràmetres un enter de 32 bits, corresponent al codi d'una tecla premuda, i un punter a un enter de 8 bits que retornarà el valor de la tecla decodificat.*

```
void padNumDecoding(int32_t key_num, int8_t * value) {
  switch(key_num) {
  case 0:
  case 1:
  case 2:
    *value = key_num + 1;
    break;
  case 4:
  case 5:
  case 6:
    *value = key_num;
    break;
  case 8:
  case 9:
  case 10:
    *value = key_num - 1;
    break;
  case 12:
    *value = -2; // '*' separador de decimals
    break;
  case 13:
```

```
      *value = 0;
      break;
    case 15:
      *value = -1; // '-1' signe negatiu
      break;
    default:
      *value = 32; // Caràcter no vàlid
      break;
    }
}
```

El mètode consisteix fonamentalment d'una implementació de la descodifiació de la *taula 2*, les interaccions de les tecles en mode d'operacions, mitjançant una estructura tipus *switch-case*.

TECLA	FUNCIÓ
NUM	NUM
#	OK
*	separador decimal
D	singe negatiu
ALTRES	no vàlid

Taula 2: Taula que relaciona les funcionalitats de les tecles en qualsevol dels modes d'operació de la calculadora.

El tercer mètode implementat ha estat *void readOP(float * op, char name)*, serà el mètode encarregat de llegir un dels operadors d'una de les operacions sel·leccionades per l'usuari. No retorna cap valor, però requereix com a paràmetres un punter a flotant, que serà el valor de l'operand llegit, i un tipus *char*, que es correspondrà amb la lletra de l'operand.

```
void readOP(float * op, char name) {
  char string[2];
  int8_t value = 0;

  // Variables controladores d'estats, signe i decimal.
  int8_t signFlag = 0, decFlag = 0, decCounter = 1;

  LCD_SendChar(name);
  LCD_SendString(" = ");
  KEY_READ = readKeyboard();
  while((TECLES[KEY_READ] != '#') && (*op <= 99999)) {
    padNumDecoding(KEY_READ, &value);

    if(value != 32) {
      SLEEP_MS(100); // Latència entre la lectura i l'aparició en pantalla

      if (value == -2){ // INTERACCIÓ AMB ELS DECIMALS
        if (!decFlag) {
          decFlag = 1;
          LCD_SendChar('.');

        }
      } else if (value == -1) { // INTERACCIÓ AMB EL SIGNE
        if ((!signFlag) && (*op == 0.0)) {
          signFlag = 1;

          LCD_SendChar('-');
```

```
            }
        } else { // INTERACCIÓ GENÈRICA, AFEGIR UN NÚMERO
            if (!decFlag) {
                if ((10*(*op) + value) <= NUM_THRESHOLD){
                    *op = 10*(*op) + value;

                    LCD_SendString(itoa(value, string, 10));
                }
            } else {
                *op = (*op) + ((float)value)/((float)(10*decCounter));

                LCD_SendString(itoa(value, string, 10));
                decCounter++;
            }
        }
    }

    KEY_READ = readKeyboard();
  }

  if(signFlag)
    *op *= -1;
}
```

En un primer lloc, el mètode s'encarrega de mostrar per pantalla el nom de l'operand, una lletra que es correspondrà amb o bé 'A' o bé 'B', i conseqüentment un símbol d'igualtat, '=', a l'espera de que l'usuari hi introdueixi contingut mitjançant el teclat. En un bucle tipus *while*, anem afegint contingut a l'operador, punter a *op*, sempre que no s'hagi premut l'OK, o bé el valor de l'operador mai sigui major a 99999, és a dir no sigui un número de més de 5 xifres. Això, però, és una qüestió purament deguda a la pantalla que tenim, ja que més endavant per a representar el resultat de l'operació ens volem assegurar que, per un costat, l'operació càpiga en una fila i, per l'altre costat, el resultat en una altre. En una iteració, es descodifica la lectura del teclat mitjançant una invocació a *void padNumDecoding(int32_t key_num, int8_t * value)*, i en cas de que el caràcter sigui vàlid, és a dir que sigui diferent de 32, aleshores s'accedeix a un dels modes d'edició del valor de l'operand en funció de la tecla premuda.

- **Interacció amb els decimals.** Aquest mode s'activa sempre que es premi la tecla '*', i la primera vegada és l'encarregat d'activar el flag dels decimals, per a forçar el *mode decimal* a l'hora d'afegir números del teclat al valor de *op*.
- **Interacció amb el signe.** Aquest mode s'activa sempre que es premi la tecla 'D', i en el cas d'activar-se abans d'introduïr cap valor aleshores s'activa el flag de signe, *signFlag*.
- **Interacció amb els números ò genèrica. Mode part entera.** Aquest mode s'activa quan s'introdueix un número per teclat, i el flag dels decimals no es troba activat. Afegeix la xifra com a part entera al final del tot.
- **Interacció amb els números ò genèrica. Mode part decimal.** Aquest mode s'activa quan s'introdueix un número per teclat, i el flag dels decimals es troba activat. En aquest cas, afegeix la xifra com a decimal al final de tot.

Abans de finalitzar la iteració, al sortir del condicional es torna a llegir el teclat i es repteix tot seguidament. Al finalitzar el *loop*, es modifica la condició de signe amb el flag de signe, *signFlag*, multiplicant el valor de *op* per un factor de (-1) en el cas que s'hagi activat.

Els quarts mètodes implementats han estat, *void readOPERATORS(float * opA, float * opB)*, que no retorna cap valor però s'encarregarà de llegir els operadors d'una de les operacions sel·leccionades, demanant com a paràmetres precisament dos puntes a valors flotants corresponents als valors de cada operador. També hem implementat, *void RESULTAT(float * opA, float * opB, float * opRES, char opSIM)*, que s'encarrega de mostrar el resultat de l'operació demanant com a paràmetres punters a flotants corresponents a valors dels operadands i del resultat, i un tipus *char* corresponent al símbol de l'operació efectuada.

```
void readOPERATORS(float * opA, float * opB){
  LCD_ClearDisplay(); // Neteja de la pantalla

  readOP(opA, 'A'); // Llegim el primer operand

  SLEEP_MS(100); // Latència entre OK i segon operand
  LCD_GotoXY(0, 1); // Desplaçem el cursor a l'inici de la segona fila per al segon operand
  readOP(opB, 'B'); // Llegim el segon operand
}

void RESULTAT(float * opA, float * opB, float * opRES, char opSIM){
  char string[16]; // string per al ftoa, llargada màxima igual a la longitud de la pantalla

  LCD_ClearDisplay(); // Neteja de pantalla

  // Títol del resultat
  LCD_SendString("  [ RESULTAT ]  ");

  SLEEP_MS(500); // Pausa entre el títol del resultat i el pròpi resultat

  LCD_ClearDisplay(); // Neteja de pantalla
  LCD_SendString(ftoa(*opA, string, 10)); // Mostrat del priemr operand
  LCD_SendChar(opSIM); // Mostrat de la operació efectuada
  LCD_SendString(ftoa(*opB, string, 10)); // Mostrat del segon operand
  LCD_SendString(" =");

  LCD_GotoXY(0, 1); // Apuntem amb el cursor a l'inici de la segona fila
  LCD_SendString("= ");
  LCD_SendString(ftoa(*opRES, string, 10)); // Mostrem el resultat de l'operació
}
```

Els cinquens mètodes implementats han estat els que hem anomenat com a tipus *operacions*, on simplement es tracta de variacions del mateix mètode particularitzats a cada tipus d'operacions. Les funcions han estat, *void calcSUM(void)*, *void calcMULT()* i *void calcDIV()*.

```
void calcSUM() {
  float A = 0, B = 0, RES = 0; // Operands i resultat

  // Sel·lecció operació per pantalla
  LCD_ClearDisplay();
  LCD_SendString("    [ SUMA ]    ");
  LCD_GotoXY(0, 1);
  LCD_SendString("     A + B      ");

  SLEEP_MS(500); // Retenim el contingut 0.5s

  readOPERATORS(&A, &B); // Llegim els operands

  RES = A + B; // Efectuem l'operació

  RESULTAT(&A, &B, &RES, '+'); // Mostrem el resultat

  // Esperem a que l'usuari premi '#' per tornar al menú de la calculadora
  KEY_READ = readKeyboard();
  while(TECLES[KEY_READ] != '#'){
    KEY_READ = readKeyboard();

  }SLEEP_MS(100); // Latència entre botó premut i acció
}

void calcMULT() {
  float A = 0, B = 0, RES = 0; // Operands i resultat
```

```
    // Sel·lecció operació per pantalla
    LCD_ClearDisplay();
    LCD_SendString("    [ MULT ]    ");
    LCD_GotoXY(0, 1);
    LCD_SendString("     A * B      ");

    SLEEP_MS(500); // Retenim el contingut 0.5s

    readOPERATORS(&A, &B); // Llegim els operands

    RES = A * B; // Efectuem l'operació

    RESULTAT(&A, &B, &RES, '*'); // Mostrem el resultat

    // Esperem a que l'usuari premi '#' per tornar al menú de la calculadora
    KEY_READ = readKeyboard();
    while(TECLES[KEY_READ] != '#'){
      KEY_READ = readKeyboard();

    }SLEEP_MS(100); // Latència entre botó premut i acció
}

void calcDIV() {
    float A = 0, B = 0, RES = 0; // Operands i resultat

    // Sel·lecció operació per pantalla
    LCD_ClearDisplay();
    LCD_SendString("  [ DIVISIO ]   ");
    LCD_GotoXY(0, 1);
    LCD_SendString("     A / B      ");

    SLEEP_MS(500); // Retenim el contingut 0.5s

    readOPERATORS(&A, &B); // Llegim els operands

    RES = A / B; // Efectuem l'operació

    RESULTAT(&A, &B, &RES, '/'); // Mostrem el resultat

    // Esperem a que l'usuari premi '#' per tornar al menú de la calculadora
    KEY_READ = readKeyboard();
    while(TECLES[KEY_READ] != '#'){
      KEY_READ = readKeyboard();

    }SLEEP_MS(100); // Latència entre botó premut i acció
}
```

El sisè mètode implementat ha estat *void calcMULT(void)*, que no retorna ni demana cap paràmetre, però és l'encarregat de gestionar el contigut del menú tipus «scroll» d'informació en la calculadora.

```
void info(void){
    int level = 0;

    // Misstage títol de la secció
    LCD_ClearDisplay();
    LCD_SendString("    [ INFO ]    ");
    LCD_GotoXY(0, 1);
    LCD_SendString("A-B:SCR");

    KEY_READ = readKeyboard(); // Lectura teclat
```

```
  while(TECLES[KEY_READ] != '#'){ // Bucle de la secció per a implementar el menú
    if(KEY_READ != 32){ // Verifiquem que s'ha premut algún caràcter vàlid
      SLEEP_MS(100); // Latència per a fer més suau els desplaçaments

      // PUJAR O BAIXAR
      if (KEY_READ == 3){
        level = (level-1);

        if(level < 0)
          level = 3;

      } else if (KEY_READ == 7) {
        level = (level+1)%4;

      }

      // ACTUALITZACIÓ PANTALLA
      LCD_ClearDisplay();
      switch(level){
        case 1:
          LCD_SendString("[OP INFO]");
          LCD_GotoXY(0, 1);
          LCD_SendString("(D) -1 || (*)DEC");
          break;
        case 2:
          LCD_SendString("[DESC]");
          LCD_GotoXY(0, 1);
          LCD_SendString("EMB CALC");
          break;
        case 3:
          LCD_SendString("[AUTORS]");
          LCD_GotoXY(0, 1);
          LCD_SendString("Sergio & Ferran");
          break;
        default:
          LCD_SendString("    [ INFO ]    ");
          LCD_GotoXY(0, 1);
          LCD_SendString("A-B:SCR");
          break;
      }

    }
    KEY_READ = readKeyboard(); // Lectura nova del teclat
  }
}
```

Aquest mètode implementa un menú tipus «scroll» circular, on apareixen les seccions de l'apartat info de la calculadora, controlat per pujar cap a amunt amb la tecla 'A' i per baixar amb la tecla 'B'. La implementació ha estat la mateixa que en el cas del mètode principal, pel que l'explicació és extrapolable.

Darrerament, hem implementat el mètode principal *void calculadora (void)*. En un primer moment, mostra per pantalla un títol de l'aplicació, i fins que no es prem OK, '#', no comença l'aplicació *per se*. Una vegada s'interacciona amb la pantalla d'inici, apareix un menú de tipus «scroll» que conté totes les interaccions possibles amb la calculadora, les tres opcions d'operacions més un d'informació. El desplaçament es controla mitjançant les tecles 'A' i 'B', a partir d'una variable de control de nivell circular que s'incrementa en una unitat si es prem 'B', és a dir baixem pel menú, o bé decrementa en una unitat si es prem 'A'. Aquestes interaccions es controlen mitjançant una lectura del teclat al principi de cada iteració d'un bucle tipus *while true*, execució infinita o fins que es força l'apagat o reset del microcontrolador, on s'invoca el mètode *int32_t readKeyboard (void)* que actualitza la variable de lectura *volatile int32_t KEY_READ*.

En una estructura tipus *switch* es consideren totes les interaccions possibles en el menú principal, on les tecles 1-4 sel·leccionen alguna de les quatre funcionalitats de la calculadora, i per tant invoquen als seus

respectius mètodes associats, i tant pujar com baixar apliquen els seus respectius increments i decrements sobre la variable de control *level*. Finalment, abans de repetir la iteració es neteja la pantalla i s'actualitza amb el contigut corresponent al valor de *level*, mitjançant una altre estructura tipus *switch*.

```
volatile int32_t KEY_READ = 0;

void calculadora (void){
  int8_t level = 0;

  // Missatge de benvinguda
  LCD_ClearDisplay();
  LCD_SendString(" Calculadora v1 ");
  LCD_GotoXY(0, 1);
  LCD_SendString("    Prem (#)    ");

  // Bucle principal
  while(1){
    KEY_READ = readKeyboard(); // Lectura del teclat

    if(KEY_READ != 32){ // Verifiquem que s'hagi premut alguna tecla realment.
      SLEEP_MS(100); // latència en el menú, per suavitzar els desplaçaments

      switch(KEY_READ){
        case 0: // (1) SUMAR
          calcSUM();
          break;
        case 1: // (2) MULTIPLICAR
          calcMULT();
          break;
        case 2: // (3) DIVIDIR
          calcDIV();
          break;
        case 3: // (A) PUJAR
          level = (level-1);

          if(level < 0)
            level = 2;
          break;
        case 4: // (4) INFO
          info();
          break;
        case 7: // (B) BAIXAR
          level = (level+1)%3;
          break;
      }

      // Actualitzem contingut en pantalla
      LCD_ClearDisplay();
      switch(level){
        case 1:
          LCD_SendString("(1) A + B");
          LCD_GotoXY(0, 1);
          LCD_SendString("(2) A * B");
          break;
        case 2:
          LCD_SendString("(3) A / B");
          LCD_GotoXY(0, 1);
          LCD_SendString("(4) INFO");
          break;
        default:
          LCD_SendString("    [ MENU ]    ");
```

```
            LCD_GotoXY(0, 1);
            LCD_SendString("A-B:SCR 1-4:SEL");
            break;
        }
      }
    }
}
```

Per acabar, hem implementat el següent programari en el mètode principal a *main.c*. Fent una crida de *void baseInit(void)*, declarat a l'arxiu de capçelera *base.h* i que serveix per a inicialitzar el microcontrolador, *void LCD_Init(void)*, declarat a l'arxiu de capçelera *lcd.h* i que inicialitza la pantalla de la placa, i *void initKeyboard(void)*, el mètode que hem implementat per a la inicialització del teclat. A continuació simplement invocant el mètode principal de la calculadora *void calculadora(void)* aconseguim fer còrrer la aplicació en el microcontrolador.

```
#include "Base.h"
#include "lcd.h"
#include "keyboard.h"

int main(void) {
  baseInit(); // Inicialització bàsica
  LCD_Init(); // Inicialització del LCD
  initKeyboard(); // Inicialització del teclat

  LCD_Config(1, 0, 0); // Configuració del LCD per a que el cursor no molesti

  calculadora(); // Mètode principal aplicació calculadora
  return (0);
}
```

A partir dels tests fets a laboratòri, hem pogut verificar el correcte funcionament de totes les funcionalitats de la calculadora implementada, i per tant validem el codi desenvolupat.

Annexos.

Annex A. Fitxer de capçelera *keyboard.h*

```
/*
 * keyboard.h
 *
 *  Created on: Apr 16, 2024
 *      Author: Sergio Mancha
 */

#ifndef KEYBOARD_H_
#define KEYBOARD_H_

#define PRECISIO_DECIMAL 100 // 2 decimals
#define NUM_THRESHOLD 9999

// Inicialització del teclat
void initKeyboard(void);

// Funció per la lectura del teclat
int32_t readKeyboard (void);

// Funció per a la gestió de les interrupcions a través del teclat
void intConfigKeyboard(void);

// Funció per a la lectura simultània de diverses tecles
int16_t readMultiKey (void);

// Implementació d'una calculadora
void calculadora (void);

// Mètodes implementats per a l'ús de la calculadora
void padNumDecoding(int32_t key_num, int8_t * value);

void readOP(float * op, char name);
void readOPERATORS(float * opA, float * opB);

void RESULTAT(float * opA, float * opB, float * opRES, char opSIM);

void calcSUM(void);
void calcMULT(void);
void calcDIV(void);
void info(void);

char *ftoa(float num, char *str, int32_t radix);

#endif /* KEYBOARD_H_ */

// THE END
```

Annex B. Resultats experimentals, configuracions dels registres.

GPIOD						Port D
GPIOD_MODER	0x55000155	01010101000000000000000101010101	0x00000000	RW	0x40020C00	GPIO port mode register
MODER0 (bits 0-1)	0x1	01				General purpose output mode
MODER1 (bits 2-3)	0x1	01				General purpose output mode
MODER2 (bits 4-5)	0x1	01				General purpose output mode
MODER3 (bits 6-7)	0x1	01				General purpose output mode
MODER4 (bits 8-9)	0x1	01				General purpose output mode
MODER5 (bits 10-11)	0x0	00				Input (reset state)
MODER6 (bits 12-13)	0x0	00				Input (reset state)
MODER7 (bits 14-15)	0x0	00				Input (reset state)
MODER8 (bits 16-17)	0x0	00				Input (reset state)
MODER9 (bits 18-19)	0x0	00				Input (reset state)
MODER10 (bits 20-21)	0x0	00				Input (reset state)
MODER11 (bits 22-23)	0x0	00				Input (reset state)
MODER12 (bits 24-25)	0x1	01				General purpose output mode
MODER13 (bits 26-27)	0x1	01				General purpose output mode
MODER14 (bits 28-29)	0x1	01				General purpose output mode
MODER15 (bits 30-31)	0x1	01				General purpose output mode

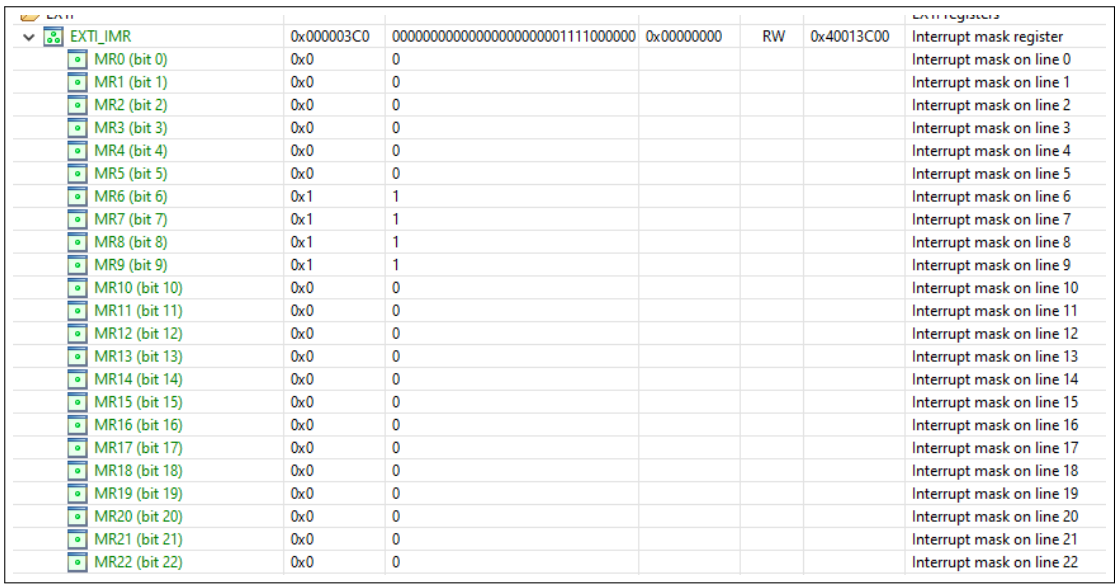
Annex B.1. Configuració dels bits registre GPIO_MODER al mètode void initKeyboard(void).

GPIOD_PUPDR	0x00555055	00000000010101010101000001010101	0x00000000	RW	0x40020C0C	GPIO port pull-up/pull-down register
PUPDR0 (bits 0-1)	0x1	01				Pull-up
PUPDR1 (bits 2-3)	0x1	01				Pull-up
PUPDR2 (bits 4-5)	0x1	01				Pull-up
PUPDR3 (bits 6-7)	0x1	01				Pull-up
PUPDR4 (bits 8-9)	0x0	00				No pull-up, pull-down
PUPDR5 (bits 10-11)	0x0	00				No pull-up, pull-down
PUPDR6 (bits 12-13)	0x1	01				Pull-up
PUPDR7 (bits 14-15)	0x1	01				Pull-up
PUPDR8 (bits 16-17)	0x1	01				Pull-up
PUPDR9 (bits 18-19)	0x1	01				Pull-up
PUPDR10 (bits 20-21)	0x1	01				Pull-up
PUPDR11 (bits 22-23)	0x1	01				Pull-up
PUPDR12 (bits 24-25)	0x0	00				No pull-up, pull-down
PUPDR13 (bits 26-27)	0x0	00				No pull-up, pull-down
PUPDR14 (bits 28-29)	0x0	00				No pull-up, pull-down
PUPDR15 (bits 30-31)	0x0	00				No pull-up, pull-down

Annex B.2. Configuració dels bits registre GPIO_PUPDR al mètode void initKeyboard(void).

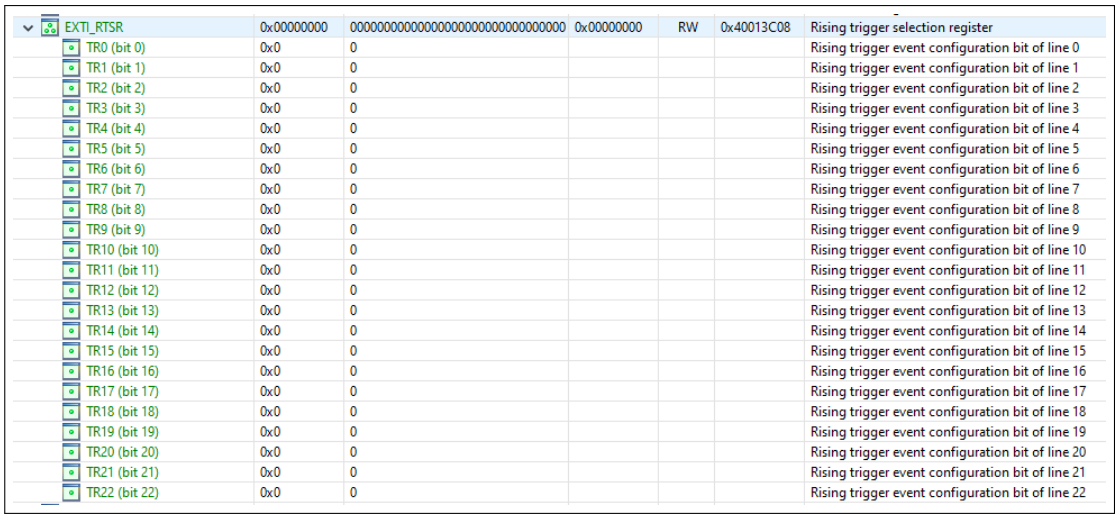
GPIOD_OTYPER	0x0000000F	00000000000000000000000000001111	0x00000000	RW	0x40020C04	GPIO port output type register
OT0 (bit 0)	0x1	1				Output open-drain
OT1 (bit 1)	0x1	1				Output open-drain
OT2 (bit 2)	0x1	1				Output open-drain
OT3 (bit 3)	0x1	1				Output open-drain
OT4 (bit 4)	0x0	0				Output push-pull (reset state)
OT5 (bit 5)	0x0	0				Output push-pull (reset state)
OT6 (bit 6)	0x0	0				Output push-pull (reset state)
OT7 (bit 7)	0x0	0				Output push-pull (reset state)
OT8 (bit 8)	0x0	0				Output push-pull (reset state)
OT9 (bit 9)	0x0	0				Output push-pull (reset state)
OT10 (bit 10)	0x0	0				Output push-pull (reset state)
OT11 (bit 11)	0x0	0				Output push-pull (reset state)
OT12 (bit 12)	0x0	0				Output push-pull (reset state)
OT13 (bit 13)	0x0	0				Output push-pull (reset state)
OT14 (bit 14)	0x0	0				Output push-pull (reset state)
OT15 (bit 15)	0x0	0				Output push-pull (reset state)

Annex B.3. Configuració dels bits registre GPIO_OTYPER al mètode void initKeyboard(void).



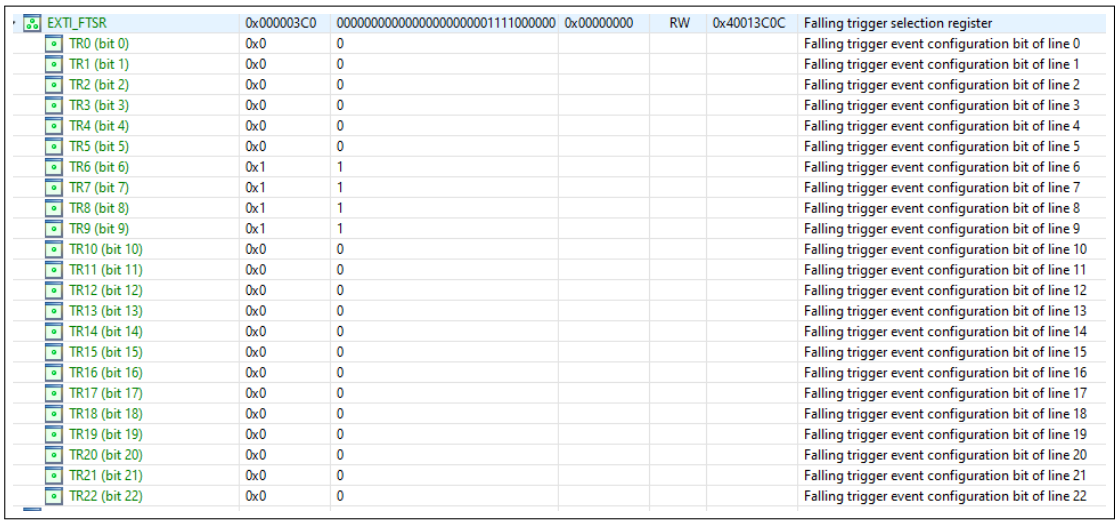
EXTI_IMR	0x000003C0	00000000000000000000001111000000	0x00000000	RW	0x40013C00	Interrupt mask register
MR0 (bit 0)	0x0	0				Interrupt mask on line 0
MR1 (bit 1)	0x0	0				Interrupt mask on line 1
MR2 (bit 2)	0x0	0				Interrupt mask on line 2
MR3 (bit 3)	0x0	0				Interrupt mask on line 3
MR4 (bit 4)	0x0	0				Interrupt mask on line 4
MR5 (bit 5)	0x0	0				Interrupt mask on line 5
MR6 (bit 6)	0x1	1				Interrupt mask on line 6
MR7 (bit 7)	0x1	1				Interrupt mask on line 7
MR8 (bit 8)	0x1	1				Interrupt mask on line 8
MR9 (bit 9)	0x1	1				Interrupt mask on line 9
MR10 (bit 10)	0x0	0				Interrupt mask on line 10
MR11 (bit 11)	0x0	0				Interrupt mask on line 11
MR12 (bit 12)	0x0	0				Interrupt mask on line 12
MR13 (bit 13)	0x0	0				Interrupt mask on line 13
MR14 (bit 14)	0x0	0				Interrupt mask on line 14
MR15 (bit 15)	0x0	0				Interrupt mask on line 15
MR16 (bit 16)	0x0	0				Interrupt mask on line 16
MR17 (bit 17)	0x0	0				Interrupt mask on line 17
MR18 (bit 18)	0x0	0				Interrupt mask on line 18
MR19 (bit 19)	0x0	0				Interrupt mask on line 19
MR20 (bit 20)	0x0	0				Interrupt mask on line 20
MR21 (bit 21)	0x0	0				Interrupt mask on line 21
MR22 (bit 22)	0x0	0				Interrupt mask on line 22

Annex B.4. Configuració dels bits registre EXTI_IMR al mètode void intConfigKeyboard (void).



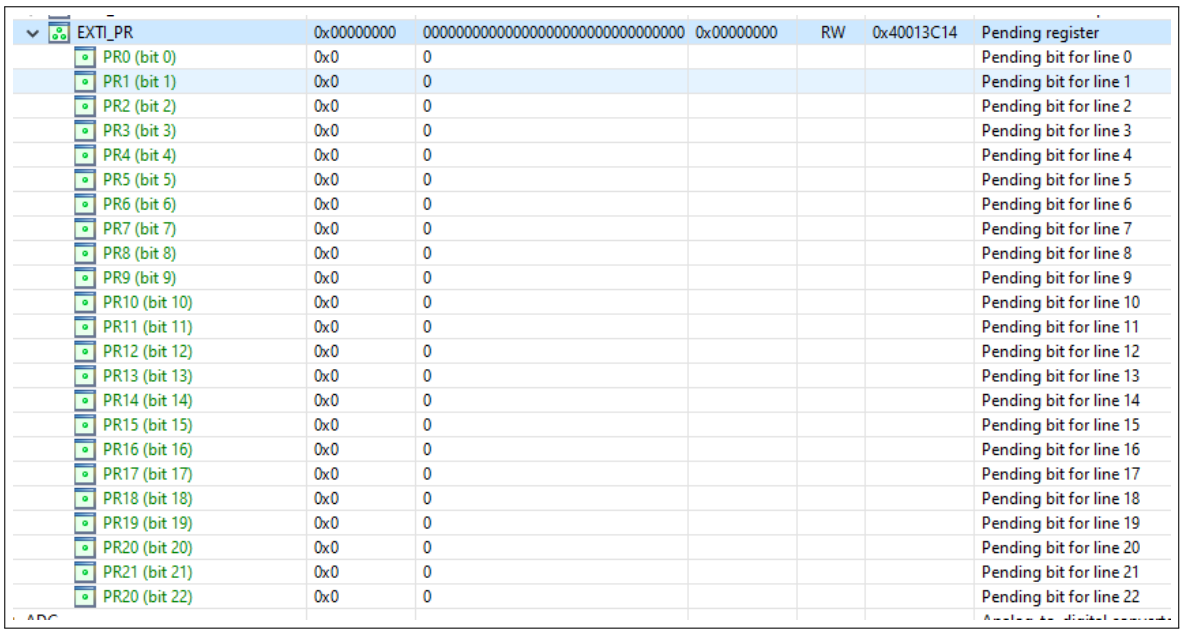
EXTI_RTSR	0x00000000	00000000000000000000000000000000	0x00000000	RW	0x40013C08	Rising trigger selection register
TR0 (bit 0)	0x0	0				Rising trigger event configuration bit of line 0
TR1 (bit 1)	0x0	0				Rising trigger event configuration bit of line 1
TR2 (bit 2)	0x0	0				Rising trigger event configuration bit of line 2
TR3 (bit 3)	0x0	0				Rising trigger event configuration bit of line 3
TR4 (bit 4)	0x0	0				Rising trigger event configuration bit of line 4
TR5 (bit 5)	0x0	0				Rising trigger event configuration bit of line 5
TR6 (bit 6)	0x0	0				Rising trigger event configuration bit of line 6
TR7 (bit 7)	0x0	0				Rising trigger event configuration bit of line 7
TR8 (bit 8)	0x0	0				Rising trigger event configuration bit of line 8
TR9 (bit 9)	0x0	0				Rising trigger event configuration bit of line 9
TR10 (bit 10)	0x0	0				Rising trigger event configuration bit of line 10
TR11 (bit 11)	0x0	0				Rising trigger event configuration bit of line 11
TR12 (bit 12)	0x0	0				Rising trigger event configuration bit of line 12
TR13 (bit 13)	0x0	0				Rising trigger event configuration bit of line 13
TR14 (bit 14)	0x0	0				Rising trigger event configuration bit of line 14
TR15 (bit 15)	0x0	0				Rising trigger event configuration bit of line 15
TR16 (bit 16)	0x0	0				Rising trigger event configuration bit of line 16
TR17 (bit 17)	0x0	0				Rising trigger event configuration bit of line 17
TR18 (bit 18)	0x0	0				Rising trigger event configuration bit of line 18
TR19 (bit 19)	0x0	0				Rising trigger event configuration bit of line 19
TR20 (bit 20)	0x0	0				Rising trigger event configuration bit of line 20
TR21 (bit 21)	0x0	0				Rising trigger event configuration bit of line 21
TR22 (bit 22)	0x0	0				Rising trigger event configuration bit of line 22

Annex B.5. Configuració dels bits registre EXTI_RTSR al mètode void intConfigKeyboard (void).



EXTI_FTSR	0x000003C0	00000000000000000000001111000000	0x00000000	RW	0x40013C0C	Falling trigger selection register
TR0 (bit 0)	0x0	0				Falling trigger event configuration bit of line 0
TR1 (bit 1)	0x0	0				Falling trigger event configuration bit of line 1
TR2 (bit 2)	0x0	0				Falling trigger event configuration bit of line 2
TR3 (bit 3)	0x0	0				Falling trigger event configuration bit of line 3
TR4 (bit 4)	0x0	0				Falling trigger event configuration bit of line 4
TR5 (bit 5)	0x0	0				Falling trigger event configuration bit of line 5
TR6 (bit 6)	0x1	1				Falling trigger event configuration bit of line 6
TR7 (bit 7)	0x1	1				Falling trigger event configuration bit of line 7
TR8 (bit 8)	0x1	1				Falling trigger event configuration bit of line 8
TR9 (bit 9)	0x1	1				Falling trigger event configuration bit of line 9
TR10 (bit 10)	0x0	0				Falling trigger event configuration bit of line 10
TR11 (bit 11)	0x0	0				Falling trigger event configuration bit of line 11
TR12 (bit 12)	0x0	0				Falling trigger event configuration bit of line 12
TR13 (bit 13)	0x0	0				Falling trigger event configuration bit of line 13
TR14 (bit 14)	0x0	0				Falling trigger event configuration bit of line 14
TR15 (bit 15)	0x0	0				Falling trigger event configuration bit of line 15
TR16 (bit 16)	0x0	0				Falling trigger event configuration bit of line 16
TR17 (bit 17)	0x0	0				Falling trigger event configuration bit of line 17
TR18 (bit 18)	0x0	0				Falling trigger event configuration bit of line 18
TR19 (bit 19)	0x0	0				Falling trigger event configuration bit of line 19
TR20 (bit 20)	0x0	0				Falling trigger event configuration bit of line 20
TR21 (bit 21)	0x0	0				Falling trigger event configuration bit of line 21
TR22 (bit 22)	0x0	0				Falling trigger event configuration bit of line 22

Annex B.6. Configuració dels bits registre EXTI_FTSR al mètode void intConfigKeyboard (void).



EXTI_PR	0x00000000	00000000000000000000000000000000	0x00000000	RW	0x40013C14	Pending register
PR0 (bit 0)	0x0	0				Pending bit for line 0
PR1 (bit 1)	0x0	0				Pending bit for line 1
PR2 (bit 2)	0x0	0				Pending bit for line 2
PR3 (bit 3)	0x0	0				Pending bit for line 3
PR4 (bit 4)	0x0	0				Pending bit for line 4
PR5 (bit 5)	0x0	0				Pending bit for line 5
PR6 (bit 6)	0x0	0				Pending bit for line 6
PR7 (bit 7)	0x0	0				Pending bit for line 7
PR8 (bit 8)	0x0	0				Pending bit for line 8
PR9 (bit 9)	0x0	0				Pending bit for line 9
PR10 (bit 10)	0x0	0				Pending bit for line 10
PR11 (bit 11)	0x0	0				Pending bit for line 11
PR12 (bit 12)	0x0	0				Pending bit for line 12
PR13 (bit 13)	0x0	0				Pending bit for line 13
PR14 (bit 14)	0x0	0				Pending bit for line 14
PR15 (bit 15)	0x0	0				Pending bit for line 15
PR16 (bit 16)	0x0	0				Pending bit for line 16
PR17 (bit 17)	0x0	0				Pending bit for line 17
PR18 (bit 18)	0x0	0				Pending bit for line 18
PR19 (bit 19)	0x0	0				Pending bit for line 19
PR20 (bit 20)	0x0	0				Pending bit for line 20
PR21 (bit 21)	0x0	0				Pending bit for line 21
PR20 (bit 22)	0x0	0				Pending bit for line 22

Annex B.7. Configuració dels bits registre EXTI_PR al mètode void intConfigKeyboard (void).